Katamaran: semi-automated verification of ISA specifications

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

Georgy Lukyanov
Newcastle University, United Kingdom
g.lukyanov2@newcastle.ac.uk

Dominique Devriese
Vrije Universiteit Brussel, Belgium
dominique.devriese@vub.be

1 Introduction

An instruction set architecture (ISA) is an abstract specification of the syntax and semantics of machine code. It defines an envelope of allowed behaviour for CPU designers and a set of assumptions that software designers can rely on. Instead of informal prose and pseudo-code [e.g. 1], rigorous, executable formalisations of ISAs disambiguate the contract and improve testability and support modification, experimentation and formal study [e.g. 3, 12, 19]. Such formalisations are a crucial requirement for formal verification of both hardware [e.g. 8] and software [e.g. 15].

We are interested in verifying that critical safety guarantees of the ISA are upheld by the semantics of all instructions. Our long-term goal is to verify security guarantees offered by ISAs, specifically features like Intel SGX [16], virtual memory or capability machines [6]. We want to verify these properties in a form that can be used to reason about programs, as a way to ultimately verify security properties of real systems.

For achieving this, we take inspiration from recent formulations of capability safety in capability machines and high-level languages [10, 20, 22, 24]. Contrary to, for example, Nienhuis et al. [18], such techniques directly enable reasoning across encapsulation boundaries. These approaches use (essentially) a general purpose program logic, and formulate capability safety as a *universal contract* that automatically holds for arbitrary programs. The universal contract expresses guarantees provided by the machine and can be used for manually verifying trusted programs that interact with untrusted programs. We believe that this approach generalises well beyond capability safety.

However, proving such results about a language or ISA currently requires a lot of manual reasoning. For example, an in-progress Coq formalisation of capability safety for a simple capability machine with 19 instructions requires about 17kLoC of proofs [11]. Real ISAs can be much larger, for example 30kLoC of Sail specifications for ARMv8.3 [3]. Consequently, scaling up ISA property proofs raises important proof engineering challenges. For the verification effort to scale reasonably in terms of the size and complexity of the specification and for making it robust to changes, proof automation is a necessity. Uninteresting parts of the proof should be dealt with automatically, but at the same time, a human should be able to intervene and prove certain cases manually or provide heuristics to steer the automation. In this text, we present Katamaran: a semi-automated verification tool for Sail, intended to accommodate these requirements. The tool is a semi-automated separation logic verifier for Sail in the tradition of SmallFoot [23], VeriFast [13] and others. We intend to mechanically verify it against an abstract separation logic interface, making the tool similar to Bedrock [7], HOLfoot [23], VeriStar+VeriSmall [2, 21], Vst-Floyd [5] and the mechanisation of Featherweight VeriFast [14].

In the remaining sections, we give a more detailed motivation of how we want to verify of ISA properties in Katamaran (Section 2) and a high-level overview of the design and current status of the tool and our next plans (Section 3).

Our developments are publicly available on Github [9].

2 Motivation

To understand how we intend Katamaran to semi-automatically prove ISA properties, consider the procedure

```
execute_store:(d s:reg)→unit
```

in Figure 1, which defines the semantics of a store instruction in a simple capability machine. The code is written in μSail, a subset of Sail that we briefly describe in Section 3. Ignoring the contracts in red and blue for now, it reads a capability from a destination register `d`, checks its bounds and permissions, reads a word from a source register `s`, and stores it to memory using an omitted `write_mem` procedure.

The lines in red state the universal contract for instructions on the machine. The contract is specified in an underlying separation logic, in terms of an abstract predicate `safe:cap+int→ℙ`, essentially stating that the instruction starts out with safe values in all registers and should also finish with safe register contents. Additionally, under the rules of the underlying separation logic and the predicate `safe`, this contract also implies that the instruction can only access memory reachable through the register contents and will respect any registered invariants of the system (see, for example, Skorstengaard et al. [20] for more details).

To verify the contract, Katamaran applies *symbolic execution* [4] and automatically computes the blue intermediate assertions by applying program logic rules. These intermediate assertions consist of pure logical facts (the "path condition") and spatial separation logic assertions (the "symbolic heap").

PL'18, January 01–03, 2018, New York, NY, USA
2018.

```
{ ∗_{r∈reg} r ↦ w_r ∗ safe(c_r)}
  let: c  : cap  := call read_reg_cap d in
{( ∗_{r∈reg} r ↦ w_r ∗ safe(c_r)) ∗ ⌜c = w_d ∧ c = cap(p, b, e, a)⌝}
  let: wa : bool := call write_allowed c.perm in
  let: wb : bool := call within_bounds c in
  assert (wa && wb) ;;
{... c = cap(p, b, e, a) ∧ p ⊒ W ∧ b ≤ a < e⌝}
  let: w : cap + int := call read_reg_word s in
{... c = cap(p, b, e, a) ∧ p ⊒ W ∧ b ≤ a < e ∧ w = w_s⌝}
  call write_mem c.cursor w ;; call update_pc
{ ∗_{r∈reg} r ↦ w_r ∗ safe(c_r)}
```

Figure 1. Contract checking for `execute_store d s`

Function calls are symbolically executed by instantiating function contracts. For instance, `execute_store` uses the underlying function

```
{safe(cap(p, b, e, −)) ∗ safe(w) ∗ ⌜b ≤ a < e ∧ p ⊒ W⌝}
  write_mem a w
{safe(cap(p, b, e, −)) ∗ safe(w)}
```

The precondition of a function contract usually needs to be massaged to match the symbolic heap and path condition when the call is made. This means instantiating quantified variables, applying logic rules – like the *frame rule* – and lemmas for both pure and impure assertions. Katamaran attempts to automatically perform this matching and otherwise delegates to user-provided heuristics that may tell it to apply specific lemmas for making the matching go through.

Like other tools, handling of straight-line code is fully automatic and branches are dealt with by exploring all paths separately. Easy pure assertions are solved immediately and others are collected as *verification conditions (VCs)*, to be proved separately (see Section 3).

We conjecture that Katamaran should be able to automatically discharge most uninteresting parts of ISA property proofs because typical Sail specifications tend to contain a lot of mostly straight-line code, (almost) no loops, no higher-order functions and no dynamic memory allocation. Nevertheless, we plan to provide escape hatches for manually discharging the obligations that Katamaran cannot handle.

3 Overview and Status

Finally, we provide an overview of Katamaran's workflow, depicted in Figure 2, and its implementation status.

Machine specifications As a specification language, Katamaran reuses Sail which is translated to μSail – a subset of Sail deeply-embedded into Coq. We believe that all of Sail is compilable to μSail via transformations like monomorphisation and desugaring. μSail's features and a comparison to Sail are documented in our code repository [9]. For the moment, the translation is still vaporware and we are writing examples as μSail ASTs directly. One notable difference to Sail's existing Coq backend is the deep rather than shallow

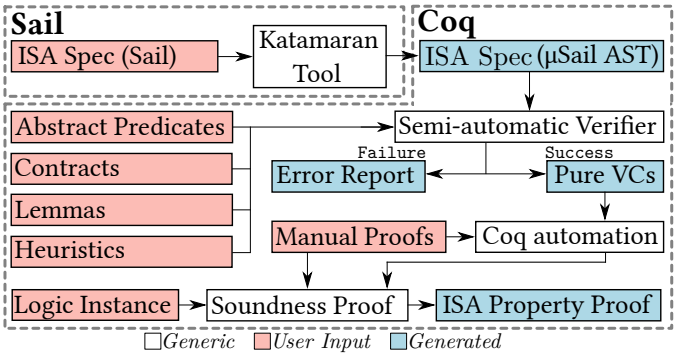

Figure 2. Katamaran Overview

embedding which allows for more parts of the verifier to be implemented in Gallina rather than Ltac.

ISA property specifications ISA properties are specified by way of *contracts*, like the one in Figure 1, which may use predicates like `safe`. Katamaran treats predicates and also – like Bedrock [7] VeriStar [21] – the assertion logic abstractly, relying on a generic logic interface that the user instantiates. This modularity allows the user to choose a separation logic rich enough to define all the predicates.

Notably absent from Sail and μSail are definitions of what constitutes memory of a specified machine. Katamaran leaves it up to the user to define a *memory model* as part of her logic instantiation and provide primitive access to it. For instance, the `write_mem` function can be defined and its contract proved sound externally in the logic, and given as inputs to Katamaran.

ISA property proofs Katamaran's semi-automatic μSail verifier will attempt to check the desired ISA property. The user may control the behaviour of the verifier through *lemmas*, for instance to fold/unfold recursive predicates, and *heuristics* that use lemmas to guide the spatial reasoning.

If successful, the verifier outputs a list of pure VCs for each possible execution path. If these can be proven (by existing Coq automation or, if necessary, manual Coq proofs), the contracts are provable in the underlying logic. If verification fails, Katamaran will terminate with an error message.

At the time of writing, the symbolic executor is the most mature component of Katamaran, although it is still missing a soundness proof and some features, like for instance the automatic application of lemmas via heuristics. Lemmas can be invoked manually though by a form of ad-hoc ghost statement in the syntax.

Our next plans for Katamaran are to develop it further and apply it to a number of increasingly challenging applications. This starts with simple examples, and ends with verifying capability safety in full-fledged capability machine ISAs. Intermediate goals notably include properties of artificial ISAs from the literature [e.g. 20] and a form of memory safety of Redfin [17] — a specialised ISA for aerospace applications designed with formal verification in mind.

References

[1] AMD. 2019. AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions. https://www.amd.com/system/files/TechDocs/24594.pdf

[2] Andrew W. Appel. 2011. VeriSmall: Verified Smallfoot Shape Analysis. In *Certified Programs and Proofs*. Springer Berlin Heidelberg, 231–246.

[3] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 71:1–71:31. https://doi.org/10.1145/3290384

[4] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Symbolic Execution with Separation Logic. In *Programming Languages and Systems*. Springer Berlin Heidelberg, 52–68.

[5] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *Journal of Automated Reasoning* 61, 1 (June 2018), 367–422. https://doi.org/10.1007/s10817-018-9457-5

[6] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. 1994. Hardware Support for Fast Capability-Based Addressing *(ASPLOS VI)*. ACM. https://doi.org/10.1145/195473.195579

[7] Adam Chlipala. 2011. Mostly-automated Verification of Low-level Programs in Computational Separation Logic. *SIGPLAN Not.* 46, 6 (June 2011), 234–245. https://doi.org/10.1145/1993316.1993526

[8] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. 2017. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proceedings of the ACM on Programming Languages* 1, ICFP (Aug. 2017), 24:1–24:30. https://doi.org/10.1145/3110268

[9] Katamaran Developers. 2020. Katamaran Code Repository. https://github.com/skeuchel/katamaran.

[10] Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities Using Logical Relations and Effect Parametricity. In *European Symposium on Security and Privacy*. IEEE. https://doi.org/10.1109/EuroSP.2016.22

[11] Aïna Linn Georges, Alix Trieu, and Lars Birkedal. [n.d.]. Mechanized Reasoning about a Capability Machine. https://cs.au.dk/~birke/papers/iris-capabilities-prisc-conf.pdf Principles of Secure Compilation, 2020.

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

[13] Bart Jacobs, Jan Smans, and Frank Piessens. 2010. A Quick Tour of the VeriFast Program Verifier. In *Programming Languages and Systems*. Lecture Notes in Computer Science, Vol. 6461. Springer Berlin Heidelberg, 304–311.

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

[15] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[16] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution *(HASP '13)*. ACM. https://doi.org/10.1145/2487726.2488368

[17] Andrey Mokhov, Georgy Lukyanov, and Jakob Lechner. 2019. Formal Verification of Spacecraft Control Programs (Experience Report) *(Haskell '19)*. ACM. https://doi.org/10.1145/3331545.3342593

[18] Kyndylan Nienhuis, Alexandre Joannou, Anthony Fox, Michael Roe, Thomas Bauereiss, Brian Campbell, Matthew Naylor, Robert M Norton, Simon W Moore, Peter G Neumann, et al. 2019. *Rigorous engineering for hardware security: formal modelling and proof in the CHERI design and implementation process.* Technical Report. University of Cambridge, Computer Laboratory. Accepted for publication at IEEE S&P 2020.

[19] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-End Verification of Processors with ISA-Formal. In *Computer Aided Verification (Lecture Notes in Computer Science)*. Springer International Publishing, 42–58. https://doi.org/10.1007/978-3-319-41540-6_3

[20] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2017. Reasoning About a Capability Machine with Local Capabilities - Provably Safe Stack and Return Pointer Management (without OS Support). (2017). http://people.cs.kuleuven.be/dominique.devriese/wellbracketed-locally.pdf In submission.

[21] Gordon Stewart, Lennart Beringer, and Andrew W. Appel. 2012. Verified Heap Theorem Prover by Paramodulation *(ICFP '12)*. ACM. https://doi.org/10.1145/2364527.2364531

[22] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns. In *OOPSLA*. ACM. https://people.mpi-sws.org/~swasey/papers/ocpl/ocpl-20170418.pdf

[23] Thomas Tuerk. 2009. A Formalisation of Smallfoot in HOL. In *Theorem Proving in Higher Order Logics*. Springer Berlin Heidelberg, 469–484.

[24] Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. 2019. Linear Capabilities for Fully Abstract Compilation of Separation-Logic-Verified Code. *Proc. ACM Program. Lang.* ICFP (2019). accepted.