Work in progress: A formally verified shadow stack for RISC-V

Matthieu Baty
IRISA
Inria, CNRS
Rennes, France
matthieu.baty@inria.fr

Guillaume Hiet
IRISA
CentraleSupelec, Inria, CNRS
Cesson-Sévigné, France
guillaume.hiet@centralesupelec.fr

Pierre Wilke
IRISA
CentraleSupelec, Inria, CNRS
Cesson-Sévigné, France
pierre.wilke@centralesupelec.fr

***Abstract*—In recent years, the disclosure of several significant security vulnerabilities such as Spectre and Meltdown has revealed the trust put in some presumed security properties of commonplace hardware to be misplaced. A way of rebuilding this trust would be to make these security properties explicit and offer corresponding computer-checked proofs.**

Formally proving security properties about hardware systems might seem prohibitively complex and expensive. This paper addresses this concern by describing a realistic and accessible methodology for specifying and proving security properties during hardware development. We describe the formal specification and implementation of a shadow stack mechanism on an RV32I processor. Our final objective would be to prove that this security mechanism is correct, i.e., any illegal modification of a return address does indeed result in the termination of the whole system.

***Index Terms*—RISC-V, Formal Methods, Hardware Verification, Shadow Stack**

1. Introduction

Formal methods can be used to build trust in the properties of hardware components. Their use in the domain was historically mostly confined to computer safety, as exemplified in Intel hardware with the formal verification of the floating-point operations following the discovery of the Pentium FDIV bug [1] or with the verification of cache coherence protocols [2]. However, they are also a good fit for computer security. Indeed, proposed security mechanisms can be formally specified and proven to offer some guarantees.

In a recent example, some such properties have been formally established for the Morello architecture, an extended version of the Arm ISA which includes constructs for fine-grained memory protection [3]. In contrast to our work, these properties were proved at the ISA level, whereas for many interesting security properties, a lower level, such as register transfer level, is more appropriate. Alas, although they offer some promising ways of tackling the long-standing issue of security in hardware design, formal methods are still far from being a common sight in the industry at large.

This article proposes a realistic and accessible methodology for specifying and proving security properties during hardware development. We focus on a concrete example where such a property is defined and subsequently proved. We modify an existing formal model of an embedded RISC-V processor, equipping it with a hardware-based return address shadow stack that acts as a runtime safeguard against return address modifications. We then prove that this security mechanism works as it should. Although this example is quite simple, it allows us to demonstrate a gamut of techniques that we could rely on to prove other security properties on a wide range of hardware. Note that even though the property we intend to demonstrate does not depend on side-effects, the language we use makes it possible to demonstrate timing-related properties with a similar methodology thanks to its cycle-accurate semantics.

In section 2, we introduce the notions used throughout the rest of this paper. In particular, we give a brief introduction to Kôika [4], the formal Hardware Description Language (HDL) in which our model is defined. Our main contributions are detailed in section 3. We start by describing some tools we implemented in order to be able to reason about Kôika programs, before turning our attention to our implementation of the shadow stack mechanism. Afterwards, we present the guarantees that we expect our model to enforce alongside the proof that it indeed does enforce them. We mention related work in section 4 and future work in section 5 before concluding in section 6.

2. Background

2.1. Formal methods and hardware design

Formal methods made their way into the hardware industry. Currently, most of their uses in this field have to do with functional verification, that is, with proving that some specification is respected. Usually, a piece of hardware is modelled in a language at some level of abstraction. Properties corresponding to its specification are then defined and proved to hold for the model [5].

This approach leaves much room for improvement. Not only does it lead to duplicated efforts, but using different models for generating hardware and for reasoning also means that there can be discrepancies between the hardware and its formal representation. Proofs certified by a formal system can be misleading if the model they are based upon is itself wrong.

This work was supported by a grant from the French National Cybersecurity Agency (ANSSI).

Another limitation that comes with traditional model checking is related to abstraction. At most levels of abstraction, there are some interesting properties that cannot be proved. For instance, timing attacks are out-of-scope at a functional level.

A way of avoiding these issues would be to reason directly on the model used for production. This is not the standard method since most HDLs used in industrial settings do not have a formal semantics. Formal HDLs do exist, although they tend to be limited in terms of compatible tooling. It is also possible to retrofit formal semantics onto existing languages such as VHDL or Verilog.

2.2. Shadow stacks

Memory corruptions are still one of the most significant vulnerabilities in software developed in low-level languages like C or C++. Indeed, the developer is in charge of the application memory management in those languages, which can lead to spatial and temporal memory safety errors. Attackers can exploit such vulnerabilities to leak confidential data or modify the application's intended behavior. For example, they can exploit some buffer overflow on the stack to modify the return address, one of the most popular attacks of this type. Hardware-based security mechanisms implementing Control Flow Integrity, like Intel CET [6], are appealing solutions to protect software against such attacks. They offer more robust protection than software-based approaches, since software attacks cannot modify them.

We are interested in the property that functions do indeed return to the instruction following their call. A possible solution to verify such a property is to maintain a shadow stack. The processor pushes the expected return address onto this stack for each function call and pops it whenever it returns. If the address a function tries to return to, using the regular function stack and return address register, is not equal to the one on top of the shadow stack, we can deduce that something went wrong and react accordingly. Of course, we must also protect this shadow stack and prevent any regular write into the memory performed by the application code to modify the shadow stack content.

Shadow stacks can be implemented either in software [7] or in hardware. Although software implementations provide some benefits (chief among them being their compatibility with existing hardware), we will focus on hardware implementations. These offer the advantage of working with any program without the need for patching.

2.3. Kôika

Kôika [4] is an open-source formal hardware design language. We decided to rely on this tool for modeling the shadow stack as it is a reasonable basis for formal reasoning due to it being embedded within the general-purpose formal language Coq.

Kôika is based on BlueSpec [8], a general-purpose, high-level HDL with a focus on automatic generation of control logic, which is convenient for concurrent systems such as pipelined processors. It supports Verilog output through a formally verified compiler. Thus, it can use Verilog compatible tools (e.g., simulators and FPGA bitstream generators). Efficient simulation is possible using the project's custom simulator "Cuttlesim" [9]. Kôika lends itself better to efficient simulation than Verilog because its form is close to usual software, which allows the standard software optimization techniques to be applied. The project also provides some formally demonstrated properties about the language itself.

In this section, we give a brief introduction to Kôika. We do not aim at being exhaustive, only at making this paper understandable on its own.

2.3.1. Rules and conflicts. Kôika models are composed of the following elements:

- a set of registers, along with their types and their initial values;
- a set of external calls and their types, allowing the model to interact with its environment (e.g., an external memory or peripheral devices);
- a set of rules, which are atomic actions describing state transitions that act as building blocks of the logic of the model — for instance, the definition of a processor may contain rules such as fetch, decode and execute;
- a schedule built using the rules, describing the order in which the rules should be applied.

Kôika is smart enough to run rules in parallel when they are not in a conflict with each other. For instance, there can be write conflicts due to two rules attempting to write to the same register in the same cycle. If running a rule during a cycle would lead to a conflict, then it is skipped for the time although rules that appear later in the schedule may still be executed. In fact, the "One Rule At A Time" theorem guarantees that the circuits Kôika generates are functionally equivalent to systems running rules sequentially.

Combined with scheduling, this behavior can be used to simplify the definition of pipelined systems: conflicts help determine how to pipeline a model without the user needing to give all the details explicitly.

2.3.2. External calls. Kôika is a pure functional language, but that does not prevent it from representing impure actions through the notion of external calls. These can be used to represent interaction with the external world. When generating Verilog code from a Kôika model, external calls can be bound to Verilog modules.

For instance, external calls can help model external memory. Indeed, a limitation of Kôika is that it can only rely on Verilog registers for representing memory, whereas on FPGAs, the natural solution would be to make use of block RAM. Even for a rather simple model such as ours, this is limiting for testing. The alternative is to delegate memory accesses to raw Verilog code which ensures that block RAM is used, and the way to do this is through external calls. The main downside of this solution is that the Verilog code cannot be reasoned about directly from Kôika.

3. Contributions

3.1. The processor model

Conveniently, Kôika includes a simple model of a pipelined RISC-V processor that can be specialized to cover part of the RV32I or the RV32E part of the standard. This model does not aim for exhaustiveness and is not proven to conform to the RISC-V specification (although it passes the test suite for all the instructions it implements). It is used as a testing place and a way to showcase Kôika's more advanced features. Our model is a slightly tweaked and expanded version of this example.

3.2. The stack model

We added a shadow stack module to the processor. We can prove its isolation from the core model, in that the only way of acting on it is through its two methods, push and pop. These methods are called automatically when the current instruction corresponds to a function call or return. They both expect one argument: the address of the instruction following the current function call for push and the stack's return address for pop.

3.2.1. Detecting function calls and returns in machine code. Contrary to ISAs such as x86, which have dedicated call and return instructions, RISC-V uses the same instruction for multiple purposes. This choice is common for RISC (Reduced Instruction Set Computer) ISAs. The JAL and JALR instructions implement both unconditional jumps and function calls. However, the arguments that are passed to them make their role clear. The Application Binary Interface describes which instructions calls should be interpreted as function calls or returns, depending on their arguments.

In fact, the RISC-V specification includes information regarding how shadow stacks (which they call return-address stacks) should behave:

> For RISC-V, hints as to the instructions' usage are encoded implicitly via the register numbers used. A JAL instruction should push the return address onto a return-address stack (RAS) only when rd = x1/x5. JALR instructions should push/pop a RAS as shown in the table [that follows].

rd	$rs1$	$rs1 = rd$	RAS action
$!link$	$!link$	–	none
$!link$	$link$	–	pop
$link$	$!link$	–	push
$link$	$link$	0	pop, then push
$link$	$link$	1	push

Return-address stack prediction hints encoded in register specifiers used in the instruction. [...] *link* is true when the register is either x1 or x5.

3.2.2. Dealing with a detected stack buffer overflow. On a system with a full-fledged operating system, a stack buffer overflow could be left for the system to manage. For instance, the affected program could be killed, and an error could be logged or displayed to the user. In our simple embedded system, not all of these options are open. The two main possibilities are:

- ending the current execution;
- correcting the return address using the shadow stack information (or just relying purely on it and ignoring return arguments).

The latter option might be tempting. However, it comes with significant downsides. If the return address has been modified, then the rest of the stack has likely been impacted and cannot be considered safe. Also, arbitrarily modifying the return address is illegal according to the ISA. On the other hand, stopping execution is not problematic, as it does not take the processor to an illegal state.

Our verified stack implementation halts execution on a mismatch. In order to prove anything about our processor halting, we first need to define what this means for Kôika models — this is tricky since Kôika does not have a notion of halting execution of a model. We deal with this by emitting an external call and putting our processor into a sink state. This is done through a variable that guards the execution of all the rules in the schedule. We make sure that whenever the external call corresponding to the shutdown is emitted, no changes of state can possibly happen in subsequent cycles. When exporting to Verilog, we link this external call to a module that really halts the execution of the processor.

An alternative could have been to jump to an exception handler, where e.g. logging could take place before taking either action, some form of reset could happen or a reboot could be triggered. In our minimal example, we don't care about what happens past the detection and we content ourselves with setting the variable guarding all rules in places where a call to such a handler could have existed.

3.3. Adapting Kôika

The fact that a language has a formally defined semantics does not imply that it is directly usable for proving general properties. Although it is possible to prove some things about the behavior of simple Kôika circuits using only the language itself, Kôika is overall more of a language about which there are proofs than a language to build formally certified hardware. It turns out that trying to demonstrate even basic properties of complex models leads to serious performance issues due to the way Kôika's structure interacts with Coq's proving facilities. In particular, the fact that the rules we consider are rather large, together with Kôika's heavy use of dependent types, makes interactive reasoning prohibitively slow.

The bulk of our work was dedicated to finding a way around this issue. We implemented a function converting Kôika models to a form that is better suited to formal reasoning and an interpretation function for this new form. Instead of having the raw Kôika rules, our new *simple form* is a mapping from each register to a symbolic expression, which depends on the initial values of each register. We proved that the simple form we construct is equivalent to the original Kôika program. We can therefore reason exclusively about this simpler form.

Furthermore, we started putting together a toolbox of theorems and tactics for reasoning about circuits. A simple example of such a theorem is `no_write_no_change`, defined hereafter.

```
Lemma no_write_no_change :=
 forall reg env sf,
 list_assoc (final_values sf) = None ->
  getenv env reg =
  getenv (interp_cycle env sf) reg.
```

This theorem comes in handy to eliminate trivial goals about registers that cannot be modified under the current assumptions. Indeed, this theorem states that if there is no variable associated with the final value of some Kôika register in our alternative form `sf`, then the value of this register cannot change during the interpretation of a cycle. This result follows from how we built the interpretation function for our custom form.

We implemented and proved the correctness of a simplification function, `propagate`, which simplifies a model under the assumption that the initial value of some register is known, by replacing all occurrences of this register by its value.

```
Lemma propagate_ok :=
  forall reg reg_vinit reg' env sf,
  getenv env reg = reg_vinit ->
  getenv (interp_cycle env sf) reg'
  = getenv
    (interp_cycle
      (propagate env reg reg_vinit) sf)
    reg'.
```

Coq makes it possible to define custom tactics to automate away part of the tedium. We could, for instance, define general tactics which take our hypotheses into account and then attempt to simplify our model as much as possible, and even recognize some simple subgoals and solve those automatically. For simple properties, proofs could be fully automated. As of now, we only have some simplification functions like `propagate` as well as lemmas like `no_write_no_change`, and we yet have to assemble these building blocks to build more powerful tactics.

3.4. Formally verified properties

In plain English, the property that we intend to demonstrate might be worded as "any illegal call to the stack module or shadow stack overflow results in halting the processor without any further change of state". Let us formalize this property.

We start by defining what an illegal call to a shadow stack is. Calls to shadow stack methods can fail in the following ways:

- calling pop with an argument that does not correspond to the top of the stack: this happens when the return address has been modified;
- calling pop when the stack is empty: this corresponds to the case where a return instruction occurs that does not match a call instruction;
- calling push when the stack is full: our implementation features a fixed size stack; therefore it may happen that the stack is too small to contain all the (otherwise legitimate) return addresses.

In all these cases, we halt the execution of the processor.

Hereafter are some useful definitions about the shadow stack (`sstack`) that we will use throughout our proof:

```
Definition sstack_empty env :=
 getenv env sstack.sz = 0.
Definition sstack_full env :=
 getenv env sstack.sz = sstack.capacity.
Definition sstack_top_address env :=
  match (getenv env sstack.sz) with
  | 0 => None
  | x => Some (getenv env (sstack.stack x))
  end.
Definition is_halt_set env :=
  sstack.halt = 1.
```

Our processor is pipelined, which implies that several instructions are in-flight at the same time. Nonetheless, there is at most one instruction at the execute stage at any point, and it just so happens that all the calls to shadow stack functions occur there.

There is no definition for the address on top of the (non shadow) stack. Although we could devise one, it is not necessary for our proof. Indeed, we can simply rely on the information that can be found in the data used for synchronizing between the decode and the execute stage of our processor to get the current instruction, from which we can deduce whether the instruction is a procedure return or not, and, if it is, the address it returns to.

Predicates `sstack_push` and `sstack_pop` express the conditions under which a push or a pop takes place. Their definition (omitted here) simply amount to checking whether the instruction in the execute stage is a call or a return instruction. The `no_mispred` construct is used for dealing with the mispredictions that can result from branch instructions. The effects of a mispredicted instruction have to be ignored. At the point where an instruction reaches the execution stage of the pipeline, it is already known whether or not it belongs to a mispredicted branch and therefore whether or not it has to be ignored. Function `stack_top_address` (not to be confused with `sstack_top_address`), corresponds to the address being returned to if the current instruction is a procedure return and to `None` if it is not.

We can once again define what a shadow stack violation is, this time formally:

```
Definition sstack_uflow env :=
  no_mispred env /\ sstack_empty env
  /\ sstack_pop env.
Definition sstack_oflow env :=
  no_mispred env /\ sstack_full env /\
  ~(sstack_pop env) /\ sstack_push env.
Definition sstack_addr_violation env :=
  no_mispred env /\ sstack_pop env
  /\ stack_top_address env
     <> sstack_top_address env.
Definition sstack_violation env :=
  sstack_uflow env \/ sstack_oflow env
  \/ sstack_addr_violation env.
```

We mentioned how we added a variable guarding each of our rules to ensure that no changes of state can occur in our model after the processor was halted. We expect that setting this variable to `true` causes all the rules to fail and therefore blocks any change of state from happening. We formalize that with the following property:

```
Lemma no_further_changes_of_state
  env extcalls_model :=
  forall reg n,
  getenv env reg =
  getenv
```

```
    (interp_n_cycles n env
     calls_model riscv_model_sf) reg.
```

With that out of the way, we can define our main property:

```
Theorem sstack_ok :=
  forall env extcalls_model,
  sstack_violation env ->
  no_further_changes_of_state
  (interp_cycle env
   extcalls_model riscv_model_sf)
  extcalls_model riscv_model_sf.
```

3.5. Proof

As a first intermediate step, we can split our goal into `stack_violation` implies `sets_halt` and `sets_halt` implies `no_further_changes_of_state`, where `sets_halt` is true if after the current cycle, `halt` is set to `true`.

The second subgoal can be easily discharged, because the code for each rule starts by checking the value of the `halt` register, but the first subgoal is more challenging.

We are currently in the middle of proving that if the `sstack_violation` predicate holds, then the `halt` register is set. This is difficult because, in order for this register to be written, we need to reason about whether the whole `execute` rule conflicts with previous rules or not. Indeed, if there is a conflict, the effects of the entire rule are discarded, included the potential writes to the `halt` register.

We know that for halt to be set, either `JAL` or `JALR` must be entering the execute stage, but this needs to be proved:

```
Lemma halt_set_by_JAL_JALR_only :=
  forall env extcalls_model,
  getenv env extcalls_model
    riscv_model_sf halt = 0
  -> sstack_violation env
  -> no_further_changes_of_state
    (interp_cycle env extcalls_model
     riscv_model_sf)
    extcalls_model koika_model_sf.
```

The current definition of the main property depends on the fact that the targeted architecture is in-order. It would need to be modified to support out-of-order architectures. As a consequence, our proof could not be directly reused for such architectures. Although it is possible to define the property so that it works on both types of architectures, this would come at the expense of simplicity and we elected to keep everything as simple as possible for a first example. This does not mean that proof reuse was ignored in this work. Indeed, we introduced many general lemmas and tactics for simplifying models, as outlined at the end of subsection 3.3.

4. Related work

We mentioned the work of Nienhuis et al. [3] on formally verifying properties of a capability extended version of the Arm ISA. This work bears some similarities with ours since they define and prove properties about hardware. However, all their reasoning is done at the ISA level. In contrast, we reason at the register transfer level, which is much closer to the concrete hardware.

Lööw et al. [10] give an example of cross stack formal verification, where formal properties about software rely on formally certified properties of the hardware. It was implemented in CakeML [11] and targets a custom architecture instead of a standardized ISA. Unlike our work, this paper is not concerned with security mechanisms, although they could be considered in this language as well. The example that it considers is the implementation and verification of a certified compiler on certified hardware.

Erbsen et al. [12] describe the implementation of a certified IoT lightbulb. This work also blends formal verification about hardware and software. The main theorem it defines relates to the validity of the behavior of the application controlling the lightbulb. Properties about the hardware, compiler, drivers, and applications are formally verified and contribute to the final proof. Since those elements may vary independently of the others, special attention was given to the proof modularity. Although this paper also comes from MIT's Programming Languages and Verification group, where Kôika was developed, this work is based on a previous formal HDL called Kami [13]. This language does not have a cycle-accurate semantics, which means that as it stands, side-channel attacks are out of its scope.

5. Limitations and future work

5.1. Functional verification

There is an official formal version of the RISC-V specification based on the Sail language [14], which includes facilities to export definitions to Coq. Proving that the processor design we used is conform to this specification would be a logical next step.

5.2. Generalizing the processor model

The processor we are targeting is quite simple (unprivileged ISA, 32 bits, minimal extensions). We could generalize our results by working with a family of processors instead of a single concrete instance. Our proof should work mostly the same way for any legal combination of RISC-V extensions. We have progressed in generalizing the processor model by generating a Kôika processor model from a list of RISC-V extensions. However, the semantics of many new instructions are yet to be defined.

Moreover, we were limited in our implementation by the fact that only the unprivileged part of the specification had been implemented. Adding support for the privileged part of the specification would open possibilities for interacting with the operating system.

5.3. Advanced security mechanisms

Once those basic examples have been shown to work, we would like to consider more ambitious security mechanisms such as a more complex version of shadow stacks or capabilities.

6. Conclusion

In this paper, we documented a workflow for building synthesizable certified hardware. The example we settled on is the implementation of a basic shadow stack for a simple pipelined RISC-V processor. We write our model in the formal HDL Kôika, and the reasoning is carried out in Coq. We are currently working on proving that some security properties related to return address overwrites hold for our processor.

Due to performance issues, an alternative representation of Kôika models had to be devised alongside conversion and interpretation functions. Since we proved these functions are correct, we can get proofs about Kôika models from proofs on our simplified form. Furthermore, we are developing a collection of lemmas and tactics to prove more complex properties.

In the near future, we intend to pursue this work by extending our lemmas and tactics toolkit and by applying it to more ambitious security mechanisms.

References

[1] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. A. Frolov, E. Reeber, and A. Naik, "Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation," in *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, ser. Lecture Notes in Computer Science, A. Bouajjani and O. Maler, Eds., vol. 5643. Springer, 2009, pp. 414–429. [Online]. Available: https://doi.org/10.1007/978-3-642-02658-4_32

[2] M. Talupur and M. R. Tuttle, "Going with the Flow: Parameterized Verification Using Message Flows," in *Formal Methods in Computer-Aided Design, FMCAD 2008, Portland, Oregon, USA, 17-20 November 2008*, A. Cimatti and R. B. Jones, Eds. IEEE, 2008, pp. 1–8. [Online]. Available: https://doi.org/10.1109/FMCAD.2008.ECP.14

[3] K. Nienhuis, A. Joannou, T. Bauereiss, A. C. J. Fox, M. Roe, B. Campbell, M. Naylor, R. M. Norton, S. W. Moore, P. G. Neumann, I. Stark, R. N. M. Watson, and P. Sewell, "Rigorous Engineering for Hardware Security: Formal Modelling and Proof in the CHERI Design and Implementation Process," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1003–1020. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00055

[4] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, "The Essence of Bluespec: a Core Language for Rule-Based Hardware Design," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, A. F. Donaldson and E. Torlak, Eds. ACM, 2020, pp. 243–257. [Online]. Available: https://doi.org/10.1145/3385412.3385965

[5] J. Harrison, "Formal Methods at Intel — An Overview," https://www.cl.cam.ac.uk/~jrh13/slides/nasa-14apr10/slides.pdf, 2010, online; accessed 16 March 2022.

[6] V. Shanbhogue, D. Gupta, and R. Sahita, "Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2019, June 23, 2019*. ACM, 2019, pp. 8:1–8:11. [Online]. Available: https://doi.org/10.1145/3337167.3337175

[7] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, P. Larsen and A. Sadeghi, Eds. ACM / Morgan & Claypool, 2018, pp. 81–116. [Online]. Available: https://doi.org/10.1145/3129743.3129748

[8] R. Nikhil, "Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, 2004, pp. 69–70.

[9] C. Pit-Claudel, T. Bourgeat, S. Lau, Arvind, and A. Chlipala, "Effective Simulation and Debugging for a High-level Hardware Language Using Software Compilers," in *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds. ACM, 2021, pp. 789–803. [Online]. Available: https://doi.org/10.1145/3445814.3446720

[10] A. Lööw, R. Kumar, Y. K. Tan, M. O. Myreen, M. Norrish, O. Abrahamsson, and A. C. J. Fox, "Verified Compilation on a Verified Processor," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 1041–1053. [Online]. Available: https://doi.org/10.1145/3314221.3314622

[11] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens, "CakeML: a Verified Implementation of ML," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 179–192. [Online]. Available: https://doi.org/10.1145/2535838.2535841

[12] A. Erbsen, S. Gruetter, J. Choi, C. Wood, and A. Chlipala, "Integration Verification Across Software and Hardware for a Simple Embedded System," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 604–619. [Online]. Available: https://doi.org/10.1145/3453483.3454065

[13] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: a Platform for High-Level Parametric Hardware Specification and its Modular Verification," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, pp. 24:1–24:30, 2017. [Online]. Available: https://doi.org/10.1145/3110268

[14] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "ISA Semantics for ARMv8-a, RISC-V, and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 71:1–71:31, 2019. [Online]. Available: https://doi.org/10.1145/3290384