Formalizing, Verifying and Applying ISA Security Guarantees as Universal Contracts

Sander Huyghebaert
Vrije Universiteit Brussel
KU Leuven
Belgium
sander.huyghebaert@vub.be

Steven Keuchel
Vrije Universiteit Brussel
Belgium
steven.keuchel@vub.be

Coen De Roover
Vrije Universiteit Brussel
Belgium
coen.de.roover@vub.be

Dominique Devriese
KU Leuven
Belgium
dominique.devriese@kuleuven.be

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 to (1) balance the needs of ISA implementations (hardware) and clients (software), (2) can be semi-automatically verified to hold for the ISA operational semantics, producing a high-assurance mechanically-verifiable proof, and (3) 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 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 have formalized for RISC-V with PMP.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; • **Security and privacy** → **Formal security models**; **Logic**

and verification; • **Theory of computation** → **Program verification**; *Program specifications*.

KEYWORDS

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

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 [4, 10, 17, 20, 21, 27, 42].

For instance, the Sail programming language [4] 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 [5], and is used for the development of the CHERI extensions [54]. 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 instructions, ISA specifications also make statements about the guarantees they uphold. 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 implementations of the ISA. Similar to the ISA's functional specification, formalizing its security guarantees (rather than just informally specifying them in prose) is vital to support reasoning about security-critical code and validating ISA extensions.

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 code. 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 sufficiently abstract to facilitate validating extensions and updates of the ISA, but still sufficiently precise for reasoning about code. The method is based on so-called *universal contracts* (UCs), which 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: one that does not just apply to some specific code (as usual in software verification) but a contract that applies to arbitrary —including untrusted— code. This universal contract expresses the restrictions that the ISA enforces on untrusted programs. When an ISA specification includes a formalization of security guarantees as a UC, it can be used to reason formally about security-critical software. Manually verified contracts for trusted code can be combined with the universal contract for untrusted code, in order to prove properties about the combined program.

It is essential to verify that the ISA functional specification and security specification are consistent. In fact, whenever the ISA is extended with additional instructions or behavior, it is important that these changes do not break the specified security guarantees, to avoid breaking security of software that relies on them. This means continuously verifying during the evolution of the ISA, that its semantics actually uphold the security properties expressed by the universal contract.

Using universal contracts for generally specifying ISA security guarantees requires (1) a program logic that is sufficiently expressive to express guarantees of a broad range of ISAs, and (2) a verification methodology that can verify the contracts in a semi-automated way against authoritative, engineer-friendly ISA definitions, which are typically implemented as definitional interpreters [e.g., 4, 10]. Universal contracts for ISAs have already been used for capturing the capability safety property of capability machine ISAs [26, 48, 52], and integrity and confidentiality properties of Armv7 [16, 33], but neither approach reconciles these two essential requirements. In the capability machine setting, the universal contracts have a very specific shape: a logical relation is used to define the authority represented by a capability and the universal contract is presented as

the fundamental property of the logical relation. Formulating this logical relation requires a logic that is sufficiently expressive, and as a result, these universal contracts have so far only been proven for theoretical ISAs in a custom-defined but non-authoritative small-step operational semantics, with relatively limited automation. In the Armv7 setting, the universal contracts are defined using a program logic that is simpler but supports more automation, for a real, authoritative ISA. However, the more basic program logic used there does not support the reasoning currently done for capability machines and similar work for capability machines has only been able to establish weaker properties [7, 41]. In this paper, we are the first to reconcile the two requirements: logic expressiveness and verification automation for authoritative, engineer-friendly interpreter-style ISA definitions. More generally, we are the first to propose a general approach to formalizing, verifying and applying the guarantees of more general ISA security primitives.

To achieve this, we use a program logic that allows concise, compositional but expressive proofs and we provide a powerful verification tool called KATAMARAN to support this: a semi-automatic separation logic verifier for SAIL that automates most boilerplate reasoning in the proofs, and allows focusing on the more interesting parts. KATAMARAN symbolically executes code in μSAIL (a core calculus of SAIL) to verify general program logic contracts and includes a generic solver for pruning unreachable branches and discharging pure *verification conditions* (VCs). It is implemented in the CoQ proof assistant and comes with a full soundness proof following a general approach [32]. This CoQ soundness proof provides very high assurance directly against the μSAIL operational semantics. Additionally, taking inspiration from related work [46], KATAMARAN doubles as a verification tool for the ISA's assembly language, essentially by treating a contract for an assembly program P as a contract for the ISA semantics under the assumption that it is executing program P, following related work [46]. Compared to other foundational verifiers like Diaframe [39], Lithium [45], MoSeL [34] or Bedrock [15], KATAMARAN is in a sense a *verified* verification tool rather than a *verifying* one, meaning that it is implemented in Gallina and does not rely on CoQ and meta-programming tactics to manage and manipulate intermediate assumptions and VCs. This approach has benefits for performance and allows extracting the verifier from Gallina to OCaml. Only a few other tools in the literature take the same approach [3, 30] and they are significantly less practical and mature than KATAMARAN.

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 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 KATAMARAN in a semi-automated approach that improves over the more manual efforts employed so far. For now, we manually translate the SAIL semantics to our internal core calculus μSAIL and we somewhat simplify the ISA, but the simplifications are minor. In particular, we fully support bounded integers and byte-addressed memory. The remaining restrictions include assumptions on the amount of PMP configuration registers, the allowed modes for PMP

configurations and similar. The verification tool itself is fully verified, so we obtain high-assurance guarantees in terms of the μSAIL semantics.

Note that, for now, we only formalize 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 [7, 41] (see Section 7 for a comparison). Thus, we solve a different problem than recent proposals to make ISAs explicit about side-channel leakage [24, 28], for which no guarantees are offered by current ISA specifications, formal or informal.

To summarize, the contributions of this paper are:

- A novel, 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 and includes an automatic solver for pure VCs. It is implemented and verified in COQ, based on a general approach described elsewhere [32]. Successful verifications produce machine-checked proofs in an IRIS-based program logic that is itself proven sound against μSAIL's operational semantics.
- A demonstration of the method for two case studies: (1) a minimal capability machine that is a subset of CHERI-RISC-V [54] and (2) the official formal SAIL semantics of RISC-V with the PMP extension, with minor simplifications.
- An evaluation of the required effort to validate a UC security guarantee against the operational semantics of an ISA, based on statistics about our two case studies. We measure the 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 [26].
- 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. The verification 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. In Section 3 we introduce universal contracts by formalizing universal contracts for two ISAs. In Section 4 we discuss our new semi-automatic logic verifier, KATAMARAN. We outline and evaluate the verification effort of our universal contracts in Section 5. In Section 6 we demonstrate the verification of a femtokernel relying on our RISC-V PMP universal contract. Finally, we discuss related work in Section 7 and conclude in Section 8.

2 BACKGROUND

In this section we give a brief introduction to separation logic and cover the security primitives we use in our case studies: capabilities and physical memory protection.
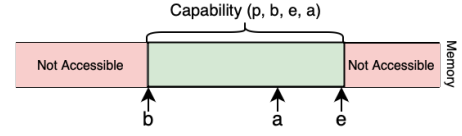

Figure 1: Concept of a capability

2.1 Separation Logic

Program logics are formal frameworks for proving properties about programs. Hoare logic uses Hoare triples { *PRE* } P { *POST* } to express a contract for program P. The logical predicate *PRE* expresses a precondition on the initial state of the system (e.g., local variables, heap memory, etc.). If this precondition is satisfied, then program *P* is guaranteed to run correctly (e.g., does not get stuck or crash) and if it terminates, the final machine state will satisfy postcondition POST. If program P returns a result, the postcondition can be written as $\{r. POST\}$ to express guarantees about the result value r, e.g. { *True* } return 42 $\{r. isEven(r)\}$. Separation logic is similar to Hoare logic, but pre- and postconditions are not expressed in standard propositional logic, but in a logic where assertions may also express exclusive ownership of (or authority over) shared mutable state of the system, such as the heap. This enables more modular reasoning, i.e. contracts for program components can more easily be combined. In separation logic, the points-to predicate $l \mapsto v$ represents ownership of a resource l and knowledge of its current value v. In this paper we will use the points-to predicate for memory locations and registers. Assertions like $l \mapsto v$ can be combined by separating conjunction, denoted as $P * Q$, expressing that P and Q are both true and that they claim ownership of disjoint subsets of the shared state. The separating implication operator $P \twoheadrightarrow Q$ (affectionately referred to as the *magic wand*) requires that Q holds when authority for the premise P is presented. The separation logic we use in Katamaran is based on IRIS [31], which makes the logic particularly expressive. For example, predicates can be defined recursively (under the restriction of guardedness), they can express authoritative knowledge over advanced forms of so-called ghost state and invariants shared between multiple threads. Also, a contract { *PRE* } P { *POST* } additionally expresses that invariants remain true at every atomic step of execution.

2.2 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 [53, 54]. Conceptually, capabilities are tokens that carry authority to access memory or an object. On CHERI, a memory capability extends a traditional pointer with more information such as, amongst others, the bounds and permissions.

Capabilities can be represented as a quadruple, (p, b, e, a), consisting of the permission p of the capability, the begin and end addresses b and e, and a cursor a. Permissions on a capability machine can include: the null permission O, the read permission R,

Sander Huyghebaert, Steven Keuchel, Coen De Roover, and Dominique Devriese

and the read and write permission RW. Figure 1 illustrates a capability's range of authority $[b, e]$ and cursor a, pointing to a current memory location. A special case is the permission E, which models enter capabilities[1] [12]. 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, but the caller cannot access their private data and capabilities. As such, they set up 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 studied in this paper is a custom-built capability machine we call MINIMALCAPS, which supports memory and object capabilities. It contains a subset of instructions from CHERI-RISC-V [54], including branching, jumping, and arithmetic instructions. The instructions are a superset of what is supported in Cerise [25, 26, 52]. A word on the machine is either an integer or a capability and these can be stored in general-purpose registers (GPRs) and in memory.

2.3 RISC-V PMP

The Physical Memory Protection (PMP) extension of RISC-V allows to restrict access to physical address regions [44]. RISC-V defines three privilege levels: User, Supervisor and Machine (the first two are optional). 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 [35].

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 brevity, 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) 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 PMP entry specifies a configuration, which for our purposes consists of 4 bits $LRWX$, where L defines whether the PMP entry is locked and RWX stands for Read, Write and Execute respectively. The policy in Figure 2 grants read-only access to User and Supervisor mode (U- and S-mode) in PMP entry 1 and read-write access in entry 3. PMP entry 2 is locked, indicating

---

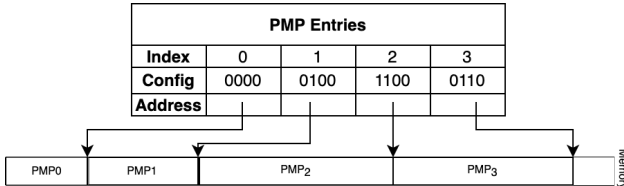[1]Also known as sentry capabilities in the context of CHERI [54]

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

```
function fdeCycle() = {          function fdeStep() =
    fdeStep();                       let w = fetch(PC) in
    fdeCycle()                       let i = decode(w) in
}                                    execute(i)
```

Figure 3: *fdeCycle* **and** *fdeStep* **definitions as found in** SAIL **specifications.**

that its *read-only* permission applies to M-mode (machine mode) as well. 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. By default, M-mode has full permissions over memory while U-mode and S-mode have no permissions. Non-locked PMP entries grant permissions to U-mode and S-mode. A locked PMP entry revokes permissions in all modes including M-mode. Such an entry can only be modified by resetting the system, *i.e.,* one cannot write to the associated configuration and address register (and in the case of TOR, 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 LRWX bits determine whether the access succeeds or fails, otherwise 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: 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.

2.4 ISA specifications in Sail

When using SAIL an ISA semantics is defined through a definitional interpreter for the ISA's assembly language. This includes the *Fetch-Decode-Execute cycle* of the ISA. In the remainder of this paper, we will use the function name *fdeCycle* to refer to the SAIL function implementing this cycle, although it may be named differently in a practical SAIL specification. Furthermore, SAIL specifications model memory as part of the global state, as well the ISA's registers. In Figure 3 we sketch a typical implementation of *fdeCycle*: it indefinitely recurses on invoking the *fdeStep* function, which fetches, decodes and executes the current instruction.

To give a better intuition for the *execute* function, we show the specification of the *store* instruction in Figure 4. SAIL models typically define *execute* as a *scattered definition* that pattern matches on its first argument, the instruction to be executed. Figure 4 shows such a *clause* for *store*. The *store* instruction carries a source register containing the value we want to write into memory, a base register containing the capability to write to memory, and an immediate offset. The clause performs the write to memory by first reading the capability from register *rb* and deriving a capability *c* from *bc* with its cursor incremented by the immediate. We assert that the capability has the write permission and then continue by reading

```
function clause execute(store(rs, rb, immediate)) = {
    let bc = read_reg_cap(rb);
    let c = {bc with cap_cursor = bc.cap_cursor
              + sail_sign_extend(immediate, integer_size)};
    assert(writeAllowed(c.cap_permission));
    let w = read_reg(rs);
    write_mem(c, w);
    update_pc();
}
```

Figure 4: Sail specification of the store instruction of our capability machine (simplified).

the contents of *rs* and writing it to memory. Finally, the program counter gets updated.

3 SECURITY GUARANTEES AS UNIVERSAL CONTRACTS

In this section, we explain our novel, general method based on universal contracts in more detail by considering the examples of our capability machine, MinimalCaps and RISC-V PMP.

In our approach, UCs are formulated using separation logic, as introduced in Section 2. Thanks to separation logic, the UCs only have to consider the part of the state an adversary has access to, and can be expected to scale to concurrency which we want to take advantage of in the future. A contract $\{Pre\}$ $code$ $\{r.\,Post\}$ usually applies only to specific *code*. What we call a universal contract, is one that applies to any choice of *code*, rather than only specific ones. Such a contract specifies guarantees that are enforced by the semantics for arbitrary or untrusted code.

While universal contracts apply to arbitrary assembly code, they take a slightly different form in our setting. The arbitrary programs that our contract applies to take the form of arbitrary instructions encoded in memory. Our universal contract is thus defined as a Hoare triple over the *fdeCycle*.

3.1 Capability Safety for MinimalCaps

The security guarantee offered by a capability machine is capability safety, a property which expresses bounds on the authority of arbitrary untrusted code. More intuitively, capability safety ensures that arbitrary code can only access the resources and operations for which it has been granted explicit access. Syntactic properties like capability monotonicity [37], which expresses that the transitively available authority cannot increase during execution, have been shown insufficient in the presence of object capabilities [18]. Therefore, we use a semantic formulation of capability safety based on logical relations [18, 25, 26, 48, 50, 52] and formulate the property as a contract over the *fdeCycle*, following Cerise [25, 26, 52]. The contract states that if we start from a configuration of safe values, arbitrary code will not be able to exceed the authority of those values.

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

$$\mathcal{E}(w) = \begin{cases} \left(\text{pc} \hookrightarrow w * \underset{r \in \text{GPR}}{\text{\Large$*$}} (\exists w.\, r \mapsto w * \mathcal{V}(w)) \right) \, {-\!\!*} \\ \qquad\qquad\qquad\qquad\qquad \text{wp } fdeCycle() \, \{ \, True \, \} \end{cases}$$

Figure 5: Logical relations for capability safety

$$\left\{ \begin{matrix} (\exists c.\, (pc \mapsto c) * \mathcal{V}(c)) * \\ {\Large*}_{r \in \text{GPR}} (\exists w.\, r \mapsto w * \mathcal{V}(w)) \end{matrix} \right\} \, fdeCycle() \, \left\{ \, True \, \right\}$$

Figure 6: Universal Contract for Capability Safety for MinimalCaps.

Figure 5 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 [43], where 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, wp $fdeCycle()$ $\{$ $True$ $\}$. The standard Iris predicate wp $fdeCycle()$ $\{$ $True$ $\}$ represents the *weakest precondition* for $fdeCycle()$ to execute correctly without breaking any invariants. It can be alternatively interpreted as $\exists Pre, Pre * \{Pre\}$ $code$ $\{Post\}$: some precondition Pre needs to hold that is sufficient to guarantee postcondition $Post$ after executing $code$ [13].

First-time readers may ignore Iris's always modality ($\square$), which requires that the authority does not depend on exclusive ownership of resources, and Iris's later modality ($\triangleright$), which justifies the cycle in the definition of $\mathcal{V}$. We refer to prior work for more explanation [e.g., 26].

The universal contract for MinimalCaps is a contract for the fetch-decode-execute cycle, depicted in Figure 6. It asserts that if the machine is executed (*i.e.*, *fdeCycle()* is invoked) with authorized capabilities in pc and general-purpose registers, then it will execute correctly and not break any invariants. The postcondition $True$ is trivial, as before when we used wp $fdeCycle()$ $\{$ $True$ $\}$. Indeed,

fdeCycle() is an infinite loop, so the postcondition is not very relevant anyway. Nevertheless, even with an irrelevant postcondition, our contracts still express the preservation of invariants during execution (as we will see in Section 6).

As demonstrated before [25, 26, 52], 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[2], 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 never break established invariants.

3.2 Memory Integrity for RISC-V PMP

Universal contracts can be applied beyond the specific setting of capability machines. While other work has already employed universal contracts for specific non-capability machines, namely Armv7 [16, 33], we provide evidence that our method and our tool KATA-MARAN are general and can be used to capture security guarantees of different security primitives. To this end we apply our approach to RISC-V with support for exceptions (synchronous interrupts) and the PMP extension, as explained in Section 2. The universal contract captures 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 some minor simplifications: only two (rather than 16 or 64) PMP configuration entries in top-of-range (TOR) mode are supported, no virtual memory, and only M-mode and U-mode (no S-mode).

We define the universal contract for this machine, over the fetch-decode-execute cycle as shown in Figure 7. 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 l, a predicate that represents ownership of memory accessible according to the PMP policy entries at the current privilege l (discussed further below).

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).

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

---

[2]In fact, IRIS invariants can also express protocols on private state [31].

$$\left\{ \begin{array}{l} Normal(l, h, mpp, entries) \\ *\triangleright(CSR\ Modified(l, entries) \twoheadrightarrow wp\ fdeCycle()\ \{\ True\ \}) \\ *\triangleright(Trap(l, h, entries) \twoheadrightarrow wp\ fdeCycle()\ \{\ True\ \}) \\ *\triangleright(Recover(l, h, mpp, entries) \twoheadrightarrow wp\ fdeCycle()\ \{\ True\ \}) \end{array} \right\}$$
$$fdeCycle()\ \{\ True\ \}$$

$$Normal(l, h, mpp, entries) =$$
$$\left\{ \begin{array}{l} (\exists i.\ pc \mapsto i) * cur\_privilege \mapsto l * mtvec \mapsto h * \\ (\exists c.\ mcause \mapsto c) * mstatus \mapsto [mpp] * \\ mepc \mapsto mepc * PMP\_entries\ entries * \\ \displaystyle\mathop{\scalebox{1.5}{$*$}}_{r \in GPR}\ (\exists w.\ r \mapsto w) * PMP\_addr\_access\ entries\ l \end{array} \right.$$

$$Trap(l, h, entries) =$$
$$\left\{ \begin{array}{l} pc \mapsto h * cur\_privilege \mapsto Machine * mtvec \mapsto h * \\ (\exists c.\ mcause \mapsto c) * mstatus \mapsto [l] * \\ (\exists c.\ mepc \mapsto c) * PMP\_entries\ entries * \\ \displaystyle\mathop{\scalebox{1.5}{$*$}}_{r \in GPR}\ (\exists w.\ r \mapsto w) * PMP\_addr\_access\ entries\ l \end{array} \right.$$

Figure 7: Universal Contract for Memory Integrity for RISC-V with PMP.

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 8. 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, so it cannot be accessed.

3.3 Verifying and applying a Universal Contract

Formalizing security guarantees is useful for two purposes.

First, it can be used to verify whether the security guarantees are consistent with the ISA's operational semantics, *e.g.,* whether all instructions correctly check the (PMP or capability-based) policy before accessing memory. This verification is a non-trivial effort

$$PMP\_addr\_access\ entries\ m =$$
$$\mathop{\scalebox{1.5}{$*$}}_{a \in addrs}((\exists p, PMP\_access\ a\ entries\ m\ p) \twoheadrightarrow \exists w, a \mapsto w)$$

Figure 8: PMP_addr_access predicate implementation

that should be repeatable during the evolution of the ISA, so we provide tool support in the form of Katamaran, a semi-automatic verification tool for Sail, discussed in Section 4. We discuss verifying the UCs for MinimalCaps and RISC-V PMP in Section 5.

Secondly, the UC can be used to verify the security of trusted software interacting with untrusted software. If such software is verified against the UC, its security will hold in any implementation of the ISA that respects the UC. Such a verification entails verifying trusted code using the ISA operational semantics and invoking the UC when a jump to untrusted code happens. We demonstrate in Section 6 that our UC supports this using the *femtokernel* case study: a minimal RISC-V PMP machine mode kernel that interacts with untrusted user-mode code over a simple system call. Interestingly, this verification reuses Katamaran as a verification tool for RISC-V assembly, by reusing an idea from previous work [46].

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. [25]'s capability safety proof for a simple capability machine with 19 instructions requires about 7kLOC of Coq proofs. 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 [32]. 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.[3] 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 static symbolic execution* [6, 8] of the function bodies. During the execution, Katamaran tries to discharge proof obligations automatically using solvers and leaves residual VCs for the user where this fails. The library contains a *generic solver* for some of μSail's background theory, which can be complemented by a *user solver* for user-defined predicates. 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 VCs will be in first-order predicate logic, which the user can discharge using Coq's built-in proof automation.

Katamaran's symbolic executor is implemented as a monadic interpreter in a specification monad [2, 38, 49]. Such specification monads allow angelic and demonic non-determinism which we use to explore the execution paths of Sail programs. The specification monad is implemented as a predicate transformer, which combines assertions and assumptions encountered on different execution paths into a verification condition. The resulting verification conditions are not regular Coq propositions but rather ASTs in a syntactically represented language of propositions, allowing us to simplify them and prune unreachable paths during verification. Additionally, we use Kripke indexing techniques to track logical variables in scope and path constraints [32]. This enforces that path constraints are monotonically increasing along execution paths and hides the plumbing.

A question that arises is whether the generated VCs 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 9. The contracts of both kinds of functions and the code of the μSail functions are inputs to the symbolic executor from which it produces VCs. Following the recipe proposed by [30, 32], we first reduce the soundness of the symbolic executor to a shallow executor: a monadic interpreter with the same structure as the symbolic executor but written in a specification monad over propositions of the meta-logic. A binary Kripke logical relation [32] links the constituents of both interpreters, and allows
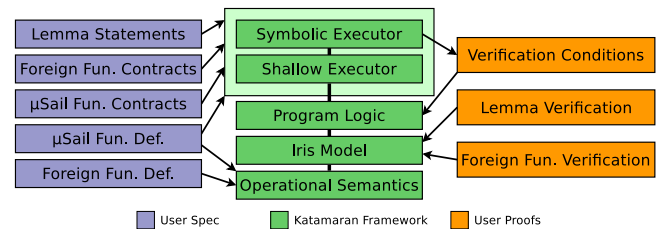

Figure 9: Structure of Katamaran

---

[3]Sail's existing Coq backend only translates to a shallow embedding.

Table 1: Katamaran lines of code calculated by `coqwc`.

Component	Spec	Proof
μSail syntax and semantics	1939	547
Symbolic executor	3102	2396
Background theory solver	786	410
Shallow executor	536	762
Program logic	1226	1058
Iris model	596	679
Other	2609	1325
Total	10794	7177

us to conclude that symbolic VCs imply shallow VCs. A second soundness proof connects this to an axiomatic program logic: given proofs of the shallow VCs the function bodies are also verifiable in the program logic.

The program logic consists of separation logic contracts $\{ PRE \} \, c \, \{r. \, POST\}$. We assign meaning to these contracts using the Iris separation logic framework [31]. 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 contracts to the operational semantics: every contract that holds semantically implies partial correctness. This is sufficient for us; we assume it is verified separately that the machine cannot get stuck.

An overview of the sizes of different parts of Katamaran can be found in Table 1. Even though we run Katamaran within Coq's typechecker[4], it is fast enough for interactive experimentation with definitions in our case studies, with immediate verification of the corresponding contracts. For instance, the longest runs of the symbolic executor are the verification of the femtokernel blocks that we describe in Section 6, where we symbolically execute μSail code to derive a symbolic executor for assembly code. Both blocks combined take 1.77s to symbolically execute (on an Intel i7-12700) and in total 247 calls of μSail and foreign functions are executed of which 105 calls are executed by interpreting function contracts and 142 are executed by interpreting function bodies.

5 VERIFICATION OF UNIVERSAL CONTRACTS USING KATAMARAN

Using Katamaran, we have verified that the two universal contracts from Section 3 are consistent with the operational semantics of the ISAs. In this section, we explain the two verifications and evaluate the proof effort involved.

---

[4]Extracting or natively compiling the code is possible in theory, but currently the overhead outweighs the benefits.

5.1 MinimalCaps

The verification of capability safety in the literature so far has required significant manual effort [26, 48, 52]. In this section, we demonstrate our semi-automatic approach.

The contract for *fdeCycle()* iterates the following contract for *fdeStep()*, which executes a single FDE cycle.

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

This internal contract requires an induction hypothesis IH:

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

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. Some other helper functions are given more specific contracts.

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 at least read permission, and guarantees that the capability is still safe afterwards, as well as the word read from memory:

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

To give an idea of how these contracts are verified using Katamaran, Figure 10 shows the μSail implementation of MinimalCaps' store instruction, with verification annotations in red (not part of the code itself), and Figure 11 displays the contracts for the functions used in the implementation. This instruction takes 3 arguments: source and target GPRs *rs* and *rb* and an integer *immediate*. The instruction will write the value of *rs* to *cursor + immediate*, if register *rb* contains a capability with this *cursor*.

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 write word w to memory. We use a few lemmas to modify the precondition in order to satisfy the precondition of *write_mem*, which requires authority for destination capability c and word w. Note that lemmas are proven sound in the Iris model for the case study using Iris Proof Mode. For simplicity, we assume that $rb = R0, rs = R1$ and ignore the non-relevant parts of the precondition for this discussion.

The *move_cursor* lemma produces a $\mathcal{V}$ predicate for c based on the one for bc, which differs only in the cursor field. 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 $perm \neq$ E from RW $\leq_p perm$.

$$\{(\exists c.\, pc \mapsto c * \mathcal{V}(c) \underset{r \in \mathrm{GPR}}{\textstyle \bigstar} (\exists w.\, r \mapsto w * \mathcal{V}(w))\}$$

$store(rs : \mathrm{GPR}, rb : \mathrm{GPR}, immediate : \mathrm{int}) : \mathrm{bool} :=$

  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 $update\_pc$;

$$\{(\exists c.\, pc \mapsto c * \mathcal{V}(c)) * \underset{r \in \mathrm{GPR}}{\textstyle \bigstar} (\exists w.\, r \mapsto w * \mathcal{V}(w))\}$$

Figure 10: Capability safety for the store instruction (slightly simplified).

The $write\_mem\ c\ w$ call checks that c's cursor is within bounds but assumes it has the write permission. If all checks pass, word w will be written to this address. The checks are critical for capability safety of MinimalCaps and the machine will go into a failed state if they 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 utilizes the $move\_cursor$ lemma again to generate a $\mathcal{V}$ predicate for the updated pc.

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

Figure 11: Contracts for functions and lemmas used in exec_sd (for registers r, values v and w and capabilities c)

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

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

Figure 12: Contracts for functions interacting directly with memory

Arriving at the end of the $exec\_sd$ 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 $fdeCycle()$. The contract for the $fdeCycle()$ itself is proven manually due to the use of Iris's later modality and Löb induction, which Katamaran does not (yet) support. Note that in the proof of the contract of $fdeCycle()$, we can use the semi-automatically proven contracts, *i.e.,* we don't need to reason about $fdeStep()$ manually in the body of $fdeCycle()$. The $fdeCycle()$ 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.

5.2 RISC-V PMP

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 12. 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 universal contract proof iterates a contract for the single-cycle $fdeStep()$, depicted in Figure 13. It specifies that executing an instruction will leave the CPU in one of the states Normal, CSRModified, Trap or Recover, mentioned earlier, with specific values for the ISA registers. Not shown are the predicates for ownership over the general-purpose registers, *i.e.,* $\bigstar_{r \in \mathrm{GPR}} (\exists w.\, r \mapsto w)$, 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.,* $p = Machine$. 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 use some interesting lemmas shown in Figure 14. 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
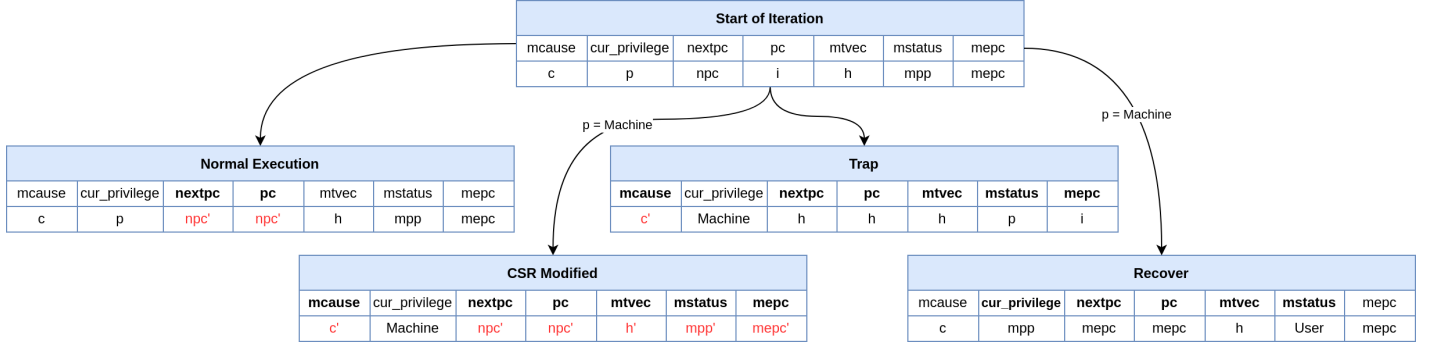
Figure 13: Contract for taking a step on RISC-V (*i.e.,* executing an instruction). New existentially quantified logic variables are shown in red, modified registers are shown in bold. Constraints on the Start of Iteration logic variables are indicated on the arrows (we require for CSR Modified and Recover state transitions that we started from a state running in Machine mode, *i.e.,* $p = Machine$).

Figure 14: Contracts for lemmas used in RISC-V PMP case study.

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 points-to predicate 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 points-to predicate. All these lemmas are proven correct in the Iris model 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.

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 the *fdeCycle()* and the lemmas used in the case study manually.

5.3 Evaluation

In this section we evaluate our semi-automatic approach to universal contract verification. Our aims for the approach are that universal contracts should be agnostic of software abstractions and verified against the operational semantics of ISAs. Furthermore, we want to minimize the effort to re-verify a universal contract for a modified ISA. We evaluate the proof effort required in our case study absolutely as well as relatively to Cerise [25, 26, 52].

Cerise is close to our MinimalCaps case study because it establishes a very similar formulation of capability safety in an expressive, Iris-based program logic. However, they work for a simpler but otherwise similar capability machine ISA with a small-step operational semantics. In addition to the universal contract which is formulated as a logical relation and associated "fundamental

theorem", they also prove functional specifications for instructions which they use to verify example software. The lines of code in both developments should be interpreted and compared with caution, because the approaches are quite different (e.g. different style of defining semantics) and because Cerise's functional specifications and verified examples do not have analogues in the MinimalCaps case study. The Cerise proofs are large manual Iris Proof Mode proofs, with limited automation.

Table 2 presents statistics on our two case studies and some relevant statistics for Cerise [25, 26, 52]. 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 a subset of CHERI-RISC-V. 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 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

| | Sail LoC | Oper. Sem. | Generatable | | | | | Specs LoC | | | Proofs LoC | | | Model and Solver | | | Universal Contract LoC | Total LoC (no μSail or Oper. Sem.) |
			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		
MinimalCaps	571	-	1096	52	3	9	39	397	47	98	120	65	70	350	3	82	172	1508
RISC-V PMP	-	-	2025	66	4	8	16	883	61	89	226	150	390	167	9	178	215	2462
Cerise	-	1190	-	-	-	142	-	-	-	2318	-	-	2918	1351	-	-	-	6729

Table 2: 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 [25]), 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 to 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 the Sail LoC as separate from our case studies and therefore do not include it the total at the right of the table.

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 120 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 VCs 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. We specified a few pure predicates and report on the LoC for the solver. Finally, we were able to validate our universal contracts in 172 LoC (*i.e.,* reasoning about the contract for the *fdeCycle*) for MinimalCaps and 215 LoC for RISC-V PMP. We end Table 2 with the total LoC for the case studies, not including the μSail specification (or operational semantics for Cerise), 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 Minimal-Caps 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 to Cerise, so for a meaningful discussion, we tried to gather statistics in a way that is maximally fair, but the reader should keep in mind that this comparison is not entirely fair. For example, although we do not count Cerise proofs that are unrelated to the universal contract, the proofs do 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 to our μSail specification. More interesting are 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 uninteresting parts, leaving us with a smaller amount of proof code for lemmas that are directly related to capability safety. The Iris model is already partially instantiated in the Katamaran codebase, making the MinimalCaps LoC for this part smaller than that of Cerise.

6 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 our universal contracts are strong enough to support 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 arguably already been demonstrated [25, 52]. 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 the technique, consider the minimal *femtokernel* in Figure 15, which configures the PMP extension to 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*, adv) and [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 from the first region

Sander Huyghebaert, Steven Keuchel, Coen De Roover, and Dominique Devriese

```
 1:  kernel:   la       ra, adv
 2:            csrrw    x0, pmpaddr0, ra
 3:            li       ra, max
 4:            csrrw    x0, pmpaddr1, ra
 5:            li       ra, 0xf00
 6:            csrrw    x0, pmp0cfg, ra
 7:            la       ra, ih
 8:            csrrw    x0, mtvec, ra
 9:            la       ra, adv
10:            csrrw    x0, mepc, ra
11:            csrrw    x0, mstatus, x0
12:            mret
13:  ih:       auipc    ra, 0
14:            lw       ra, 12(ra)
15:            mret
16:  data:     42
17:  adv:      . . .
```

Figure 15: The femtokernel sets up the PMP entries to protect itself, the interrupt handler and its internal state.

and grants read, write and execute permissions to user mode to the second region (lines 5–6). Both entries are unlocked so machine mode code can also access the first region. The kernel then installs its handler (lines 7–8) and jumps to the adversary in user mode (lines 9–12) by loading the adversary address into the *mepc* register (lines 9–10), clearing the *mstatus* register (line 11), *i.e.,* setting the *MPP* field to user mode, and performing the jump (line 12). 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. Our IRIS model supports a general notion of invariants, so we can register this property as an invariant (and in fact, we do the same for the memory storing the femtokernel instructions):

$$inv_{femto} = \boxed{data \mapsto_{word} 42} \qquad \text{for data the label from Fig. 15.}$$

Ownership of the remaining memory is sufficient to establish PMP_addr_access entries User for the PMP configuration (*entries*) set by the femtokernel initialization code. As a result, we can invoke the universal contract to establish safety of the jumps to *unknown user-mode code* at the end of the initialization and exception handler. It then remains to prove contracts for the kernel and interrupt handler, establishing that they jump to user-mode code in a correct state (*i.e.,* in user-mode, with the intended PMP configuration) and use but don't break the registered invariants.

Inspired by Islaris [46], we can largely automate the remaining verification, by reusing existing components and proofs of KATAMA-RAN to derive a sound verifier for known assembly code. Essentially, the idea is that a contract for {*Pre*} *B* {*Post*} for a basic block of assembly code can also be regarded as a contract for the ISA semantics, under the assumption that it is looking at basic block *B*. Essentially, to verify a contract for a basic block of assembly instructions {Pre_0} $\overline{instr_{0..n}}$ {$Post_n$}, we iteratively verify *fdeStep()*

contracts that look roughly like this:

$$\{Pre_i * \text{pc} \mapsto a * a \mapsto c * decode(c) = instr_i\} \ fdeStep()$$
$$\{Post_i * \text{pc} \mapsto a * a \mapsto c\}$$

We are thus able to verify the contracts for the basic blocks of the femtokernel, *i.e.,* the initialization and handler code, leaving only some manual proofs to register invariants for code, data, and adversarial memory and invoke the universal contract.

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, which we list for reference in Appendix A.

7 RELATED WORK

Universal contracts have been used to capture security properties of capability-based high-level languages [18, 50] and capability-based ISAs [25, 25, 26, 48, 51]. Our formalization of capability safety for MINIMALCAPS is close to the one in Cerise [25, 25, 26]. Some versions of Cerise support additional features like local or uninitialized capabilities. They use a verification approach that requires significant effort to prove that the universal contracts hold, in contrast to our semi-automatic verification approach enabled by KATAMARAN.

Nienhuis et al. [41] prove *reachable capability monotonicity* up to security domain transitions 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 *capability safety* security property we formulate as a universal contract can be used in the verification of programs to be executed on the ISA. It has not been shown yet that capability monotonicity up to security domain transitions is a strong enough property to perform such full-system proofs. Second, the approach taken differs from ours in that they express their security property in the meta-logic directly and automate the *boring* parts of the proof away with standard automation, like tactics and auto-generated proof scripts, whereas we use an embedded separation logic and provide our semi-automatic logic verifier, KATAMARAN, based on symbolic execution. Our more abstract description of the security property should be more future proof against ISA modifications and extensions. For example, in our universal contracts it does not 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. [7] 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 reachable capability monotonicity property (up to domain transitions), the authors define four properties of arbitrary CHERI instruction execution and use that to verify a concrete implementation, i.e. Morello. In comparison to Nienhuis et al., this more abstract definition of the security property allows more automation and thereby reach the scale of Morello. Bauereiss et al. mention that proving stronger properties, such as capability safety, requires proof techniques that do not scale up to full-scale industry architectures. Part of the reason is that current automation techniques for separation logics in a foundational setting [11, 13, 15, 34, 39, 45] are still insufficient. 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 [22] formally verify the correct execution of CHERI-instructions and liveness properties for CHERI-Flute [2022]: a concrete implementation of CHERI-RISC-V. Their results apply only to this specific ISA implementation, making their work very different from ours. Similarly, Cheang et al. [14] prove functional correctness of RISC-V PMP for the Rocket Chip implementation, as a step towards verifying the Keystone [35] framework.

Dam et al. and Khakpour et al. use a basic Hoare logic to formulate ISA security primitives for Armv7. This suffices for their security primitive, where the CPU transitions only from untrusted code to trusted code on interrupt or syscall, making it easy to define the scenarios for which trusted code needs to ensure secure behavior. Capability safety exemplifies a more complicated security property because arbitrarily many security boundaries can be crossed: any enter capability that is directly or indirectly accessible from current architectural registers. Because of this, phrasing such guarantees requires a sufficiently expressive logic. Other security primitives can be expected to similarly require or benefit from advanced logic features like ghost state, invariant, ownerhsip etc.

Guarnieri et al. [28] 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. [23], who propose adding guarantees about side-channel leakage to the ISA in an *augmented ISA (aISA)*. Both 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 future work, we intend to add support for confidentiality guarantees and side channels.

The conventional way to reason about separation logic in proof assistants is to use a shallow embedding of propositions and provide meta-programming facilities, like tactics or plugins, which can be used to implement proof steps that *interactive forward symbolic execution* [11] of program fragments at the meta-level [13, 15, 34, 39, 45]. In the background a proof term is constructed which has to be checked by the system for each run. In contrast, Katamaran is not interactive and uses a deep embedding of propositions and is implemented at the object-level language of Coq called Gallina and we verified once and for all that each run of the symbolic executor is sound. As a consequence, the usage and implementation of Katamaran is closer to standalone verifiers and frameworks [19, 29, 36, 40, 47]. A downside of the approach is that existing meta-level machinery cannot be used during symbolic execution for automation including simple symbolic evaluation, which instead have to be specifically implemented. The deep embedding also incurs an increased complexity in terms of explicitly dealing with logic variable allocation. VeriSmall [3] and Mechanized Featherweight VeriFast [30] are to the best of our knowledge the only other mechanized symbolic executors for separation logic based on deep embeddings. However, both largely serve as a proof of concept mechanizations for practical systems [9, 29] while Katamaran aspires to be a practical mechanized implementation in itself.

8 CONCLUSION

Our work shows that universal contracts, together with Katamaran, form a compelling, tool-supported method for formalizing, verifying and applying the security guarantees of ISAs. We have demonstrated that the approach applies to ISAs with very different security primitives (capabilities versus PMP+exceptions), to industrially relevant ISAs like RISC-V PMP (with only minor simplifications remaining) and that the approach balances the requirements of ISA designers and authors of security-critical software. We have demonstrated how we can check the consistency of universal contracts with functional ISA semantics and how they can be applied to reason about security-critical software. Finally, we have shown that Katamaran is able to effectively semi-automate the verification of universal contracts as well as the verification of security-critical software, offering convenient ways for users to inject manually proven lemmas in an automatic verification.

In other words, our current results already demonstrate the viability and generality of the approach to verify ISA security properties using universal contracts and Katamaran. In the future, we intend to scale up our approach by applying it to larger ISAs, supporting complex semantic features (such as asynchronous interrupts, concurrency etc) and other security properties (e.g. confidentiality). We also intend to automate the translation from Sail to μSail and improve Katamaran automation further.

ACKNOWLEDGMENTS

AVAILABILITY

The source code of the Coq development for this paper is available at https://github.com/katamaran-project/katamaran/releases/tag/ccs23.

REFERENCES

[1] *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.

[2] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, New York, NY, USA, 2017. Association for Computing Machinery. doi: 10.1145/3009837.3009878.

[3] Andrew W. Appel. Verismall: Verified smallfoot shape analysis. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs*, pages 231–246, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-25379-9.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[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.

[9] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, *Programming Languages and Systems*, pages 52–68, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. ISBN 978-3-540-32247-4.

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

[11] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W Appel. Vst-floyd: A separation logic tool to verify correctness of c programs. *Journal of Automated Reasoning*, 61(1):367–422, 2018.

[12] 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.

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

[14] 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.

[15] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. *SIGPLAN Not.*, 46(6):234–245, jun 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993526. URL https://doi.org/10.1145/1993316.1993526.

[16] Mads Dam, Roberto Guanciale, Narges Khakpour, Hamed Nemati, and Oliver Schwarz. Formal verification of information flow security for a simple arm-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 223–234, 2013.

[17] 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.

[18] 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.

[19] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37036-6.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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 2021.

[26] 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, 2021.

[27] 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.

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

[29] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 41–55, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-20398-5.

[30] Bart Jacobs, Frédéric Vogels, and Frank Piessens. Featherweight VeriFast. *Logical Methods in Computer Science*, Volume 11, Issue 3, September 2015. doi: 10.2168/LMCS-11(3:19)2015. URL https://lmcs.episciences.org/1595.

[31] 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.

[32] 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.

[33] Narges Khakpour, Oliver Schwarz, and Mads Dam. Machine assisted proof of armv7 instruction level isolation properties. In *Certified Programs and Proofs: Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings 3*, pages 276–291. Springer, 2013.

[34] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. Mosel: A general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.*, 2(ICFP), jul 2018. doi: 10.1145/3236772. URL https://doi.org/10.1145/3236772.

[35] 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.

[36] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-17511-4.

[37] Sergio Maffeis, John C Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted web applications. In *S&P*, pages 125–140. IEEE, 2010.

[38] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi: 10.1145/3341708.

[39] Ike Mulder, Robbert Krebbers, and Herman Geuvers. Diaframe: Automated verification of fine-grained concurrent programs in iris. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 809–824, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523432. URL https://doi.org/10.1145/3519939.3523432.

[40] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016. URL https://doi.org/10.1007/978-3-662-49122-5_2.

[41] 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.

[42] 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.

[43] 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.

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

[45] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. Refinedc: Automating the foundational verification of c code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 158–174, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454036. URL https://doi.org/10.1145/3453483.3454036.

[46] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. Islaris: Verification of machine code against authoritative isa semantics. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 825–840, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523434. URL https://doi.org/10.1145/3519939.3523434.

[47] Malte H. Schwerhoff. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. Doctoral thesis, ETH Zurich, Zürich, 2016.

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

[49] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, New York, NY, USA, 2013. Association for Computing Machinery. doi: 10.1145/2491956.2491978.

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

[51] 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.

[52] Thomas Van Strydonck, Aïna Linn Georges, Armaël Gueneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. Proving full-system security properties under multiple attacker models on capability machines. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 80–95, August 2022. doi: 10.1109/CSF54842.2022.9919645.

[53] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy*, 2015. doi: 10.1109/SP.2015.9.

[54] 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.

A END-TO-END SECURITY STATEMENT ABOUT FEMTOKERNEL

Lemma *femtokernel_endToEnd* :

$mem\_has\_instrs\ \mu\ 0\ femtokernel\_init \rightarrow$

$mem\_has\_instrs\ \mu\ 72\ femtokernel\_handler \rightarrow$

$mem\_has\_word\ \mu\ 84\ 42 \rightarrow$

$read\_register\ \gamma\ cur\_privilege = Machine \rightarrow$

$read\_register\ \gamma\ pmp0cfg = femtokernel\_default\_pmpcfg \rightarrow$

$read\_register\ \gamma\ pmpaddr_0 = 0 \rightarrow$

$read\_register\ \gamma\ pmp1cfg = femtokernel\_default\_pmpcfg \rightarrow$

$read\_register\ \gamma\ pmpaddr_1 = 0 \rightarrow$

$read\_register\ \gamma\ pc = 0 \rightarrow$

$\langle \gamma, \mu, \delta, fdeCycle() \rangle \longrightarrow^* \langle \gamma', \mu', \delta', s' \rangle \rightarrow$

$\mu'\ 84 = 42.$