Universal Contracts Approach for Semi-Automatic Verification of ISA Security Guarantees

Anonymous Author(s)

Abstract

Progress has recently been made on specifying instruction set architectures (ISAs) in an executable formalism (for example, in the Sail domain-specific programming language) 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 such that they (1) can be semi-automatically validated against the ISA semantics, producing a 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 on the ISA. Universal contracts can be kept agnostic of software abstractions and they can strike the right balance between providing sufficient detail for reasoning about software and preserving implementation freedom of ISA designers and CPU implementers. We semi-automatically verify such 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 universal contracts method by applying it to two ISAs that offer very different security primitives: (1) MinimalCaps: a simplified custom-built capability machine ISA and (2) a (somewhat simplified) version of RISC-V with PMP. We partially 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.

CCS Concepts: • **Security and privacy → Logic and verification**; **Formal security models**.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Keywords: universal contracts, ISA security, semi-automatic verification, capability safety, RISC-V

ACM Reference Format:
Anonymous Author(s). 2018. Universal Contracts Approach for Semi-Automatic Verification of ISA Security Guarantees. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

An instruction set architecture (ISA) is a contract between software and hardware designers, defining the syntax, semantics and properties of machine code. It is traditionally defined informally in prose in long architecture manuals. Because they are in prose, such ISA specifications can be imprecise, omit details, and offer no way to test/verify advertised guarantees, which is particularly important when talking about security features. A recent trend is to provide formal and executable specifications [3, 10, 14, 15, 21, 27] of ISAs for disambiguation, testability, experimentation and formal study.

For instance, the Sail [3] 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, and is used for the development of the CHERI extensions [33]. Such formal specifications are necessary for formally verifying hardware (processors) and software (compilers, programs written in assembly).

In addition to functional specification of their semantics, ISAs also make meta-theoretical statements of guarantees that programmers rely on, particularly security guarantees offered by the ISA. 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 properties 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. These ISA properties are traditionally also stated in prose, but they should be formalized as well to support reasoning about security-critical code and validating ISA extensions.

We are primarily interested (at least for now) in formalizing the security guarantees which informal ISAs offer about integrity through direct channels. In that sense, our work is

closely related to recent work on validating security guarantees of capability machine ISAs [8, 25] (see Section 8 for a thorough comparison). Less closely related are recent proposals to make ISAs explicit about side-channel leakage Ge et al. [17], Guarnieri et al. [22], for which no guarantees are offered by current ISA specifications, formal or informal. Ultimately, confidentiality guarantees and side-channel leakage will also need to be specified in a security specification for an ISA, but formalizing already existing direct-channel integrity guarantees, as universal contracts, is a first, necessary, step that we address in this paper.

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 and 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 conveniently 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 which is designed to be abstract enough to be validated against extensions or updates of the ISA, but still sufficiently precies for reasoning about code. The method is based on so-called *universal contracts*, which have already been employed successfully for formalizing capability-safety of high-level languages [13, 30, 32], as well as assembly languages [20, 29, 31]. 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 programming language enforces on untrusted programs. The program logic allows to combine manually verified contracts for trusted code with the universal contract for untrusted code and prove properties about the combined program. In addition to formulating the expected security properties of an ISA, it is also important to verify whether ISA instructions correctly enforce these security properties.

In the context of ISAs, universal contracts have so far only been formalized and proven for the capability safety property of simplified capability machine ISAs, and this has been a significant effort [20, 29, 31]. We propose universal contracts as a general approach to formally capture the guarantees of more general security primitives, which we demonstrate by formalizing and proving 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 Private Memory Protection (PMP) extension and synchronous interrupts (*i.e.,* exceptions). In the process we also demonstrate how Katamaran, our semi-automatic separation logic verifier enables automating most boilerplate reasoning in the proofs thus allowing us to focus on the more interesting parts.

To summarize, the contributions of this paper are:

- A general method 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 predicates, lemma invocations, heuristics and includes a basic automatic solver for pure verification conditions. It is implemented and verified in Coq, based on a technical approach that is described in a separate publication. Successful verifications produce machine-checked proofs in an Iris-based program logic that is itself proven sound against the operational semantics of μSail.
- Demonstration of the method for a minimal capability machine, and for a (somewhat simplified) version of RISC-V with the PMP extension. We also present the (partial) verification of a simple example RISC-V program that relies on the PMP guarantees for securing its internal state.

The rest of the paper is structured as follows, Section 2 explains the security primitives used in our case studies, in Section 3 we present 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 our universal contracts approach and case studies in Section 7, where we also demonstrate the (partial) verification of a *femtokernel* relying on the security guarantee we formalized for RISC-V PMP case study. Finally, we discuss related work in Section 8 and conclude the paper in Section 9

2 Background

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

2.1 Capability Machines

Capability machines are a special type of processors that offer capabilities. Conceptually, capabilities are tokens that carry authority to access memory or an object. When capabilities represent software-defined authority, like invoking objects

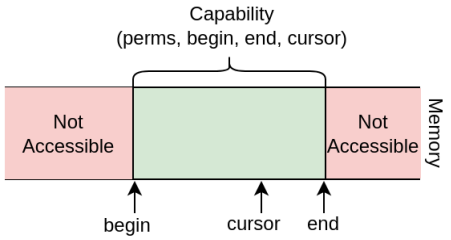

Figure 1. Concept of a capability

or closures, they are referred to as object capabilities. An example of a mature capability machine ISA extension (or family of extensions) is CHERI [33].

Capabilities can be represented as a quadruple, (p, b, e, a), consisting of the permission p of the capability, the begin addresses 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.

The first case study in this paper is a custom-built capability machine we call MinimalCaps, whose semantics specified in Sail has been translated manually to μSail (see Section 4). It contains a minimal subset of instructions from CHERI-RISC-V [33], 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). For simplicity, MinimalCaps does not yet offer a form of object capabilities. We intend to remove this limitation in the near future.

2.2 RISC-V PMP

RISC-V is an open ISA based on established reduced instruction set computing (RISC) principles [4]. Its development is governed by the RISC-V Foundation, which has recently adopted a Sail operational semantics as the official formalization of the ISA.

The RISC-V Privileged Architecture Specification provides the optional Physical Memory Protection (PMP) extension to restrict access to physical address regions [2]. 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 [24].

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 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 permissions of this entry, read only, apply 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 with the PMP extension. The case study itself is a manual translation from the Sail code to μSail, with some additional simplifications:

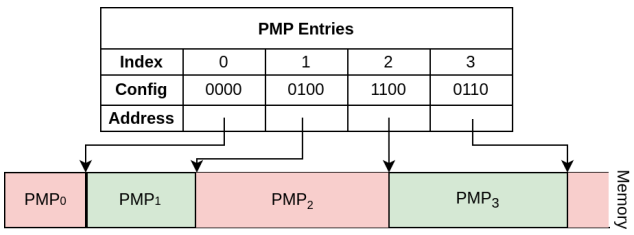

Figure 2. An example RISC-V PMP policy in Top-of-Range mode (TOR).

- Integers are unbounded.
- Only two PMP configuration entries are supported.
- There is no virtual memory.
- 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 programming language security guarantees for high-level languages [13, 30, 32] as well as assembly languages [20, 29, 31]. Essentially, the idea is to formulate the security guarantees offered by a programming language in the form of a contract that holds for arbitrary, potentially untrusted, code, *i.e.,* a universal contract. The contract expresses the restrictions that the programming language enforces on untrusted programs and needs to be proven to hold under the operational semantics of the programming language.

The 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 [28]. Furthermore, separation logic can be used for sequential and for concurrent programs.

We define a universal contract as the security guarantee gained from a Hoare triple over an arbitrary piece of code, where the pre-condition 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. This is because we work with Sail semantics, which essentially defines a definitional interpreter for an 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 rest of this paper, we will use the function name *fdeCycle* to refer to the Sail function implementing this, even though it may be named differently in a particular Sail spec.

The goals of our universal contracts approach are to enable reasoning about the ISA and programs written in it, as well as offering freedom of implementation. More concretely, we want to define universal contracts for security guarantees offered by the specification but leave it sufficiently abstract so the universal contract remains valid under ISA modifications. We propose to specify the universal contracts for an ISA separately from the ISA functional specification, into a *security specification*.

We conjecture that the universal contracts we defined in the case studies we present 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 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 [12, 26]. 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.'s [20] capability safety proof for a simple capability machine with 19 instructions requires about 17kLoc. 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 [5, 6], a new 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, the translation from Sail to μSail has to be performed manually, but we intend to automate it in the future.

Like Sail specifications, 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. Katamaran relies on foreign functions to this end, i.e., functions implemented in Coq of which the signature has been declared in μSailso they are callable. Furthermore, 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 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* [7, 9] 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 (i.e., 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 prove with the full proof automation that Coq provides.

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 [23] 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

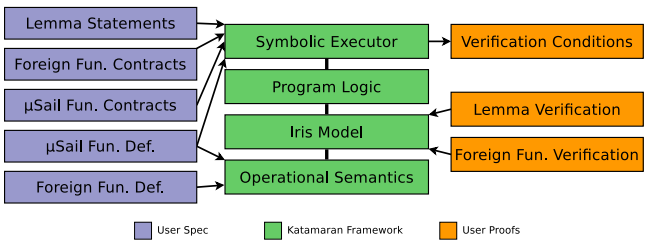

Figure 3. Structure of Katamaran

$$\mathcal{V}(w) \begin{cases} \mathcal{V}(z) & = & True\ (z\ is\ an\ integer) \\ \mathcal{V}(O, -, -, -) & = & True \\ \mathcal{V}(R, b, e, -) & = & \underset{a\in[b,e]}{*} \boxed{\exists w, a \mapsto w * \mathcal{V}(w)} \\ \mathcal{V}(RW, b, e, -) & = & \underset{a\in[b,e]}{*} \boxed{\exists w, a \mapsto w * \mathcal{V}(w)} \end{cases}$$

Figure 4. Logical relations for capability safety

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.

Coen: Perhaps mention that this is a reasonable assumption to make.

5 Capability Safety

In this section we present the universal contract and verification of our first case study, 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* stating 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. in the context of our MinimalCaps machine, a subset of CHERI-RISC-V. Our specification of capability safety is based on that of [13, 19, 20, 30, 32]. Figure 4 shows the logical relation $\mathcal{V}$ that defines the authority of words (*i.e.,* integers and capabilities). The logical relation is defined using separation logic [28], where $*$ is separating conjunction (readers not familiar with separation logic can interpret it as conjunction) and $\mapsto$ the points-to-predicate. A points-to assertion $a \mapsto w$ represents ownership of the memory location at address a and knowledge of its current contents w. The notation $\underset{a\in[b,e]}{*} P$ indicates that P holds separately for all addresses $a \in [b, e]$.

These logical relations express the authority represented by a value or capability, in the form of separation logic predicates that must hold for safely passing the value or capability to untrusted code. The definition says that memory capabilities are safe to pass to an adversary when the addressable locations a are owned by an invariant requiring the word stored at address a to always remain safe. 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, but we refer to existing work for more explanations about that [13, 19, 20, 30, 32].

5.1 Universal Contract

We define our universal contract for this machine with the following machine invariant:

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

Our program logic contains points-to predicates for registers, $r \mapsto w$, describing a single register, named r, with contents w. 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. We can then define our universal contract, specified as a contract over the fetch-decode-execute cycle as in Figure 5.

The universal contract states that the ISA security guarantee offered by the MinimalCaps ISA (capability safety) will produce safe values in the registers for each step of the fetch-decode-execute loop, when starting from a safe configuration. Additionally, the contract also implies that the machine will only use authority that it has access to through the values in the registers.

5.2 Verification

The verification of capability safety in the literature so far requires significant manual effort [20, 29, 31]. In this section, we demonstrate our approach to verifying universal contracts which is semi-automatic.

The contracts for individual instructions require the machine invariant as a precondition and upon successful execution of the instruction, the machine invariant will still hold.

For other functions, the contracts are more specific to what each function does. Consider the contract for the *read_mem* function, which reads the word in memory denoted by the cursor of the given capability. The contract is written as a Hoare triple, $\{P\}$ *read_mem c* $\{r.\, Q\}$, where P is the precondition and if *read_mem c* executes successfully then we bind the result value to the variable r (the identifier before the ".") and Q will hold. Note that we can use the variable r in Q and if we do not use the result variable in the postcondition, we will omit it and simply write $\{Q\}$. The contract of *read_mem* requires the given capability is safe before executing *read_mem*, and guarantees that the capability is still safe afterwards, and that the read word is safe as well:

$$\{\mathcal{V}(c)\}\ read\_mem\ c\ \{w.\, \mathcal{V}(w) * \mathcal{V}(c)\}$$

To give you an idea of how these contracts are verified using Katamaran, Figure 6 shows the μSail implementation

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

Figure 5. Universal Contract for Capability Safety for MinimalCaps

$\{(\exists c.\, pc \mapsto c * \mathcal{V}(c)) * (\forall r \in \text{GPR}.\, \exists w.\, r \mapsto w * \mathcal{V}(w))\}$
store(*rs* : GPR, *rb* : GPR, *immediate* : int) : bool :=
  let *base_cap* := **call** *read_reg_cap rb* **in**
  let (*perm*, *beg*, *end*, *cursor*) := *base_cap* **in**
  let *c* := (*perm*, *beg*, *end*, *cursor* + *immediate*) **in**
  let *w* := **call** *read_reg rs* **in**
  $\{r_0 \mapsto base\_cap * \mathcal{V}(base\_cap) * r_1 \mapsto w_1 * \mathcal{V}(w_1) * \ldots\}$
  lemma *move_cursor base_cap c*;
  $\{r_0 \mapsto base\_cap * \mathcal{V}(base\_cap) * r_1 \mapsto w_1 * \mathcal{V}(w_1) * \mathcal{V}(c) * \ldots\}$
  call *write_mem c w*;
  call *update_pc*;
  true
$\{(\exists c.\, pc \mapsto c * \mathcal{V}(c)) * (\forall r \in \text{GPR}.\, \exists w.\, r \mapsto w * \mathcal{V}(w))\}$

Figure 6. Capability safety for the store instruction.

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 an immediate of the instruction, 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 *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 *rs*.

Next, we use a few lemmas that will modify the precondition so that the contract of *write_mem* is respected, which requires that the given capability that denotes the address to write to is safe and that the word to write to memory is safe. For simplicity we will assume that $rb = R0$, $rs = R1$ and ignore the non-relevant parts of the precondition for this discussion.

The *move_cursor* lemma will generate a $\mathcal{V}$ 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 does not mention the cursor of the capability.

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 and has the write permission. If these checks pass, the given word will be written to the address in memory denoted by the cursor of the capability argument. These 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 would expect, utilizes the *move_cursor* lemma to generate a $\mathcal{V}$ 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. We specify similar contracts for the other instructions, which means that all instructions uphold the machine invariant. We see the contract for the fetch-decode-execute cycle as the universal contract of the ISA, that expresses an authority boundary on (untrusted) code. With the verification of these contracts we can conclude that our MinimalCaps ISA enforces the capability safety security guarantee.

6 Memory Integrity for RISC-V PMP

In this section we present our memory integrity property for RISC-V with exceptions (synchronous interrupts) and the PMP extension.

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 and, after taking a step (*i.e.,* an iteration of the loop), the machine will end up in one of the four following states:

$$\{\mathcal{V}(c)\}\ \mathit{read\_mem}\ c\ \{v.\,\mathcal{V}(v) * \mathcal{V}(c)\}$$
$$\{r \mapsto w\}\ \mathit{read\_reg}\ r\ \{v.\,v = w * r \mapsto w\}$$
$$\{r \mapsto w\}\ \mathit{read\_reg\_cap}\ r\ \{c.\,c = w * r \mapsto w\}$$
$$\{\mathcal{V}(c) * \mathcal{V}(w)\}\ \mathit{write\_mem}\ c\ v\ \{\mathcal{V}(c)\}$$
$$\{pc \mapsto c * \mathcal{V}(c)\}\ \mathit{update\_pc}\ \{\exists c.pc \mapsto c * \mathcal{V}(c)\}$$
$$\{\mathcal{V}(p,b,e,a)\}\ \mathit{move\_cursor}\ (p,b,e,a)\ (p,b,e,a') \hookleftarrow \{\mathcal{V}(p,b,e,a) * \mathcal{V}(p,b,e,a')\}$$

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)

1. **Normal**: the instruction executed fine with no traps occuring, no modified CSRs and didn't recover from a trap, the only CSR modifications that are made are to the *nextpc* and *pc* registers, which are updated to point to the next instruction.
2. **CSR Modified**: this state is reached when executing an instruction that can possibly write values to CSRs. The instruction that can result in this state for our case study is *CSSRW*, which reads the current specified CSR register and then writes a new value to it. Given the limitation that our RISC-V PMP machine only has M- and U-mode, the machine needs to be running in M-mode to modify CSRs (the machine only has CSRs controlled by M-mode), if this is not the case, a trap will occur. Upon successful execution of *CSSRW* we can have any possible value in the CSR specified and the *nextpc* and *pc* will be updated to point to the next instruction.
3. **Trap**: during instruction execution a trap occured, an example of a trap is a memory load violation (for example, a memory read for an address for which the current privilege level has no read permission according to the active PMP entries configuration). Another example is described in the state above, **CSR Modified**. When a trap occurs, the machine transfers control to M-mode with c', the new value in *mcause*, categorizing which kind of trap occured. The *nextpc* and *pc* registers are modified to continue instruction execution from where the trap handler is installed, *h*, which resides in the *mtvec* register. At some point the machine might want to continue executing the program that caused the trap. To achieve this, the address of the instruction causing the trap is stored in *mepc* and the privilege level of when the trap occured is stored in the *MPP* field of the *mstatus* register.
4. **Recover**: the machine can recover from a trap (*i.e.,* returning to where the trap occured) by executing the MRET instruction. The execution of the machine needs to continue from where the trap occured and in the same privilege level. The information required for this is available in the *mepc* and *mstatus* registers. Therefore, the machine sets the *nextpc* and *pc* registers to the value of *mepc* and the *cur_privilege* to the value of *MPP* of the *mstatus* register. The *MPP* value is cleared by setting it to *User*.

The assertions that these states represent consist of pointsto predicates for the registers of the machine, as well as PMP-related assertions that we explain below. The possible states that we can reach are the separation implications prefixed by the ▷ modality, with the exception of **Normal**. Once we end up in one of the states prefixed by the ▷ modality, we consider that the new **Normal** state in which we continue execution of the machine. The ▷ modality allows us to reason

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

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

about the state *after* taking a step in the machine. We verifiy that the universal contract holds manually using our Iris model, relying on the semi-automically verified contracts by Katamaran. More concretely, this means that we manually verify the *fdeCycle*, of the form *call step* ;; *call loop*, where the contract for *step* (which will fetch, decode and execute a single instruction) is semi-automatically verified using Katamaran. The contract of our step function and the execute functions defined for each instruction type (*i.e.,* RTYPE, ITYPE, ...), do not require the $\triangleright$ modality and therefore their precondition only describes the state of the machine at the start of the iteration and the postcondition is a disjunction over the possible states we can reach. This means that these contracts can be semi-automatically verified using Katamaran with the aid of the lemmas we have defined for this case study (some of which are shown in Section 6.1).

PMP_addr_access, one of the predicates used for this security guarantee, 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. We consider two situations for addresses, either there exists some permission p for the address or we have no access to the address at all (*i.e.,* no permissions). If we have a permission for an address a then we will get $\exists p, \mathit{PMP\_access\ a\ entries\ m\ p} \mathrel{-\!\!*} (\exists w, a \mapsto w)$, which we will need for the read and write contracts (which are guarded by requiring p to be at least *Read* or *Write* respectively). If we have no permission for an address a we will end up with $\bot \mathrel{-\!\!*} (\exists w, a \mapsto w)$ and will be prevented from reading or writing from a. We check the permissions with the implementation of the PMP check algorithm in our model, represented by the function *PMP_access*, which takes as arguments an address to check, a, a list representing the PMP entries, *entries*, and the current privilege mode, m. This gives us a pointsto predicate for the address and the word stored at that address, $a \mapsto w$.

$$\mathit{PMP\_addr\_access\ entries\ m} = \underset{a \in addrs}{*} (\exists p, \mathit{PMP\_access\ a\ entries\ m\ p} \mathrel{-\!\!*} (\exists w, a \mapsto w))$$

Figure 9. PMP_addr_access predicate implementation

In Figure 10 we show the contracts for reading from and writing to memory. We ensure that we have read or write access by constraining the permission we get from *PMP_access* to grant us at least read or write permission. We also require that we have access and ownership of the address that we want to read or write from, $a \mapsto w$. The postconditions of these functions give the resources used back and in the case of *write_ram* we get an updated pointsto predicate back, where the address now points to the new value, $a \mapsto v$.

Taking a step in the machine is accompanied by the contract shown in Figure 11 and entails executing a single instruction and updating the *pc* to the value of the *nextpc* register upon successful execution of the instruction. In Figure 11 we make these states more explicit by showing the register file upon reaching such a state. Missing in this figure, focused on the CSRs of the machine, is the predicate for ownernship over the general-purpose registers, *i.e., GPRS*, and predicates capturing ownership of the PMP entries, *PMP_entries*, and *PMP_addr_access*, giving us ownership to the memory locations we can access under the PMP enforced policy. All but one of these predicates is preserved as is upon a state change. The predicate that can be modified, however, is *PMP_entries* in the **CSR Modified** state, allowing to modify PMP entries. For the **CSR Modified** state we require to be running in *Machine* mode, *i.e., p* (the current privilege level at the start of the iteration) has to be equal to *Machine*. In the case of **Trap** we will transfer into *Machine* mode. Finally, we can recover from a trap and end up in the **Recover** state, in which we restore the values in the registers to those when the trap occured.

6.1 Verification

To verify the memory integrity property for RISC-V with PMP we define some lemmas to aid in the semi-automatic verification of our case study. We show interesting lemmas regarding the PMP extension in Figure 12. The first lemma, *open_PMP_entries* opens the *PMP_entries* predicate, so that we gain direct access to the PMP CSRs. This lemma is required for the PMP check algorithm, which needs to be able

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

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

Figure 10. Contracts for functions interacting directly with memory

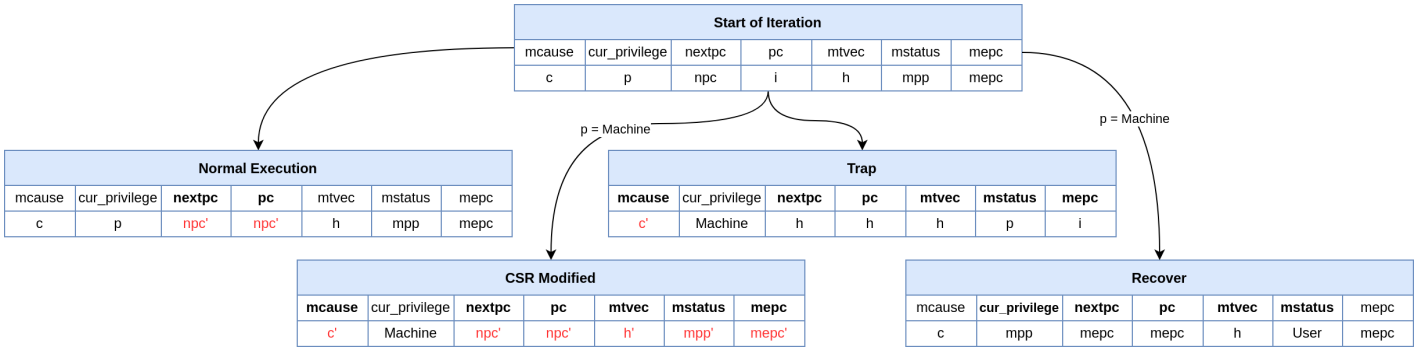

Figure 11. Contract for taking a step on RISC-V (*i.e.,* executing an instruction). New existentially quantified logic variables are shown in red, modified registers are shown in bold. Constraints on the Start of Iteration logic variables are indicated on the arrows (we require for CSR Modified and Recover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

to read the contents of the PMP CSRs. We also define dual lemma for *mathitopen_PMP_entries* called *close_PMP_entries*, where the pre- and postcondition are interchanged so that we can pack the CSRs into the *PMP_entries* predicate. 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 main difference between the PMP CSRs and GPRs is that we are not interested in the contents of the GPRs and only care about the ownership of these registers (which means we always use existentials for the contents of the GPRs). Therefore we define an additional lemma for the PMP CSRs, *update_PMP_entries*, which will only give us a *PMP_entries* predicate for possibly modified PMP CSRs if we are currently running in *Machine* mode.

The two lemmas needed for interacting with memory are *extract_PMP_ptsto* and *return_PMP_ptsto*. *extract_PMP_ptsto* states that if we have a current PMP configuration denoted by *entries*, the points-to predicates for those, given by *PMP_addr_access* knowledge that the address we want to access is a valid address (*i.e.,* between the minimal address 0 and the maximum address *maxAddr*) and the *PMP_access* predicate confirms that we have the requested access permission (*acc*) then we can give points-to predicate back for the address to some word w, $addr \mapsto w$. One other interesting part of the postcondition of *extract_PMP_ptsto* is the magic wand, $-\!\!*$, which says that if we give back the points-to predicate we extracted, we will have full ownership over memory again. This is exactly what happens in the *return_PMP_ptsto* lemma, which requires a points-to predicate for the address and the $-\!\!*$ mentioned above. From this precondition we can get to the postcondition predicate of *PMP_addr_access* by using a lemma that encodes the cancellation rule of separation logic [11]: $P * (P -\!\!* Q) \vdash Q$. These lemmas are proven correct in the Iris model of our case study and are explicity invoked in the function definitions of the case study using ghost statements to aid the semi-automatic verification of the contracts of these functions by Katamaran.

7 Evaluation

In this section we evaluate our semi-automatic approach to universal contract verification.

7.1 Proof effort

In Section 5 and Section 6 we have presented the universal contracts, *i.e.,* the contract for *fdeCycle*, as well as the more interesting contracts of our case studies. A case such as MinimalCaps, which only requires standard separation logic, can be semi-automatically verified with the help of spatial lemmas and pure predicates. Besides the verification of the spatial lemmas and solvers for the pure predicates, no additional proof effort was required, On the other hand, the RISC-V PMP case required the $\triangleright$ modality, for which we currently have no reasoning support in Katamaran, requiring manual verification of contracts relying on the *later* modality. The contracts requiring $\triangleright$ were kept to a minimum, only the *Fetch-Decode-Execute cycle* uses it. The majority of the functions in our case could be semi-automatically verified, therefore the bulk of the proof effort is in defining the lemmas and pure predicates we need and verifying those.

In Table 1 we present statistics for our two case studies. Spatial lemmas aid the semi-automatic verification of the contracts using Katamaran and need to be invoked in the appropriate place in the μSail functions. The case studies both contain a pure predicate that implements a simple permissions lattice. The RISC-V PMP case study requires extra pure predicates that implement the PMP check algorithm. We have defined a solver for all pure predicates.

The majority of the contracts we have specified for the μSail functions can be verified semi-automatically by Katamaran, with only a handful of contracts from the RISC-V PMP case requiring manual effort to validate the contract. Katamaran leaves us with a simplified verification condition that we can continue to prove manually using the built-in Coq tactics. We could, however, not use Katamaran for the

$$\{PMP\_entries\ entries\} \quad open\_PMP\_entries \quad \left\{ \begin{array}{l} \exists cfg0,\ addr0,\ cfg1,\ addr1, \\ (pmp0cfg \mapsto cfg0 * pmpaddr0 \mapsto addr0 * \\ pmp1cfg \mapsto cfg1 * pmpaddr1 \mapsto addr1 * \\ entries = [(cfg0, addr0); \\ \qquad (cfg1, addr1)]) \end{array} \right\}$$

$$\left\{ \begin{array}{l} pmp0cfg \mapsto cfg0 * pmpaddr0 \mapsto addr0 * \\ pmp1cfg \mapsto cfg1 * pmpaddr1 \mapsto addr1 * \\ cur\_privilege \mapsto Machine \end{array} \right\} \quad update\_PMP\_entries \quad \left\{ \begin{array}{l} cur\_privilege \mapsto Machine * \\ PMP\_entries\ [(cfg0, addr0); (cfg1, addr1)] \end{array} \right\}$$

$$\left\{ \begin{array}{l} PMP\_entries\ entries * \\ PMP\_addr\_access\ entries\ p * \\ 0 <= addr * addr <= maxAddr * \\ PMP\_access\ addr\ entries\ p\ acc \end{array} \right\} \quad extract\_PMP\_ptsto \quad \left\{ \begin{array}{l} PMP\_entries\ entries * \\ (\exists w, addr \mapsto w * \\ (addr \mapsto w \mathrel{-\!\!*} PMP\_addr\_access\ entries\ p) \end{array} \right\}$$

$$\left\{ \begin{array}{l} PMP\_entries\ entries * \\ (\exists w, addr \mapsto w * \\ (addr \mapsto w \mathrel{-\!\!*} PMP\_addr\_access\ entries\ p) \end{array} \right\} \quad return\_PMP\_ptsto \quad \left\{ \begin{array}{l} PMP\_entries\ entries * \\ PMP\_addr\_access\ entries\ p \end{array} \right\}$$

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

	MinimalCaps	RISC-V PMP
µSail Fns	48	60
LoC µSail Fns	441	567
Foreign Fns	3	3
Spatial Lemmas	8	7
Lemma Invoc.	34	14
Lemma Proofs LoC	44	70
Pure Pred.	1	4

Table 1. Statistics on the MinimalCaps and RISC-V PMP case studies.

main and *loop* contracts of the RISC-V PMP case study, as the contracts of these functions required reasoning about the ▷ modality, which Katamaran does not support, so we have done a more manual proof using the Iris Proof Mode instead.

The spatial lemmas and foreign function contracts are verified using the Iris Proof Mode.

7.2 Applying the universal contract: femtokernel verification

So far we have focused on the verification of the security guarantees of our universal contracts. In this section we demonstrate that we can use the universal contract for the verification of properties of programs running on top of an ISA. We focus on our RISC-V case, for which such a verification using universal contracts has not been demonstrated to the best of our knowledge.

The program that we verified is a minimal kernel, a *femtokernel*, shown in Figure 13 that installs the interrupt handler and sets up the highest-priority PMP entry to protect our kernel, the interrupt handler and its internal state, a word

```
kernel :  la     ra, adv
          csrrw  pmpaddr0, ra, t0
          lui    ra, max
          csrrw  x0, pmpaddr1, ra
          lui    ra, 0x0
          csrrw  x0, pmp0cfg, ra
          lui    ra, 0xf
          csrrw  x0, pmp1cfg, ra
          la     ra, adv
          csrrw  x0, mepc, ra
          la     ra, ih
          csrrw  x0, mtvec, ra
          lui    ra, 0x0
          csrrw  x0, mstatus, ra
          mret

ih :      auipc  ra, 0
          lw     ra, 12(ra)
          mret
data :    42
adv :     ...
```

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

in memory that has the value 42. Another PMP entry is initialized to give read, write and execute access for programs running in user mode for the rest of memory. The *max* variable refers to the maximum size of memory available on the machine.

The contracts for the kernel and interrupt handler are similar to one another, where the important part is the assertion that the internal state of the kernel, with value 42, will always contain the value 42, *i.e.*, code running in user

mode cannot modify (or even read) the internal state of the kernel. The contracts also require ownership over the GPRs and CSRs, and in the case of the kernel it will, for example, update the PMP entries related CSRs with the boundaries and configuration discussed at the beginning of this section.

We were able to verify the contracts for the basic blocks of the femtokernel, *i.e.,* the kernel and interrupt handler, by reusing existing components of Katamaran. Reiterating the universal contract for the RISC-V case, the PMP related predicates state that we will only get points-to predicates for those addresses we have some permission for. If we have no permission for an address then we will not be able to get a points-to predicate for it. The read and write from and to memory function contracts are guarded by a permission required of at least read and write respectively, which prohibits unauthorized access. This prohibits adversarial code from reading the internal state of the interrupt handler, which cannot be accessed by user mode due to the PMP entries we configure in the kernel.

8 Related Work

Universal contracts for security were, to the best of our knowledge, first used in [13], 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.*[30] 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.*[29] 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 in [20, 29, 31, 32]. They have taken a verification approach that required signification effort to prove that the universal contracts hold, in contrast with our semi-automatic verification approach enabled by Katamaran.

Nienhuis *et al.*[25] 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. [25] is done by Bauereiss et al. [8] 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 they 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. [8], 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 [16] formally verify the correct execution of the CHERI-instructions and liveness properties for the CHERI-RISC-V ISA. In this work they 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 [1], 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.* 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.*[18], 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.*, as a first effort towards verifying the Keystone [24] 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 cycles*, 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, ...) into account. To mitigate that limitation we intend to automate the translation from Sail to μSail, which will allow us to more easily consider realistic ISAs. We intend to improve automation of Katamaran further to reduce the number of lemma invocations that need to be added manually in the μSail functions. For this, we will extend Katamaran with user-provided heuristics that can invoke these lemmas based on what is currently on the heap and what needs to be proven.

References

[1] 2022. *CTSRD-CHERI/Flute: RISC-V CPU, simple 5-stage in-order pipeline, for low-end applications needing MMUs and some performance.* https://github.com/CTSRD-CHERI/Flute

[2] 2022. Specifications - RISC-V international. https://riscv.org/technical/specifications/ Accessed: 2022-04-30.

[3] 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. 2019. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, POPL, Article 71 (Jan. 2019), 31 pages. https://doi.org/10.1145/3290384

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

[5] ANONYMIZED AUTHOR, ANONYMIZED AUTHOR, and ANONYMIZED AUTHOR. 2020. Katamaran: semi-automated verification of ISA specifications. (2020). Extended Abstract.

[6] ANONYMIZED AUTHOR, ANONYMIZED AUTHOR, ANONYMIZED AUTHOR, and ANONYMIZED AUTHOR. 2022. Verified Symbolic Execution with Kripke-Specification Monads (and no Meta-Programming). (2022). Under submission..

[7] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (2018).

[8] Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. 2022. Verified Security for the Morello Capability-enhanced Prototype Arm Architecture. In *Programming Languages and Systems*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 174–203.

[9] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Symbolic Execution with Separation Logic. In *Programming Languages and Systems*. Springer Berlin Heidelberg.

[10] Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Andrew Wright, and Adam Chlipala. 2021. A Multipurpose Formal RISC-V Specification. (April 2021). arXiv:2104.00762 [cs]

[11] Arthur Charguéraud. 2020. Separation logic for sequential programs (functional pearl). *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug 2020), 116:1–116:34. https://doi.org/10.1145/3408998

[12] Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* The MIT Press.

[13] Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about object capabilities with logical relations and effect parametricity. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 147–162.

[14] Shaked Flur, Kathryn E. Gray, Christopher Pulte, Susmit Sarkar, Ali Sezgin, Luc Maranget, Will Deacon, and Peter Sewell. 2016. Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA. In *Principles of Programming Languages*. ACM, 608–621. https://doi.org/10.1145/2837614.2837615

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

[16] Dapeng Gao and Tom Melham. 2021. End-to-End Formal Verification of a RISC-V Processor Extended with Capability Pointers. In *2021 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 24–33.

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

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

[19] Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2021. 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.

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

[21] Shilpi Goel, Warren A. Hunt, and Matt Kaufmann. 2017. Engineering a Formal, Executable X86 ISA Simulator for Software Verification. In *Provably Correct Systems*. Springer International Publishing, 173–209. https://doi.org/10.1007/978-3-319-48628-4_8

[22] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware/Software Contracts for Secure Speculation *(S&P 2021)*. IEEE.

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

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

[25] 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. 2020. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *IEEE Symposium on Security and Privacy (SP)*. 1003–1020. https://doi.org/10.1109/SP40000.2020.00055

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

[27] Alastair Reid. 2017. Who Guards the Guards? Formal Validation of the Arm v8-m Architecture Specification. 1, OOPSLA (Oct. 2017), 88:1–88:24. https://doi.org/10.1145/3133912

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

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

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

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

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

[33] 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. 2020. *Capability Hardware Enhanced RISC Instructions: Cheri Instruction-Set Architecture (Version 8)*. Technical Report. University of Cambridge, Computer Laboratory.