A generic framework to develop and verify security mechanisms at the microarchitectural level: application to control-flow integrity

Matthieu Baty
Inria, CNRS, University of Rennes
matthieu.baty@inria.fr

Pierre Wilke
CentraleSupélec, Inria, CNRS, University of Rennes
pierre.wilke@centralesupelec.fr

Guillaume Hiet
CentraleSupélec, Inria, CNRS, University of Rennes
guillaume.hiet@centralesupelec.fr

Arnaud Fontaine
ANSSI
arnaud.fontaine@ssi.gouv.fr

Alix Trieu
ANSSI
alix.trieu@ssi.gouv.fr

***Abstract*—In recent years, the disclosure of several significant security vulnerabilities has revealed the trust put in some presumed security properties of commonplace hardware to be misplaced. We propose to design hardware systems with security mechanisms, together with a formal statement of the security properties obtained, and a machine-checked proof that the hardware security mechanisms indeed implement the sought-for security property.**

Formally proving security properties about hardware systems might seem prohibitively complex and expensive. In this paper, we tackle this concern by designing a realistic and accessible methodology on top of the Kôika Hardware Description Language for specifying and proving security properties during hardware development. Our methodology is centered around a verified compiler from high-level and inefficient to work with Kôika models to an equivalent lower-level representation, where side effects are made explicit and reasoning is convenient.

We apply this methodology to a concrete example: the formal specification and implementation of a shadow stack mechanism on an RV32I processor. We 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. Furthermore, we show that this modification of the processor does not impact its behaviour in other, unexpected ways.

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

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

I. Introduction

Formal methods can be used to build trust in computer systems. The CompCert C compiler [20] or the seL4 microkernel [18] are examples of their successful application to software verification. Formal verification of hardware was historically mostly confined to functional correctness, as exemplified by the formal verification of the floating-point operations in Intel hardware following the discovery of the Pentium FDIV bug [17], or by the verification of protocols, e.g. for cache coherence [31].

However, hardware components also play a fundamental role in establishing security and trust in the whole hardware/-software stack. Being at the lowest level of the stack, they pose several challenges (e.g. complex interactions with software and difficulty for patching) and open many opportunities (e.g. for providing a sound foundation for software security, protected from modifications incurred by software execution) in terms of security. Formal methods should help with developing a set of formally defined and verified security primitives at the hardware level. Despite this central position in the security foundation of computing platforms, the adoption of formal methods for hardware security is still limited. As stated by several experts of the domain in the 2016 Report on the NSF Workshop on Formal Methods for Security, there is a critical need for "methods to create, analyze, and execute formal hardware security specifications" and "new security mechanisms are needed to ease the creation and verification of higher-level system security properties" [8]. Establishing security properties at the hardware level is also a mandatory stepping stone towards whole-system guarantees. In this article, we propose a solution to this main challenge: providing a solid, scalable and generic foundation for implementing and proving hardware security mechanisms at the microarchitectural level.

Some promising approaches for formally specifying hardware-based security mechanisms and proving that they enforce some security properties have been proposed in the research community. For example, CHERI [32] is an extended version of the Arm ISA, which includes constructs for fine-grained memory protection with capabilities. Recently, the use of these capabilities (the security mechanism) in the Morello architecture was shown to enforce memory protection (the security property) [4], [24]. These properties were proved at the ISA (Instruction Set Architecture) level. However, we aim at reasoning at the microarchitectural level for two reasons. First, the ISA level prevents us from capturing low-level implementation details that may induce vulnerabilities, e.g. side channels. Second, the security properties proved at the ISA level still rely on the correct implementation of this ISA by a microarchitectural model.

Some works in formal methods for hardware verification [1], [6], [15], [21] are based on models of the actual hardware. While this makes proofs easier to come up with, this also opens a possibility for discrepancies between the model and the actual hardware, thus giving a false sense of security.

A way of avoiding this issue would be to reason directly on the model used for production. However, most Hardware Description Languages (HDLs) used in industrial settings are not equipped with formal semantics, and formal proof cannot directly be applied on the models written in these HDLs.

The challenge we tackle in this article is the following: how to provide a solid, scalable and generic foundation for implementing and proving hardware security mechanisms at the microarchitectural level?

This article proposes a realistic and accessible methodology for specifying and proving security properties during hardware development. We base our work on the Kôika language [5], a HDL embedded in the Coq proof assistant. This language comes with a compiler that produces Verilog code, which can be synthesized for FPGA boards. The Kôika project also provides a 4-stage RISC-V processor, among several examples of Kôika models.

The Kôika compiler is formally verified, which means that the compilation of Kôika models down to Verilog preserves their semantics. However, the authors of Kôika do not provide scalable tools for proving non-trivial security properties on Kôika models. Moreover, naive attempts at reasoning directly on complex models written in Kôika, e.g., on a RISC-V processor, lead to prohibitive performance issues due to the large size of the terms manipulated by Coq, among other things.

To overcome these shortcomings, we expand upon the Kôika language. First, we introduce a low-level representation which is better suited to formal reasoning and enables interactive proof on hardware models. We propose a compiler from Kôika models to this low-level representation and prove that this compiler preserves semantics, i.e. properties about the Kôika model can be proved by reasoning about the low-level representation. Then, we develop a library of verified transformation passes over these representations in order to ease reasoning. For instance, we offer transformation passes to exploit hypotheses about the values of some registers at the beginning of a cycle or to replace arithmetic operations with their results when enough of their arguments are known. A mix of transformation passes applications and case analysis is sufficient to discharge the goals we are interested in.

We rely on our methodology to verify that a security mechanism targeting the aforementioned Kôika RISC-V model enforces some security policies. More precisely, we prove that a hardware-based return address shadow stack (the security mechanism) prevents the modification of return addresses (the security property), which guards against arbitrary code execution.

Our contributions can be summarized as follows:

- We propose a methodology and a framework based on Kôika to specify and implement security mechanisms and the security properties they are supposed to enforce;
- We define a low-level representation for Kôika that is more amenable to verification;
- We design a series of code transformations that simplify the models;
- We showcase this methodology on a hardware-based shadow stack security mechanism, and prove that it indeed enforces the integrity of return addresses, and does not affect the legitimate behaviour of the computer system;
- We synthesize the aforementioned processor equipped with a hardware shadow stack and run it successfully on an FPGA board.

Our contributions have been mechanised in Coq and are available at: https://gitlab.inria.fr/cidre-public/koika-llr.

The rest of this article is organized as follows. Section II introduces the Kôika language [5] and its semantics, showing the challenges that have to be overcome for formal verification. Section III, the main part of this work, introduces our proposed framework and proof methodology. In particular, we detail the modifications we implemented in the Kôika project. Section IV showcases an application of this methodology to the implementation of a verified hardware shadow stack. Section V presents the experimental evaluation of our solution. Section VI position our approach against related work. Section VII discusses limitations and future work, and Section VIII concludes this article.

II. KÔIKA

Kôika [5] is an open-source formal Hardware Description Language (HDL). Being embedded in Coq, this language provides a sound basis for reasoning about the behaviour of hardware models. Kôika is based on BlueSpec [25], a general-purpose, high-level, non-formal HDL. The distinctive feature of Bluespec, compared to other traditional HDLs such as Verilog or VHDL, is that the model is expressed in rules that should run concurrently, and it is the compiler's duty to automatically generate the control logic that ensures rules do not conflict. This way of describing hardware is particularly convenient for concurrent systems such as pipelined processors.

The Kôika compiler is formally verified and outputs Verilog code. One can thus apply Verilog-compatible tools (e.g., simulators and FPGA bitstream generators) to Kôika models. Efficient simulation is possible using the project's custom simulator "Cuttlesim" [26] directly on Kôika models rather than traditional Verilog simulators. Kôika lends itself better to efficient simulation than Verilog because its form is close to regular software, which allows for the application of standard software optimization techniques.

A. The Kôika language

1) Syntax: The syntax of Kôika programs is given by Figure 1. A Kôika program manipulates a set of registers, that form the microarchitectural state. Programs are composed of a set of rules and a schedule that determines the order in

$$
\begin{array}{rrcl}
\text{Actions} & a & ::= & v \mid x \mid \texttt{skip} \\
 & & \mid & \texttt{read}\ r \mid \texttt{write}\ r\ a \\
 & & \mid & \texttt{let}\ x\ =\ a\ \texttt{in}\ a \\
 & & \mid & x := a \\
 & & \mid & \texttt{if}\ a\ \texttt{then}\ a\ \texttt{else}\ a \\
 & & \mid & f(a, \ldots, a) \mid a; a \\
 & & \mid & \texttt{abort} \\
\text{Registers} & r & & \\
\text{Variables} & x & &
\end{array}
$$

$$
\begin{array}{rrcl}
\text{Program} & P & ::= & [\texttt{rule}\ name\ =\ a]* \\
 & & + & \texttt{schedule}\ = \overrightarrow{name} \\
 & & & \\
\text{Values} & val & ::= & \vec{b} \mid \{(k : val)*\} \mid \{val*\} \mid \texttt{enum\_val} \\
\text{Types} & \tau & ::= & \texttt{Bits}\ n \mid \texttt{Struct}\ [(k, \tau)*] \\
 & & \mid & \texttt{Array}\ n\ \tau \mid \texttt{Enum}\ [\texttt{variant}*]
\end{array}
$$

Fig. 1: Syntax of Kôika models

which rules should appear to execute. The rules are atomic actions describing transitions from one clock cycle to the next. Each rule approximately maps to one stage of a pipelined processor. Note that the hardware circuit will execute the rules concurrently during one single cycle, but the schedule helps to define their semantics as if they were executed sequentially.

Kôika actions produce values val, which can be bitvectors, structures, arrays or enums (see Figure 1). Each of these values is given a type, which describes respectively the size of the bitvector, the layout of the structure, the set of values of the enum and the type and number of elements of the array.

Actions can be constant values val, variables x, `skip` actions which do nothing, read of registers `read` r, write of values to registers `write` r a, variables bindings with the `let .. in ..` construct, variable assignments with the $x := a$ construct, conditional expressions with the construct `if .. then .. else ..`, function calls ($f(..)$), sequences of actions and `abort` actions which always fail. Function calls can be internal (defined themselves in terms of actions) or external (their semantics is given by global parameters of the model, and can be used e.g. to model the external RAM).

2) Semantics: The successful executions of actions produce a value alongside a log storing a sequence of read and write events on registers. The semantics is given in Figure 2, by way of a set of judgments of the form $\Gamma \vdash (l, a) \downarrow^{\sigma}_{L} (l', v, \Gamma')$, where Γ is an environment for `let`-bound variables, l is a "rule log", i.e. a log of events that occurred so far during the execution of the current rule, a is the action that is to be executed; and l' is an updated rule log, v is the value computed by action a and Γ' is an updated environment. The environment Γ is a stack of pairs (x, v) where x is a variable name and v is the value associated with that variable. It is updated in particular in rule BIND. We first add a binding to the environment ($(x, v_1) :: \Gamma'$), and finally remove it when the variable goes out of scope (`tl` Γ''), where `tl` is the

Semantics of actions:

$$\text{CST}\quad \frac{v \in val}{\Gamma \vdash (l, v) \downarrow (l, v, \Gamma)} \qquad \text{VAR}\quad \frac{\Gamma(x) = v}{\Gamma \vdash (l, x) \downarrow (l, v, \Gamma)}$$

$$\text{BIND}\quad \frac{\Gamma \vdash (l, a_1) \downarrow (l', v_1, \Gamma') \qquad (x, v_1) :: \Gamma' \vdash (l', a_2) \downarrow (l'', v, \Gamma'')}{\Gamma \vdash (l, \textbf{let}\ x = a_1\ \textbf{in}\ a_2) \downarrow (l'', v, \texttt{tl}\ \Gamma'')}$$

$$\text{ASSIGN}\quad \frac{\Gamma \vdash (l, a) \downarrow (l', v, \Gamma')}{\Gamma \vdash (l, x := a) \downarrow (l', tt, \Gamma'[x \mapsto v])}$$

$$\text{COND-TRUE}\quad \frac{\Gamma \vdash (l, a_1) \downarrow (l', [\texttt{true}], \Gamma') \qquad \Gamma' \vdash (l', a_2) \downarrow (l'', v, \Gamma'')}{\Gamma \vdash (l, \texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3) \downarrow (l'', v, \Gamma'')}$$

$$\text{COND-FALSE}\quad \frac{\Gamma \vdash (l, a_1) \downarrow (l', [\texttt{false}], \Gamma') \qquad \Gamma' \vdash (l', a_3) \downarrow (l'', v, \Gamma'')}{\Gamma \vdash (l, \texttt{if}\ a_1\ \texttt{then}\ a_2\ \texttt{else}\ a_3) \downarrow (l'', v, \Gamma'')}$$

$$\text{FUNCALL}\quad \frac{\Gamma_0 \vdash (l_0, a_1) \downarrow (l_1, v_1, \Gamma_1) \quad \ldots \quad \Gamma_{n-1} \vdash (l_{n-1}, a_n) \downarrow (l_n, v_n, \Gamma_n) \qquad \texttt{zip}(\texttt{args}\ f, [v_1, \ldots, v_n]) \vdash (l_n, \texttt{body}\ f) \downarrow (l', v, \Gamma')}{\Gamma_0 \vdash (l_0, f(a_1, \ldots, a_n) \downarrow (l', v, \Gamma_n)}$$

$$\text{SEQ}\quad \frac{\Gamma \vdash (l, a_1) \downarrow (l', v_1, \Gamma') \qquad \Gamma' \vdash (l', a_2) \downarrow (l'', v, \Gamma'')}{\Gamma \vdash (l, a_1\ ;\ a_2) \downarrow (l'', v, \Gamma'')}$$

$$\text{READ}\quad \frac{\texttt{may\_read}(L, r)}{\Gamma \vdash (l, \textbf{read}\ r) \downarrow (l{+}{+}[\texttt{rd}(r)], \sigma(r), \Gamma)}$$

$$\text{WRITE}\quad \frac{\Gamma \vdash (l, a) \downarrow (l', v, \Gamma') \qquad \texttt{may\_write}(L, l', r)}{\Gamma \vdash (l, \textbf{write}\ r\ a) \downarrow (l'{+}{+}[\texttt{wr}(r, v)], tt, \Gamma')}$$

$$\begin{array}{rcl}
\texttt{may_read}(L, r) & \triangleq & \texttt{wr}(r, *) \notin L \\
\texttt{may_write}(L, l, r) & \triangleq & \texttt{wr}(r, *) \notin (L{+}{+}l)
\end{array}$$

Semantics of a schedule:

$$\frac{\Gamma_0 \vdash (l_0, a) \downarrow^{\sigma}_{L} (l, v, \Gamma) \qquad (L{+}{+}l, sch) \Downarrow L'}{(L, a :: sch) \Downarrow L'}$$

$$\frac{\Gamma_0 \vdash (l_0, a) \not\downarrow^{\sigma}_{L} \qquad (L, sch) \Downarrow L'}{(L, a :: sch) \Downarrow L'} \qquad \frac{}{(L, [\,]) \Downarrow L}$$

Fig. 2: Semantics of Kôika actions and schedules

function that returns a list's tail. The rule for function calls creates a new environment for the called function with the `zip` function that combines two lists into a list of pairs. `args` f and `body` f give, respectively, the names of the arguments and the body of a function f. Moreover, these judgments use two components that are never updated in the semantics of actions: an environment σ that gives the value of each register at the beginning of the cycle, and a "schedule log" L that contains the read and write events produced by previous rules (as defined by the schedule) during the same cycle. For the sake of clarity, when there is no ambiguity about which σ and L should be used, we simply write $\downarrow$ instead of the full $\downarrow^{\sigma}_{L}$. The events in the schedule log and rule log are of the form $\texttt{rd}(r)$ or $\texttt{wr}(r, v)$, denoting respectively a read on register r, and a write of value v on register r. Operator $++$ is the concatenation of lists. The separation between the schedule log and the rule log is necessary because the effects of rules may be cancelled in the presence of conflicts between register reads and register writes. More precisely, a conflict happens in two situations: *read-after-write* (see rule READ in Figure 2), i.e. when a read on a register happens after a write on the same register by a previous rule (as defined by the scheduler); or *write-after-write* (see rule WRITE in Figure 2), i.e. when a write happens after another write on the same register by a previous rule or the same rule. The conditions under which reads and writes are permitted are decided by the `may_read` and `may_write` functions. Note that reading a register after a write has occurred within the same rule is allowed, and results in reading the value of the register at the beginning of the cycle.

The *read-after-write* conflict exists in order to respect the One-Rule-At-A-Time (ORAAT) semantics, i.e. the parallel execution of several rules is allowed if it cannot be distinguished from a sequential execution of these rules, in the order described by the user-provided schedule. Consider the rules `A: write r1 0` and `B: read r1`, and assume register `r1` initially holds the value 1. With the schedule $[A; B]$, rules A and B cannot run concurrently. On the other hand, with the schedule $[B; A]$, the two rules may execute concurrently and will result in the register `r1` having value 0.

The *write-after-write* conflict is not necessary to satisfy ORAAT, but is rather a design choice of the authors of Kôika, who consider that overwriting registers by shadowing them is an antipattern. For instance, the rule `C: write r1 1; write r2 2; write r1 3` would write twice to the same register.

In the bottom of Figure 2, we introduce three rules that describe the semantics of a schedule, built on top of the semantics of actions. We run each rule with an initially empty environment Γ_0 and an empty rule log l_0. We write $\Gamma \vdash (l, a) \not\downarrow$ to denote that the execution of action a fails, i.e. does not result in a triple (l', v, Γ'), e.g. because of a conflict. Conflicts result in the cancellation of all the effects of the conflicting rule. In the example of rule C, this means that not only none of the writes to r_1 actually happen, but the write to r_2 is also ignored, because rules are considered to be atomic, i.e. execute entirely or not at all.

These conflicts have to be detected at run-time, because the conflicting register reads or writes may be nested inside conditional expressions, and thus cannot be decided during compilation.

Kôika's compiler takes care of introducing the appropriate control logic in the generated circuit, in order to run rules in parallel, and only *commit* each rule's effects when there are no conflicts. 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.

Importantly, the presence of a conflict in a rule does not mean that something is wrong with the rule. For instance, in the following rule, a conflict occurs only if both `a` and `b` are equal to 0 at the beginning of a cycle (write-after-write on `x`).

```
Rule A:
  if (read a == 0) then write x 0;
  if (read b == 0) then write x 1.
```

However, in all other situations, the rule runs. The set of rules which end up running in a given cycle depends on the initial value of the registers. These values are stored in an association map which we call the environment. The constant changes in the environment throughout the execution of a model give rise to a dynamic system. For instance, in the context of pipelined systems such as processors, where rules correspond to individual stages of the pipeline, rule cancellations can represent stalls in the pipeline.

Finally, the semantics of a cycle is given by a function `interp_cycle`, taking as parameters a schedule sch and an initial state for registers σ, and producing as a result a new state for registers. For instance $\texttt{interp\_cycle}(sch, \sigma)(r)$ is the value of register r at the end of a cycle.

3) Ports: In the language we described so far, it is not possible to pass information from one rule to another during a single cycle, because one rule cannot read a value written by a previous rule. However, since this impacts performance dramatically, it is possible in Kôika to forward data from one rule to another through the use of ports. Ports (see Figure 3) are simply 0 or 1 and correspond loosely to versions of the associated register across time. Reads and writes are associated with a specific port. One can only read the value of a register on port 0 if no write has occurred on that register in the current cycle, but we can read on port 1 if no write has occurred on port 1. If a write has occurred on port 0, a read on port 1 will retrieve the value that was written, no matter whether the read and write occur in the same rule or successively.

The conflicts between reads and writes is now more subtle: reads on some port p are only allowed if no write has already occurred in a previous rule on any port $p' \geq p$. Writes on some port p are allowed neither after writes on ports $p' \geq p$ nor after reads on ports $p' > p$. This last part means that, for example, once a read on port 1 occurred, we cannot have a write on port 1.

$$
\begin{array}{rrcl}
\text{Ports} & p & ::= & 0 \mid 1 \\
\text{Actions} & a & ::= & \ldots \\
 & & \mid & \texttt{read}_p\ r \mid \texttt{write}_p\ r\ a
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{may\_read}(L, r, \texttt{P0}) & \triangleq & \texttt{wr}(r, *, *) \notin L \\
\texttt{may\_read}(L, r, \texttt{P1}) & \triangleq & \texttt{wr}(r, \texttt{P1}, *) \notin L \\
\texttt{may\_write}(L, l, r, \texttt{P0}) & \triangleq & \texttt{wr}(r, *, *) \notin (L{+}{+}l) \\
 & & \wedge\ \texttt{rd}(r, \texttt{P1}) \notin (L{+}{+}l) \\
\texttt{may\_write}(L, l, r, \texttt{P1}) & \triangleq & \texttt{wr}(r, \texttt{P1}, *) \notin (L{+}{+}l)
\end{array}
$$

Fig. 3: Syntax and semantics of Kôika models with added ports

```
Registers : {r, r0, ..., r20}.

Rule tick :
  write r0 0;
  write r1 0;
  ...
  write r20 0;
  write r (read r + 1).

Schedule : [tick].
```

Fig. 4: A simple model with many independent writes

The notion of ports is crucial for the performance of hardware designs, but it adds some complexity to the semantics of Kôika. Ports will be ignored in the rest of this article for the sake of simplicity. However, they are fully supported in the implementation.

B. Limitations

Although it is possible to prove some properties about the behavior of some simple Kôika circuits using the language in its current state, this becomes impractical as the models grow in complexity, for performance reasons. This issue can't be pinned on a single cause. Rather, it is the consequence of design choices in the Kôika language and of properties of the Coq language, in which the reasoning is carried out.

First, the semantics of Kôika is inherently non-modular. Consider for instance a single register write inside one of the rules of a Kôika model. In order to determine whether this register write actually occurs, one needs to check whether this rule is actually run or will be cancelled for the current cycle. Cancellation conditions, which depend on the evaluation of all the previous rules, can get quite involved. Not only is this tricky to reason with, it can also degrade the performance of proofs involving complex models.

Another limiting factor is related to the techniques for proofs by computation in Coq. When considering the behavior of programs with concrete inputs, efficient Coq tactics for complete normalization such as `cbv` or `vm_compute` can be applied. However, for programs whose inputs are not all known, this approach amounts to complete symbolic execution and leads all too often to a combinatorial explosion. On the other hand, tactics such as `cbn` or `simpl` are an apt tool for reasoning about program whose inputs are at least partially abstract. Indeed, these tactics can be parameterized to try and avoid running into the performance traps that would block `cbv` or `vm_compute`. However, the existing level of control for these tactics is not sufficient for our purposes.

We introduce an example of a Kôika model in Figure 4. It contains a single rule with a sequence of writes, each of which targets a different register. The model contains a total of twenty-one registers. Real-life examples usually contain more registers and actions than this, and include more complex constructs (e.g. conditionals and let-defined variables). Note that there can't possibly be a conflict in this rule as all writes target different registers.

Consider the following property about this model, which says that register the initial and final values of r are distinct:

$$\texttt{interp\_cycle}([\texttt{tick}], \sigma)(r) \neq \sigma(r)$$

It should be rather easy to demonstrate. In fact, the first twenty writes can safely be ignored, as they don't have any influence on the final value of r, which is the only register considered in the property. The value of r is incremented on each cycle, which means that the property is trivially true.

However, there is no straightforward way of implementing this proof using vanilla Kôika and Coq. Even for such a simple program and property, we did not manage to find a proof which terminates in less than twenty minutes. This problem only gets worse when considering larger models with more registers, actions and complex constructs.

We present our methodology to find a way around Coq's and Kôika's performance issues when reasoning on large models in Section III.

III. Proving semantic properties on Kôika models

In order to circumvent the limitations exposed earlier, we propose a low-level representation which makes all the computations about conflicts and rule cancellation explicit. While this produces rather large terms, reifying these computations makes it possible to simplify and store them. This way, for instance, the decision of whether a rule is cancelled is computed once for all, rather than each time we need to reason about the value of any register.

An overview of our methodology for proving security properties on Kôika models is given in Figure 5. In this figure, starting from the Kôika model, we obtain a first low-level representation (LLR) from a verified compiler that we describe in Section III-B. Then, this LLR0 is transformed into a sequence of LLRs, each simpler to reason about than the previous one. Applying these transformations is part of the proof development. It can be done either manually or through tactics which automatically apply appropriate passes depending on the form of the goal and of the hypotheses. Users can easily define additional tactics tailored to their work using Coq's built-in facilities. We describe these transformations in

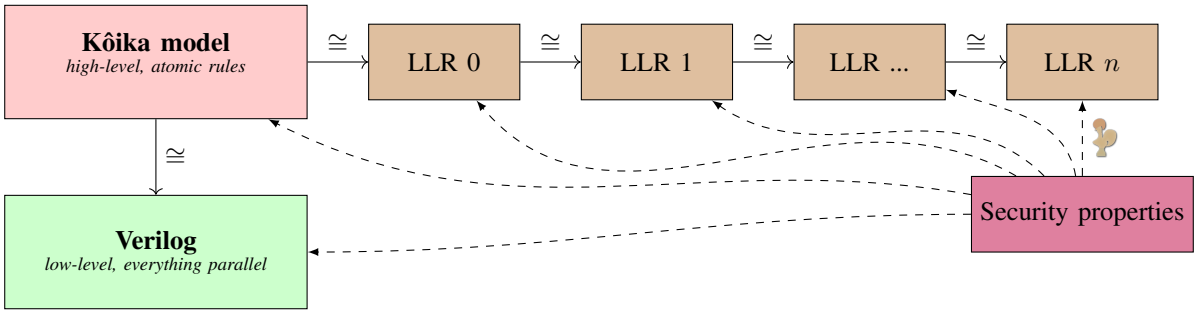

Fig. 5: Overall structure of our proofs on Kôika models

Low-Level Actions

$$\begin{array}{rcl} lla & ::= & v \in val \mid vx \mid r \mid \triangleright lla \mid lla \bowtie lla \\ & \mid & lla \ ?\ lla : lla \end{array}$$

Low-Level Representation

$$llr \quad ::= \quad \{V : \mathbb{N} \hookrightarrow lla \ ;\ R : \texttt{reg} \to \mathbb{N}\}$$

Semantics of Low-Level Actions

$$\begin{array}{rcl} [\![v]\!]^{llr}_{\sigma} & := & v \quad (v \in val) \\ [\![vx]\!]^{llr}_{\sigma} & := & [\![a]\!]^{llr}_{\sigma} \text{ if } llr.V(x) = \lfloor a \rfloor \\ [\![r]\!]^{llr}_{\sigma} & := & \sigma(r) \\ [\![\triangleright lla]\!]^{llr}_{\sigma} & := & \triangleright [\![lla]\!]^{llr}_{\sigma} \\ [\![lla_1 \bowtie lla_2]\!]^{llr}_{\sigma} & := & [\![lla_1]\!]^{llr}_{\sigma} \bowtie [\![lla_2]\!]^{llr}_{\sigma} \\ [\![lla_1 \ ?\ lla_2 : lla_3]\!]^{llr}_{\sigma} & := & [\![lla_1]\!]^{llr}_{\sigma} \neq \vec{0} \ ?\ [\![lla_2]\!]^{llr}_{\sigma} \\ & & \quad : [\![lla_3]\!]^{llr}_{\sigma} \end{array}$$

Fig. 6: Syntax and semantics of low-level actions

Section III-D. Each simplification pass is formally verified, hence the proofs we carry out on the transformed LLRs also hold the initial LLR. Finally, the LLR n is simple enough that we can carry out the proof of our security property directly. Because every step in our methodology is proven to preserve semantics, the security property we prove on LLR n also holds for LLR $n-1$, all the way up to LLR 0, but also for the Kôika model. From Kôika's compiler correctness, we have that the security property holds on the Verilog code that will be simulated or synthesized.

We propose a compiler from Kôika models to these lower-level representations (LLR) and prove that this compiler preserves semantics, i.e. properties about the Kôika model can be proved by reasoning about the LLR. Then, we apply a series of verified simplifications that depends on the specific goal we try to prove.

A. A low-level representation

The type of the low-level representation we target is given in Figure 6. A low-level representation (LLR) is composed of a variable map V, which maps variable identifiers to low-level actions, and a mapping R from register names to variable identifiers. $R[r]$ holds the identifier of the variable that corresponds to the value of register r after one clock cycle, whose contents can be recovered from the variable map V.

A low-level action is a static counterpart to Kôika actions. These low-level actions can be constant values v, variables vx, registers r, unary operations $\triangleright lla$, binary operations $lla \bowtie lla$, or conditional expressions $lla \ ?\ lla \ : \ lla$, as illustrated in Figure 6. The main difference, compared with Kôika actions, is that register read and write operations have disappeared, and actions are now free of side effects.

The low-level action associated with a variable x in our low-level representation is allowed to reference only variables whose identifier y is strictly below x. This is a well-formedness condition that ensures that there is no cyclic dependency between variables. Hence, the evaluation of these expressions terminates. We can therefore define the evaluation function $[\![lla]\!]^{llr}_{\sigma}$, which evaluates a low-level action lla into a value. It is parameterized by a low-level representation llr (for resolving variables) and an environment σ (that holds the initial values of registers). The definition is given in Figure 6.

Let us consider for instance the action $\texttt{a} + v7$. Its evaluation $[\![\texttt{a} + v7]\!]^{llr}_{\sigma}$ would be decomposed into $[\![\texttt{a}]\!]^{llr}_{\sigma} + [\![v7]\!]^{llr}_{\sigma}$. The first part $[\![\texttt{a}]\!]^{llr}_{\sigma}$, i.e. the evaluation of register $\texttt{a}$ will simply be a lookup into the environment: $\sigma(\texttt{a})$. The second part will lookup variable $v7$ in the LLR, and then recursively apply the evaluation function on the low-level action associated with $v7$.

Although the representation introduced in this paper is tailored to languages of the BlueSpec family, similar representations can be more generally applied for controlling partial interpretation in Coq. Indeed, as things stand, there are situations that Coq's interpretation tactics are unable to handle conveniently, as detailed in II-B.

B. From Kôika rules to a Low-Level Representation

Our first objective is to build a low-level representation from a Kôika model. We treat each rule in the Kôika model in the order specified by the schedule. For each rule, we perform a form of abstract interpretation on the action, as shown in the definition of $K2L$ in Figure 7. The parameters of the compiler are: 1) the Kôika action a to be compiled; 2) a projection $\Pi : \texttt{Reg} + \texttt{Var} \to \mathbb{N}$ that maps Kôika variables and registers to LLR variable identifiers; 3) a mapping $V : \mathbb{N} \to lla$ from variable identifiers (natural numbers) to low-level actions; 4) the next fresh variable identifier f; 5) the current path

condition P, as a low-level action; 6) an abstract schedule log $L^\sharp$ and 7) an abstract rule log $l^\sharp$. The result of compiling an action with $K2L$ is a 6-tuple $(lla, \Pi', V', f', F, l'^\sharp)$ where lla is the low-level action corresponding to the input Kôika action, Π', V', f' and $l'^\sharp$ are the updated values of Π, V, f and $l^\sharp$, and F is the action's failure condition, i.e. a low-level action expressing the conditions under which the current action will fail. The abstract logs record all the potential read and write events, together with a low-level action representing the condition under which these read and write events actually occurred. More precisely, abstract logs ($L^\sharp$ and $l^\sharp$) are of type $\{\texttt{wr}, \texttt{rd}\} \times \texttt{Reg} \to lla$. For example, $l^\sharp(\texttt{wr}, r)$ is a low-level action that represents the conditions under which a write has occurred on register r. The logs are updated when compiling register reads and writes (see Figure 7): the notation $l^\sharp[(k, r) \overset{\vee}{\mapsto} P]$ means the mapping $l^\sharp$ updated for key (k, r), where the new value is $old \vee P$, where old is the old value of the mapping for that key. The $\texttt{may\_read}^\sharp$ and $\texttt{may\_write}^\sharp$ are the abstract counterpart to the may_read and may_write in Kôika semantics. For instance, $\texttt{may\_read}(L, r) = \texttt{wr}(r, *) \notin (L)$ (see Figure 1). The abstract version $\texttt{may\_read}^\sharp(L^\sharp, r)$ is defined as the low-level action $\neg L^\sharp(\texttt{wr}, r)$. The condition under which a read action fails is exactly the negation of the result of the $\texttt{may\_read}^\sharp$ function. For an action write r a, it is the logical disjunction between the negation of $\texttt{may\_write}^\sharp$ and the failure condition associated with action a. The projection Π maps Kôika variables and registers to LLR variable identifiers. More precisely, for each register r, $\Pi(r)$ corresponds to the variable that will hold the final value of register r, at the end of the current cycle. Initially, $\Pi(r)$ maps to a variable that only contains the low-level action r. When compiling a register write (see Figure 7), the projection Π is updated so that $\Pi(r)$ points to the new value, in accordance with the semantics of Kôika.

One of the most involved cases for $K2L$ is that of conditional expressions. We construct a low-level representation for each case of the conditional expression: one when the condition evaluates to true, another when the condition evaluates to false. This is materialized by the two recursive calls to $K2L$, one with the path condition augmented with the condition, the other with its negation. These two calls generate two distinct projections Π_{tb} and Π_{fb}, that we merge using the binary operator $\bigcup_{V,f}^{v_{cond}}$. For every Kôika variable or register k, suppose $\Pi_1(k) = v_1$ and $\Pi_2(k) = v_2$, then $\Pi_1 \bigcup_{V,f}^{v_{cond}} \Pi_2$ will be a triple (V', Π_r, f') such that $\Pi_r(k) = n$ and $[\![n]\!]_\sigma^{V'} = [\![v_{cond} \; ? \; v_1 : v_2]\!]_\sigma^{V'}$. For instance, let us consider the first conditional expression in rule r1 from Figure 8a. Recursively analyzing the then and else branches yield two different LLRs: one when **read** a == 0 holds and register a receives the value 1, the other where register a keeps its previous value. The merged LLR will associate to register a the low-level action $(a == 0) \; ? \; 1 : a$.

Next, depending on whether conflicts occur or not, the effect of rules are discarded or applied. The function $K2L^{sched}$ compiles a Kôika schedule into a LLR. Each rule is treated

$$
\begin{array}{l}
K2L(a, \Pi, V, f, P, L^\sharp, l^\sharp) = \\
\quad \textbf{match } a \textbf{ with} \\
\quad | \;\; v \to (v, \Pi, V, f, \texttt{false}, l^\sharp) \\
\quad | \;\; vx \to \textbf{let } v = \Pi(vx) \textbf{ in } (v, \Pi, V, f, \texttt{false}, l^\sharp) \\
\quad | \;\; \texttt{read } r \to \textbf{let } v = \Pi(r) \textbf{ in} \\
\qquad \textbf{let } mr = \texttt{may\_read}^\sharp(L^\sharp, r) \textbf{ in} \\
\qquad (v, \Pi, V, f, \neg mr, l^\sharp[(\texttt{rd}, r) \overset{\vee}{\mapsto} P]) \\
\quad | \;\; \texttt{write } r \; a \to \\
\qquad \textbf{let } (lla, \Pi, V, f, F_a, l^\sharp) = \\
\qquad\quad K2L(a, \Pi, V, f, P, L^\sharp, l^\sharp) \textbf{ in} \\
\qquad \textbf{let } mw = \texttt{may\_write}^\sharp(L^\sharp, l^\sharp, r) \textbf{ in} \\
\qquad \textbf{let } (V, \Pi, f) = (V[f \mapsto lla], \Pi'[r \mapsto f], f + 1) \textbf{ in} \\
\qquad (\bot, \Pi, V, f, F_a \vee \neg mw, l^\sharp[(\texttt{wr}, r) \overset{\vee}{\mapsto} P]) \\
\quad | \;\; \texttt{if } c \texttt{ then } ta \texttt{ else } fa \to \\
\qquad \textbf{let } (lla_{cond}, \Pi, V, f, F_{cond}, l^\sharp) = \\
\qquad\quad K2L(c, \Pi, V, f, P, L^\sharp, l^\sharp) \textbf{ in} \\
\qquad \textbf{let } (v_{cond}, f) = (f, f + 1) \textbf{ in} \\
\qquad \textbf{let } V = V[v_{cond} \mapsto lla_{cond}] \textbf{ in} \\
\qquad \textbf{let } (lla_{tb}, \Pi_{tb}, V, f, F_{tb}, l^\sharp) = \\
\qquad\quad K2L(ta, \Pi, V, f, P \wedge v_{cond}, L^\sharp, l^\sharp) \textbf{ in} \\
\qquad \textbf{let } (lla_{fb}, \Pi_{fb}, V, f, F_{fb}, l^\sharp) = \\
\qquad\quad K2L(fa, \Pi, V, f, P \wedge \neg v_{cond}, L^\sharp, l^\sharp) \textbf{ in} \\
\qquad \textbf{let } (\Pi_{merge}, V, f) = \Pi_{tb} \bigcup_{V,f}^{v_{cond}} \Pi_{fb} \textbf{ in} \\
\qquad (lla_{cond} \; ? \; lla_{tb} \; : \; lla_{fb}, \Pi_{merge}, V, f, \\
\qquad\quad F_{cond} \vee (lla_{cond} \; ? \; F_{tb} \; : \; F_{fb}), l^\sharp) \\
\quad | \;\; x \;:=\; a \to \\
\qquad \textbf{let } (lla, \Pi, V, f, F, l^\sharp) = \\
\qquad\quad K2L(a, \Pi, V, f, P, L^\sharp, l^\sharp) \textbf{ in} \\
\qquad \textbf{let } (\Pi, V) = (\Pi[x \mapsto f], V[f \mapsto lla]) \textbf{ in} \\
\qquad (\bot, \Pi, V, f + 1, F, l^\sharp) \\
\quad | \;\; \texttt{let } x \;=\; a \texttt{ in } body \to \\
\qquad \textbf{let } (lla, \Pi, V, f, F_a, l^\sharp) = \\
\qquad\quad K2L(a, \Pi, V, f, P, L^\sharp, l^\sharp) \textbf{ in} \\
\qquad \textbf{let } (\Pi, V) = (\Pi[x \mapsto f], V[f \mapsto lla]) \textbf{ in} \\
\qquad \textbf{let } (lla_{body}, \Pi, V, f, F_{body}, l^\sharp) = \\
\qquad\quad K2L(body, \Pi, V, f + 1, P, L^\sharp, l^\sharp) \textbf{ in} \\
\qquad (lla_{body}, \Pi[x \mapsto \bot], V, f, F_a \vee F\_body, l^\sharp)
\end{array}
$$

$$
\begin{array}{l}
K2L^{sched}(s, \Pi, V, f, L^\sharp) = \\
\quad \textbf{match } s \textbf{ with} \\
\quad | \;\; [] \to (\Pi, V, f, L^\sharp) \\
\quad | \;\; r :: s \to \\
\qquad \textbf{let } (\_, \Pi', V, f, F_r, l^\sharp) = \\
\qquad\quad K2L(r, \Pi, V, f, \texttt{true}, L^\sharp, l_0^\sharp) \textbf{ in} \\
\qquad \textbf{let } (v_{conflict}, f) = (f, f + 1) \textbf{ in} \\
\qquad \textbf{let } V = V[v_{conflict} \mapsto F_r] \textbf{ in} \\
\qquad \textbf{let } (\Pi_{merge}, V, f) = \Pi \bigcup_{V,f}^{v_{conflict}} \Pi' \textbf{ in} \\
\qquad \textbf{let } L^\sharp_{merge} = \lambda k \to L^\sharp(k) \vee (l^\sharp(k) \wedge \neg v_{conflict}) \textbf{ in} \\
\qquad K2L^{sched}(s, \Pi_{merge}, V, f, L^\sharp_{merge})
\end{array}
$$

$$
\begin{array}{l}
K2L^P(s) = \\
\quad \textbf{let } (\Pi, V, \_, \_) = K2L^{sched}(s, \Pi_0, V_0, 1, L_0^\sharp) \textbf{ in} \\
\quad \{ \; R := \texttt{fun } r \Rightarrow \Pi(r); \;\; V := V \; \}
\end{array}
$$

Fig. 7: The Kôika to LLR compiler

sequentially. For each rule, we call $K2L$ with the current projection Π ant the current set of variables V, the current abstract schedule log $L^\sharp$ and an empty initial rule log $l_0^\sharp$. The failure condition F_r gives us the condition under which the rule failed. Like in the conditional expression case, we need to merge the projections using $\bigcup_{V,f}^{v_{conflict}}$. We also need to patch the abstract schedule log $L^\sharp_{merge}$ as described in Figure 7 so that the condition under which a read or write occurred is the disjunction of whether it already occurred in the previous schedule log $L^\sharp$ and whether it occurred in the new rule log $l^\sharp$, while not having a conflict. This step was not needed in the if-then-else case because we injected the condition (or its negation) in the path condition, which we cannot do here because the failure condition is the result of the compiler itself.

Figures 8b and 8c illustrate a complete example of compiling a Kôika schedule into a LLR. Initially, we have one variable for each of the three registers `a`, `b` and `c`. The let-bound variable `x` in rule `r1` gives rise to $v4$ in the LLR, whose value is $v2$, the initial value of register `b`. The conditions of if-then-else actions also produce variables ($v5$, $v8$ and $v13$ in our example). For each register write, a variable is created that contains the low-level action being written ($v6$, $v9$ and $v14$), and yet another variable is created that contains the new value of the register, often an if-then-else low-level action dependent on the path condition that led us to the write ($v7$, $v10$, and $v15$). At the end of rule `r1`, $v11$ contains the rule's failure condition, i.e. $v8 \wedge v5$, i.e. "there have been two writes on `a`"; and $v12$ contains the value of register `a` after rule `r1`: "if the rule fails, the value of `a` is the same as at the beginning of the cycle, otherwise it is the value after the second **`if`**". Similarly, $v16$ contains the failure condition for rule `r2`: "the rule fails if a write occurred in this rule (condition $v13$) and a write occurred in rule `r1` ($\neg v11 \wedge (v5 \vee v8)$, i.e. rule `r1` did not fail and one of the writes occurred)".

C. Correctness of the Kôika to LLR compiler

In order to reason on the LLR, we must ensure that our compiler is correct. This section walks through a series of definitions and lemmas that culminate in the proof of Theorem 2, which enables to reason about a LLR to obtain properties about a Kôika model.

We first define a relation $\overset{log}{\sim}$ between a concrete and an abstract log, which establishes that a read or write event occurs in the concrete log if and only if the condition associated to that event in the abstract log evaluates to true.

$$L \overset{log}{\sim}_V L^\sharp \triangleq \begin{cases} \forall r,\ \texttt{wr}(r,*) \in L \iff [\![L^\sharp(\texttt{wr}, r)]\!]^V_\sigma = \texttt{true} \\ \forall r,\ \texttt{rd}(r) \in L \iff [\![L^\sharp(\texttt{rd}, r)]\!]^V_\sigma = \texttt{true} \end{cases}$$

We also define a relation $\overset{reg}{\sim}$ between a concrete log and a LLR. This relation states that the projection Π sends registers to LLR variables that evaluate the same as performing a read on that register in a Kôika action. This is captured by the $\texttt{do\_read}(l, \sigma, r)$ function, which returns a value v if $\texttt{wr}(r, v) \in l$, or $\sigma(r)$ if no such write occurred.

$$l \overset{reg}{\sim} (\Pi, V) \triangleq \forall r,\ \exists n,\ \Pi(r) = n \wedge [\![n]\!]^V_\sigma = \texttt{do\_read}(l, \sigma, r)$$

We now define a matching relation between Kôika states (Γ, L, l) (variable environment, schedule log and rule log) and LLR states $(V, L^\sharp, l^\sharp)$ (variable mapping, abstract schedule log and abstract rule log). This relation, denoted $\sim_\Pi$, is indexed by a projection Π, and states that the concrete and abstract logs are related by $\overset{log}{\sim}_V$, both for the schedule and rule logs; that the projection Π sends Kôika variables to LLR variables that evaluate identically; and that the registers are accurately projected by Π, as per the $\overset{reg}{\sim}$ relation.

$$(\Gamma, L, l) \sim_\Pi (V, L^\sharp, l^\sharp) \triangleq \begin{cases} L \overset{log}{\sim}_V L^\sharp \\ l \overset{log}{\sim}_V l^\sharp \\ \forall x,\ \exists n,\ \Pi(x) = n \wedge [\![n]\!]^V_\sigma = \Gamma(x) \\ (L +\!\!+ l) \overset{reg}{\sim} (\Pi, V) \end{cases}$$

We prove the following lemma, about the correctness of the $K2L$ function, which is the essence of the proof of the compiler correctness theorem.

Lemma 1 (Correctness of $K2L$)**.** *Consider two matching states (Γ, L, l) and $(V, L^\sharp, l^\sharp)$ related by projection $\sim_\Pi$. Compiling action a with path condition P produces a low-level action lla together with a new abstract state $(V', L^\sharp, l'^\sharp)$, and a failure condition F.*

If the Kôika semantics of action a produces a new Kôika state (Γ', v, l'), then the low-level action lla produced by the compiler evaluates to Kôika value v, the failure condition F evaluates to `false`*, and (Γ', L, l') and $(V', L^\sharp, l'^\sharp)$ stay in the matching relation.*

Otherwise, if action a fails according to Kôika semantics, the failure condition produced by the compiler evaluates to `true`*. More formally,*

$$\begin{array}{l} \forall\ \ a\ \Pi\ V\ f\ L^\sharp\ l^\sharp\ \Gamma\ L\ l\ lla\ P\ F\ \Pi'\ V'\ f'\ l'^\sharp\ \Gamma', \\ \quad K2L(a, \Pi, V, f, P, L^\sharp, l^\sharp) = (lla, \Pi', V', f', F, l'^\sharp) \Rightarrow \\ \quad (\Gamma, L, l) \sim_\Pi (V, L^\sharp, l^\sharp) \Rightarrow [\![P]\!]^V_\sigma = \texttt{true} \Rightarrow \\ \quad \left(\begin{array}{l} \forall\ \ l'\ v\ \Gamma', \\ \quad \Gamma \vdash_L (l, a) \downarrow (l', v, \Gamma') \Rightarrow \\ \quad [\![lla]\!]^{V'}_\sigma = v \wedge [\![F]\!]^{V'}_\sigma = \texttt{false} \wedge \\ \quad (\Gamma', L, l') \sim_{\Pi'} (V', L^\sharp, l'^\sharp) \end{array} \right) \\ \quad \bigwedge \left(\ \Gamma \vdash_L (l, a) \not\downarrow \Rightarrow [\![F]\!]^{V'}_\sigma = \texttt{true} \ \right) \end{array}$$

Proof. By structural induction on a. □

We can use this lemma to prove Theorem 1, about $K2L^{sched}$:

Theorem 1 (Correctness of $K2L^{sched}$)**.** *$K2L^{sched}$ is a correct compiler of the semantics of Kôika schedules. More precisely,*

$$\begin{array}{l} \forall\ \ s\ \Pi\ V\ f\ L^\sharp\ \Pi'\ V'\ f'\ L'^\sharp, \\ \quad K2L^{sched}(s, \Pi, V, f, L^\sharp) = (\Pi', V', f', L'^\sharp) \Rightarrow \\ \quad (L, s) \Downarrow L' \Rightarrow \\ \quad L \overset{log}{\sim}_V L^\sharp \wedge L \overset{reg}{\sim} (\Pi, V) \Rightarrow \\ \quad L' \overset{log}{\sim}_{V'} L'^\sharp \wedge L' \overset{reg}{\sim} (\Pi', V') \end{array}$$

Proof. By induction on the schedule s.

```
Registers : {a, b, c}.

Rule r1 :
  let x := read b in
  if read a == 0
    then write a 1;
  if x == 1
    then write a (x + 1).

Rule r2 :
  if read c == 1
    then write a 3.

Schedule : [r1, r2].
```

(a) Kôika model

Register	Variable Id
a	17
b	2
c	3

(b) LLR representation (R)

Id	Value	Description
1	a	initial value of register a
2	b	initial value of register b
3	c	initial value of register c
4	$v2$	let-binding in r1
5	$v1$ == 0	first condition in r1
6	1	value written in the first **if** in r1
7	$v5$? $v6$: $v1$	value of register a after first **if** in r1
8	$v4$ == 1	second condition in r1
9	$v4$ + 1	value written in the second **if** in r1
10	$v8$? $v9$: $v7$	value of register a after second **if** in r1
11	$v8 \wedge v5$	failure condition for r1
12	$v11$? $v1$: $v10$	value of register a after r1
13	$v3$ == 1	first condition in r2
14	3	value written in the **if** in r2
15	$v13$? $v14$: $v12$	value of register a after the **if** in r2
16	$v13 \wedge \neg\, v11 \wedge (v5 \vee v8)$	failure condition for r2
17	$v16$? $v12$: $v15$	final value of register a after r2

(c) LLR representation (V)

Fig. 8: Compilation from Kôika to LLR

Base case. If the schedule is empty, the theorem holds trivially because $(\Pi', V', L'^{\sharp}) = (\Pi, V, L^{\sharp})$ and $L' = L$.

Inductive case. The schedule is of the form $r{::}sch$, and we have as an induction hypothesis that our theorem holds for schedule sch.

By the definition of $K2L^{sched}$, we have that:

1 $(\_, \Pi_1, V_1, f_1, F_r, l^{\sharp}) = K2L(r, \Pi, V, f, \texttt{true}, L^{\sharp}, l_0^{\sharp})$
2 $V_2 = V_1[f_1 \mapsto F_r]$
3 $(\Pi_m, V_m, f_2) = \Pi \bigcup_{V_2, f_1+1}^{f_1} \Pi'$
4 $L_m^{\sharp} = \lambda k \to L^{\sharp}(k) \vee (l^{\sharp}(k) \wedge \neg v_{conflict})$
5 $(\Pi', V', f', L'^{\sharp}) = K2L^{sched}(sch, \Pi_m, V_m, f_2, L_m^{\sharp})$

From the semantics of a Kôika schedule (Figure 2), the next concrete schedule log, L', will be either $L{+}{+}l'$ if the execution of r yields a rule log l', or L if the execution of a fails to produce a rule log. In order to apply our induction hypothesis and finish the proof, all we need to show is that $L' \overset{log}{\sim}_{V_m} L_m^{\sharp} \wedge L' \overset{reg}{\sim} (\Pi_m, V_m)$.

Applying Lemma 1 on line 1 above gives the following:

$$\forall\ \ L,\ L \overset{log}{\sim}_{V} L^{\sharp} \wedge L \overset{reg}{\sim} (\Pi, V) \Rightarrow \left(\begin{array}{l} \forall\ \ l',\ \Gamma_0 \vdash_L (l_0, r) \downarrow (l', \_, \_) \Rightarrow \\ \quad [\![F_r]\!]_{\sigma}^{V_1} = \texttt{false} \wedge \\ \quad l' \overset{log}{\sim}_{V_1} l^{\sharp} \wedge (L \mathbin{+\!+} l') \overset{reg}{\sim} (\Pi_1, V_1) \end{array} \right) \bigwedge \left(\Gamma_0 \vdash_L (l_0, r) \not\downarrow \Rightarrow [\![F_r]\!]_{\sigma}^{V_1} = \texttt{true} \right)$$

By disjunction of cases:

- **Case** $\Gamma_0 \vdash_L (l_0, r) \downarrow (l', \_, \_)$.
  Then, $[\![f_1]\!]_{\sigma}^{V_2} = \texttt{false}$, hence $\Pi_m(k) = \Pi'(k)$ for every k, and $[\![L_m^{\sharp}(k)]\!]_{\sigma}^{V_m} = [\![L^{\sharp}(k) \vee l^{\sharp}(k)]\!]_{\sigma}^{V_m}$ for every k. It follows that $(L{+}{+}l') \overset{log}{\sim}_{V_m} L_m^{\sharp}$, and $(L{+}{+}l') \overset{reg}{\sim} (\Pi_m, V_m)$.
- **Case** $\Gamma_0 \vdash_L (l_0, r) \not\downarrow$.
  Then, $[\![f_1]\!]_{\sigma}^{V_2} = \texttt{true}$, hence $\Pi_m(k) = \Pi(k)$ for every k, and $[\![L_m^{\sharp}(k)]\!]_{\sigma}^{V_m} = [\![L^{\sharp}(k)]\!]_{\sigma}^{V_m}$ for every k. It follows that $L \overset{log}{\sim}_{V_m} L_m^{\sharp}$, and $L \overset{reg}{\sim} (\Pi_m, V_m)$.

□

Finally, the theorem we want to prove relates the interpretation of a cycle and the LLR obtained by compiling the schedule. We define a function $\mathcal{C}_{\sigma}(llr)$ which, given a LLR llr and an initial mapping for registers σ, results in an updated register environment σ'.

$\mathcal{C}_{\sigma}(llr)(r) \triangleq \textbf{let}\ n = llr.R(r)\ \textbf{in}\ [\![n]\!]_{\sigma}^{llr}$

Theorem 2 (Kôika to LLR compiler correctness)**.** *Given a schedule s and an initial state of registers σ, retrieving the final value of a register r through the LLR compiled from this schedule or through the Kôika semantics gives identical results. More formally,*

$$\forall s\ \sigma\ r,\ \mathcal{C}_{\sigma}(K2L^{P}(s))(r) = \texttt{interp\_cycle}(s, \sigma)(r)$$

Proof. By applying Theorem 1 and applying definitions. □

D. Verified transformation passes

In order to use the LLR we produced in the previous section for proving properties, we develop a toolbox of theorems and tactics in Coq for reasoning about circuits in our low-level representation.

Among other things, we define a set of transformation passes that can be applied to a model in the low-level representation. In particular, the transformation passes provide a way

of exploiting hypotheses about the state of the environment. Often, a combination of simplifications and case analysis is sufficient for discharging a goal. Note that the reasoning is carried out inside of Coq, so the usual facilities and methods used in this language remain available at any time.

For instance, consider the model of Figure 9a, composed of a single rule. In this rule, the value of register `b` is updated on every cycle. Suppose we want to prove that when the value of `a` is 0 at the beginning of a cycle, then the value of `b` will be 0 at the end of this cycle. Formally, we would write this as:

$\forall\ \sigma\ llr,\ \sigma(\texttt{a}) == 0 \Rightarrow \mathcal{C}_\sigma(llr)(\texttt{b}) = 0.$

We compile the model of Figure 9a into an equivalent low-level representation, as shown in the second column of Table 9b. This LLR is then processed by a sequence of transformations which progressively simplify it. In this example, we employ the following transformation passes:

- `Prune`(b): remove all variables that are not used in the computation of the final value of register b. We collect all the variables that may participate in the evaluation of register b in our LLR, and prune the others. Here, the register b is represented by $v8$. Variables $v6$ and $v7$ can be pruned.
- `ExploitReg`(r,v): replace register r with value v. Applying this transformation generates a proof obligation, namely that register r holds value v at the beginning of the cycle. In our example, we have the hypothesis that register a holds the value 0.
- `Collapse`: replace all variables with their value if this symbolic value is simple enough (a constant, a reference to another variable or a register). In our example, we replace variables $v1$, $v2$ and $v5$ by their value.
- `Simplify`: compute the value of unary or binary operations when enough of their arguments are known, for all variables in the LLR. This is similar to constant folding in traditional compilers. We also simplify boolean conjunctions or disjunctions when one operand is known to be true or false.

All of our transformation passes preserve the semantics under some assumptions. For instance, assuming that the value of register `a` is known to be `0` at the beginning of a cycle, then replacing its appearances with `0` is correct. Coq verifies that this assumption is met in the current context before it can apply the transformation. Similarly, it would be invalid to evaluate a register with a LLR in which some of the required variables have been pruned away. Therefore, pruning can only be applied in contexts where the final value of a single register is being considered.

We offer additional transformation passes, such as:

- `PruneList`(l): remove all variables that are not used in the computation of the final value of at least one register appearing in list l
- `ReplaceVar`(x, v): replace variable x with value v, given a proof that this substitution is correct
- `ReplaceSubact`(se, v): same as `ReplaceVar` but for a subexpression se. All occurrences of this expression se in any variable of the tree can then be replaced with the given value v.
- `ExploitPartialInformation`: same as `ExploitReg` but works when only some bits of a register are known.

Coq makes it possible to define custom tactics to automate away part of the tedium. We define some 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 can run fully automatically.

IV. A concrete example: implementing a verified shadow stack for a RISC-V processor

In this section, we first explain the security mechanism we aim to implement, i.e. a hardware-based shadow stack in a RISC-V processor. We formally specify the properties we wish to verify, then we give the outline of the proofs we performed, using the methodology introduced in the previous section.

A. A RISC-V processor model in Kôika

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.

B. The shadow stack mechanism

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 return addresses, one of the most popular attacks of this type. Hardware-based security mechanisms implementing Control Flow Integrity, like Intel CET [27], 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 way of guaranteeing 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 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 regular writes to memory performed by application code to modify the shadow stack contents.

```
Registers : {a, b}.

Rule r1 :
  let x := read a in
  let y := read b in
  if x == 0 then
    write b (y - y)
  else
    (write b 1; write a 2).

Schedule : [r1].
```

(a) Example rule

	Initial	Prune (reg a)	ExploitReg	Collapse	Simplify	Collapse + Simplify
1	a		0			
2	b					
3	$v1 == 0$			0 == 0	1	
4	$v2$ - $v2$			b - b		
5	1					
6	2					
7	**if** $v3$ **then** a **else** $v6$					
8	**if** $v3$ **then** $v4$ **else** $v5$			**if** $v3$ **then** $v4$ **else** 1		b - b

(b) Successive transformations

Fig. 9: Example rule, its transformation into a LLR, and successive transformations

Shadow stacks can be implemented either in software [19] 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.

We added a shadow stack module to the processor provided by the Kôika project. 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`.

The `push` function checks for potential overflows and pushes the address onto the shadow stack, whereas the `pop` function checks for potential underflows, verifies that the address passed as argument matches the top of the shadow stack and pops it. If one of these checks fails, we jump to an exception handler.

C. Detecting function calls and returns in machine code

Contrary to CISC (Complex Instruction Set Computer) 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 (*jump and link*) and JALR (*jump and link register*) instructions implement both unconditional jumps and function calls. However, the arguments that are passed to them make their role clear. The Application Binary Interface (ABI) describes which JAL/JALR instructions should be interpreted as function calls or returns, depending on their arguments. In fact, the RISC-V specification[1] 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 $rd = \text{x1/x5}$. 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`.

Hence, we implement our shadow stack so that we push a return address when the destination register is `x1` or `x5`, and we pop when the source register is `x1` or `x5` and the destination register is different from the source register.[2]

D. Dealing with a detected stack buffer overflow

On a system with a full-fledged operating system, a stack buffer overflow detected through a shadow stack mechanism 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, these options are not all open. The two main possibilities for our exception handler 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.

[1] https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf, Unprivileged specification, v.20191213, Sec.2.5

[2] Registers `x1` and `x5` are also respectively known as `ra` (for return address) and `t0`.

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 consider that a system is halted when it is in a sink state:

$$\texttt{is\_halted}(\sigma) \triangleq \forall n, r, \mathcal{C}_\sigma^n(llr)(r) = \sigma(r)$$

where $\mathcal{C}_\sigma^n(llr)$ performs n iterations of the $\mathcal{C}_\sigma$ function, i.e. computes the state of each register after n cycles. In fact, it suffices to demonstrate the following property to obtain a proof of `is_halted` for an environment.

$$\texttt{nochange}(\sigma) \triangleq \forall r, \mathcal{C}_\sigma(llr)(r) = \sigma(r)$$
$$\texttt{nochange\_halted}(\sigma) : \forall \sigma, \texttt{nochange}(\sigma) \Rightarrow \texttt{is\_halted}(\sigma)$$

We add a register called `halt` to our processor, and we equip each of the rules with a guard checking the value of this register. When it is true, no rules are run. We can prove that it behaves as expected:

$$\texttt{halt\_1\_implies\_halted} :$$
$$\forall \sigma, \sigma(\texttt{halt}) = 1 \Rightarrow \texttt{is\_halted}(\sigma)$$

Applying `nochange_halted` as well as transformation passes `ExploitReg`, `Simplify` and `Prune` gets us most of the way for this proof. We then need to show that the value of any individual register is left unchanged during the next cycle. At this stage, the demonstration of this property is trivial.

For simulation and synthesis, we emit an external call bound to a Verilog module which actually halts the execution of the processor whenever we set the value of `halt` to 1.

E. Formally verified properties

We are interested in proving four properties. In plain English, they may be worded as:

- "any overflow in the shadow stack leads to the immediate halting of the processor";
- "any underflow in the shadow stack leads to the immediate halting of the processor";
- "when returning from a procedure, if the stack and the shadow stack disagree on the return address, then the processor halts immediately";
- "in all other situations, the behavior of the processor remains unchanged".

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

$$\texttt{sstack\_empty}(\sigma) \triangleq \sigma(\texttt{sstack.sz}) == 0$$
$$\texttt{sstack\_full}(\sigma) \triangleq \sigma(\texttt{sstack.sz}) == \texttt{sstack.capacity}$$
$$\texttt{sstack\_top}(\sigma) \triangleq$$
$$\begin{cases} \emptyset & \text{if } \sigma(\texttt{sstack.sz}) \text{ is } 0 \\ \sigma(\texttt{sstack.stack}[\sigma(\texttt{sstack.sz})]) & \text{otherwise} \end{cases}$$

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.

Predicates `sstack_push` and `sstack_pop` express the conditions under which a push or a pop takes place. Their definitions (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 `candidate_return_address` gives the address that the current instruction attempts to return to, assuming it is a procedure return.

We give formal definitions for the first three properties we mentioned earlier:

$$\texttt{sstack\_uflow}(\sigma) \triangleq$$
$$\texttt{no\_mispred}(\sigma) \wedge \texttt{sstack\_empty}(\sigma) \wedge \texttt{sstack\_pop}(\sigma)$$
$$\texttt{sstack\_oflow}(\sigma) \triangleq$$
$$\texttt{no\_mispred}(\sigma) \wedge \texttt{sstack\_full}(\sigma)$$
$$\wedge \neg\texttt{sstack\_pop}(\sigma) \wedge \texttt{sstack\_push}(\sigma)$$

$$\texttt{sstack\_violation}(\sigma) \triangleq$$
$$\texttt{no\_mispred}(\sigma) \wedge \texttt{sstack\_pop}(\sigma) \wedge$$
$$\texttt{candidate\_return\_addr}(\sigma) \neq \texttt{sstack\_top}(\sigma)$$

We show that these three ways of violating the shadow stack policy result in the halting of the processor.

- `sstack_uflow_implies_halt`: $\forall \sigma,\ \texttt{sstack\_uflow}(\sigma) \Rightarrow \texttt{is\_halted}(\mathcal{C}_\sigma(llr))$
- `sstack_oflow_implies_halt`: $\forall \sigma,\ \texttt{sstack\_oflow}(\sigma) \Rightarrow \texttt{is\_halted}(\mathcal{C}_\sigma(llr))$
- `sstack_addr_violation_implies_halt`: $\forall \sigma,\ \texttt{sstack\_violation}(\sigma) \Rightarrow \texttt{is\_halted}(\mathcal{C}_\sigma(llr))$

The proofs of `sstack_uflow_implies_halt`, `sstack_oflow_implies_halt` and `sstack_addr_violation_implies_halt` share some similarities. In all of them, we start by applying `halt_1_implies_halted`. We then have to prove that the final value of `halt` is 1. A logical first step is to apply the `Prune` transformation pass: we don't care about the variables that are not relevant to the final value of `halt`. We can also exploit our hypotheses. For instance, for `sstack_underflow_implies_halt`, we know, among other things, that the shadow stack is empty and that the instruction in the execute stage of the pipeline corresponds to a return instruction. We can exploit this information to simplify the model further. Some case analysis is required to fully exploit the information about the executed instruction. Indeed, as was shown in IV-C, a pop should occur in two situations:

- when `rd` is neither `x1` or `x5` and `rs1` is either `x1` or `x5`;
- when both `rd` and `rs1` are `x1` or `x5` but $\texttt{rd} \neq \texttt{rs1}$.

In all branches, the rest of the proof is trivial. The proofs of the two other properties follow a similar pattern.

There still remains a last property to demonstrate. In order to prove that our shadow stack does not interfere with the rest of the processor, we need to show that, starting from the same environment and after one cycle, in the absence of a shadow stack violation, the value of the registers that were not introduced for the shadow stack is the same in the vanilla model as in the modified one.

Formally, we write this as:

$$\begin{aligned}&\texttt{sstack\_no\_interferences}:\\ &\forall\ \sigma,\quad \sigma(\texttt{halt}) = 0 \Rightarrow \neg\texttt{sstack\_violation}(\sigma) \Rightarrow\\ &\qquad \neg\texttt{sstack\_uflow}(\sigma) \Rightarrow \neg\texttt{sstack\_oflow}(\sigma) \Rightarrow\\ &\qquad \forall\ r, r \neq \texttt{sstack}[\cdot] \Rightarrow\\ &\qquad \mathcal{C}_\sigma(llr\_basic) = \mathcal{C}_\sigma(llr\_sstack)\end{aligned}$$

Once again, we start by exploiting some known values through the `ExploitReg` transformation, and keep simplifying the model with `Simplify` and `Prune` (using the variant `PruneList` this time, with all the registers, except those that are related to the shadow stack).

The shadow stack is only ever accessed from the rule corresponding to the execute stage in our model. Furthermore, accesses to the shadow stack do not modify registers of the basic Kôika model except for `halt`. The only variables specific to the version equipped with a shadow stack that remain in the LLR after the call to `PruneList` are those that impact `halt`. However, it can be shown that `¬stack_violation` implies that the value written to `halt` is 0. In other words, the write in question does not update the value of `halt`. Therefore, some surgical applications of `ReplaceVar` allow us to remove precisely the parts which differ between our two models. Then, our two LLRs are equal, and the goal is trivially true.

F. Quantitative summary of the proof effort

At the time of this writing, the proof framework is composed of around 19k lines of Coq, with about half of it being related to the Kôika to LLR compiler or to its correctness proof, and most of the rest being related to transformation passes and the associated proofs of correctness.

The properties we described in this section took around 2k lines of Coq to prove. Complete verification of the proofs takes around 10 minutes and uses a substantial amount of RAM (around 16GB) on a 12th Gen Intel i5 processor running at 4.4GHz with 32GB of RAM.

V. Experimental evaluation

a) Simulation of Kôika with Cuttlesim: We verify the overall functional correctness of our modified RISC-V processor by running it on Cuttlesim, the C++ simulator provided by the Kôika project that directly interprets and simulates the Kôika language. We run a test suite that targets all the instructions in the RV32I subset separately. All the tests provided with the original Kôika still pass with our modified processor.

```
void bad(){
  puts("Bad!\n");
}
int f(char* s){
  char buf[16];
  strcpy(buf,s);
}
int main(int argc, char** argv){
  int attack_buf[6];
  attack_buf[5] = (intptr_t)&bad;
  f((char*)attack_buf);
}
```

Fig. 10: Vulnerable program that overwrites its return address

b) Experimental validation of the shadow stack: Even with the proof of Section III, it is valuable to test whether a trivial overwriting of a function's return address is indeed detected by our shadow stack. This cross-validates that the theorems we proved earlier indeed entail a security property.

Figure 10 shows a C program that exhibits a trivial buffer overflow. Buffer `buf` in function `f` is 16-byte long, and the `strcpy` function performs no bounds checking before copying the buffer `attack_buf`, which is 24-byte long. Because we know the memory layout of this program, we know where in `attack_buf` we should place the address we want to jump to so that it will overwrite the vulnerable function's return address. Our shadow stack module should detect this violation and halt our processor before we even jump to that new return address. We run this program with the Cuttlesim simulator, once with the shadow stack deactivated and once with the shadow stack activated. In the first case, the buffer overflow succeeds, and we observe that the program writes the string `"Bad!"` to the console. On the other hand, when run with the shadow stack activated, we observe that the execution exits abruptly. By inspecting the final state of the Kôika model manually, we see that the `halt` register is set, and the instruction that was being executed at that point is precisely the `ret` instruction in the `f` function.

c) Synthesis on the TinyFPGA BX board: We successfully ran the Verilog output of the Kôika compiler on our modified RISC-V processor through the Yosys synthesis suite. The shadow stack that we could fit on this board, in addition to the processor model, has a capacity of only seven return addresses and incurs an overhead of 5.9 % in the number of logical cells (LUTs) used on the FPGA. This limitation in size is only due to the limited number of LUTs and quantity of internal RAM of this inexpensive FPGA.

Shadow stack	Clock frequency (MHz)	Used logical cells (out of 7680)	Critical path (ns)
Without	22.07	7049 (91%)	45.4
With	20.49	7463 (97%)	48.8

VI. Related work

We now discuss related lines of work and how they differ from our results.

a) ISA-level security proofs: ISAs are the specification that CPUs must implement. Thus, they form the foundation for software-level security. As such, there have been multiple recent works focusing on proving security properties at the ISA level. One such prominent line of work aims to prove various security properties for capability-enabled ISAs. Nienhuis et al. [24] prove capability monotonicity for CHERI-MIPS while Bauereiss et al. [4] prove it for the Arm Morello prototype. The work of Georges et al. [11], [12] and Skorstengaard et al. [28], [29] prove a variety of stack safety properties that can be enforced on capability machines. Similarly, Van Strydonck et al. [30] develop a library of verified wrappers around drivers leveraging capabilities for enforcing security properties. While these works also define and prove security properties about hardware, all their reasoning is done at the ISA level and must thus assume that the ISA is correctly implemented in hardware. In contrast, we reason at the register transfer level, which is much closer to the concrete hardware.

b) HDLs for security: There have been multiple HDLs proposed in the literature for securely designing circuits. For instance, Caisson [22] and SecVerilog [33] are both HDLs that use information-flow types to ensure that generated circuits are secure. Similarly, Iodine [13] and Xenon [14] use SMT-solvers to check that cryptographic circuits execute in constant time. While these approaches allow to prove *specific* security properties, they cannot be used to prove more general properties related to the execution of a CPU like our work does. Furthermore, using a proof assistant like we do has a lighter TCB footprint.

c) Formal languages compiled to Verilog: As mentioned earlier, this work uses the Kôika [5] language and its formalization to implement, specify and verify a shadow stack. Kami [7] is another language with formal semantics that can be compiled down to Verilog. Kami was developed by the same team who published Kôika and follows the same rule-based approach. However, the project is not actively maintained since Kôika offers a more precise cycle-accurate approach. Vericert [16] is a formally verified high-level synthesis tool based on CompCert that transforms C code into Verilog, but lacks support for implementing pipelines, which is crucial for implementing efficient CPUs. Similarly, HOL4 [23] has also been used for the verified synthesis of Verilog code. Specifically, it has been used for designing an in-order CPU implementing a custom instruction set. Unlike our work, theirs is not concerned with security mechanisms, and it is unknown whether they could actually implement an efficient and pipelined CPU.

d) Micro-architectural security proofs: Erbsen et al. [10] describe the implementation of a certified IoT lightbulb in Kami. This work also blends formal verification of 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. However, this work does not focus on security properties and relies on Kami, which does not have cycle-accurate semantics.

Knox [3] is a framework for building high assurance hardware security modules (HSM). It can be used to prove that a Verilog implementation correctly refines a functional specification defined as a state machine using an SMT solver. In contrast to our work, theirs is specific to HSM and cannot be used to prove general-purpose security properties.

VII. Limitations and future work

A. Limitations

1) Sensitivity of the LLR: Some transformation passes such as ReplaceVar (see III-D) take ids of LLR variable as argument. Proofs using these passes are very sensitive to changes to the model and tend to break even when changes occur in unrelated parts of the model. A way around this problem would be to allow referring to variables through a more persistent naming scheme. We have made some preliminary experiments for improving the maintainability of our proofs by automatically generating labels to the variables based on their roles.

2) Kôika and memory: Kôika does not offer a way of using the Block RAM of FPGAs. All the registers of a Kôika model are stored in the usually much smaller LUTs (lookup tables). It is always possible to build an interface to BRAM using external calls, as was done for the data and instruction memory in the RISC-V example, but this makes reasoning about memory accesses harder.

3) Partial interpretation in Coq: We ran into issues with Coq's partial interpretation mechanism while working on the proof of the shadow stack, as we mentioned in Section II-B. The Coq tactics such as `cbv`, `cbn`, `vm_compute`, which perform computation on Coq terms, do not allow the users to have fine-grained control on the evaluation process. When the terms we try to compute are very large and contain variables rather than ground terms, Coq's evaluation engine becomes unusably slow. We see two possible remediations to this problem. The first is to find a way to implement a computation tactic in Coq that supports so-called partial interpretation, i.e. computing with variables. The second is to make our compilation to LLRs and transformations of LLRs more aggressive so that fewer variables appear in these representations, and fewer occasions to trap the evaluation tactics in costly computations.

4) Configuration of the security mechanism: In the current stage of our implementation, the shadow stack mechanism is hard coded in the hardware design. Such an approach means that parameters, such as the size of the shadows stack, cannot be adjusted at runtime. Moreover, there is only one shadow stack. These choices ease the verification of the mechanism and are consistent with the type of processor we use, which correspond to a microcontroller without privileged mode. Such a CPU often executes a single application in an embedded

device. An interesting extension would be to provide some configuration mechanisms to adapt the mechanism at runtime. Obviously, this configuration mechanism must not be accessible to untrusted application code. Thus, this requires considering a more complex CPU with privileged execution mode and OS support. This evolution also poses the challenge of formal reasoning on the interaction between hardware and some trusted code, i.e. OS kernel code.

5) Handling of security property violations: Currently, a violation of a security property results in the halting of the processor. This is rather abrupt, but it helped us validate our approach. A more adequate response would be to emit a hardware exception that an operating system could handle at its discretion.

B. Future work

1) Functional verification: There is an official formal version of the RISC-V specification based on the Sail language [2], which includes facilities to export definitions to Coq. Proving that our processor design conforms to this specification would be a logical next step.

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.

3) Other security mechanisms: Another research direction would be to consider more ambitious security mechanisms, such as a more complex version of shadow stacks or capabilities. We would also like to tackle security mechanisms requiring proper software configuration, e.g. the isolation mechanism provided by OS kernels or hypervisors leveraging hardware features. Such approaches require formalizing the interactions between software and hardware components [21].

4) Timing side-channels: Kôika's semantics is cycle accurate. This makes it possible to reason about some forms of timing side-channels. Targeting mechanisms that enforce more complex policies, such as Information Flow Tracking mechanisms, is more challenging. Indeed, such mechanisms are supposed to guarantee some forms of non-interference, which correspond to predicates on sets of traces, i.e., hyperproperties [9].

VIII. Conclusion

In this paper, we propose a methodology for building synthesizable hardware with formally verified security mechanisms. We base our work on the Kôika formal Hardware Description Language, which we modify in depth to make it practical for reasoning on hardware models. We implement a verified compiler from Kôika models to a lower-level, more explicit representation, which is more amenable to proving. In addition, we define a set of verified transformation passes on these low-level representations that can be applied to simplify objects in this representation, as needed for each proof.

We then apply our methodology to the implementation of a verified shadow stack for a simple pipelined RISC-V processor. We prove some security properties about our implementation. Notably, we show that detecting the overwrite of a return address results in the halting of the processor. This result is further confirmed by simulating the processor and running a simple example code performing a buffer overflow, which is indeed detected by our shadow stack.

While the security mechanism verified here is relatively simple, it forms the foundation for possible future work and exemplifies how more complex mechanisms could be tackled.

References

[1] de Amorim, A.A., Collins, N., DeHon, A., Demange, D., Hritcu, C., Pichardie, D., Pierce, B.C., Pollack, R., Tolmach, A.: A verified information-flow architecture. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 165–178. ACM (2014). https://doi.org/10.1145/2535838.2535839, https://doi.org/10.1145/2535838.2535839

[2] Armstrong, A., Bauereiss, T., Campbell, B., Reid, A., Gray, K.E., Norton, R.M., Mundkur, P., Wassell, M., French, J., Pulte, C., Flur, S., Stark, I., Krishnaswami, N., Sewell, P.: ISA Semantics for ARMv8-a, RISC-V, and CHERI-MIPS. Proc. ACM Program. Lang. **3**(POPL), 71:1–71:31 (2019). https://doi.org/10.1145/3290384, https://doi.org/10.1145/3290384

[3] Athalye, A., Kaashoek, M.F., Zeldovich, N.: Verifying hardware security modules with information-preserving refinement. In: Aguilera, M.K., Weatherspoon, H. (eds.) 16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022. pp. 503–519. USENIX Association (2022), https://www.usenix.org/conference/osdi22/presentation/athalye

[4] Bauereiss, T., Campbell, B., Sewell, T., Armstrong, A., Esswood, L., Stark, I., Barnes, G., Watson, R.N.M., Sewell, P.: Verified security for the Morello capability-enhanced prototype Arm architecture. In: Sergey, I. (ed.) Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings. Lecture Notes in Computer Science, vol. 13240, pp. 174–203. Springer (2022). https://doi.org/10.1007/978-3-030-99336-8_7, https://doi.org/10.1007/978-3-030-99336-8_7

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

[6] Busi, M., Noorman, J., Bulck, J.V., Galletta, L., Degano, P., Mühlberg, J.T., Piessens, F.: Provably secure isolation for interruptible enclaved execution on small microprocessors. In: 2020 IEEE 33rd Computer Security Foundations Symposium (CSF). pp. 262–276 (2020). https://doi.org/10.1109/CSF49147.2020.00026

[7] Choi, J., Vijayaraghavan, M., Sherman, B., Chlipala, A., Arvind: Kami: a Platform for High-Level Parametric Hardware Specification and its Modular Verification. Proc. ACM Program. Lang. **1**(ICFP), 24:1–24:30 (2017), https://doi.org/10.1145/3110268

[8] Chong, S., Guttman, J., Datta, A., Myers, A., Pierce, B., Schaumont, P., Sherwood, T., Zeldovich, N.: Report on the NSF workshop on formal methods for security. Available at http://dl.acm.org/citation.cfm?id=3040225. (Aug 2016)

[9] Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security **18**(6), 1157–1210 (2010)

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

[11] Georges, A.L., Guéneau, A., Strydonck, T.V., Timany, A., Trieu, A., Huyghebaert, S., Devriese, D., Birkedal, L.: Efficient and provable local capability revocation using uninitialized capabilities. Proc. ACM Program. Lang. **5**(POPL), 1–30 (2021). https://doi.org/10.1145/3434287, https://doi.org/10.1145/3434287

[12] Georges, A.L., Trieu, A., Birkedal, L.: Le temps des cerises: efficient temporal stack safety on capability machines using directed capabilities. Proc. ACM Program. Lang. **6**(OOPSLA1), 1–30 (2022). https://doi.org/10.1145/3527318, https://doi.org/10.1145/3527318

[13] von Gleissenthall, K., Kici, R.G., Stefan, D., Jhala, R.: IODINE: verifying constant-time execution of hardware. In: Heninger, N., Traynor, P. (eds.) 28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019. pp. 1411–1428. USENIX Association (2019), https://www.usenix.org/conference/usenixsecurity19/presentation/von-gleissenthall

[14] von Gleissenthall, K., Kici, R.G., Stefan, D., Jhala, R.: Solver-aided constant-time hardware verification. In: Kim, Y., Kim, J., Vigna, G., Shi, E. (eds.) CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021. pp. 429–444. ACM (2021). https://doi.org/10.1145/3460120.3484810, https://doi.org/10.1145/3460120.3484810

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

[16] Herklotz, Y., Pollard, J.D., Ramanathan, N., Wickerson, J.: Formal verification of high-level synthesis. Proc. ACM Program. Lang. **5**(OOPSLA), 1–30 (2021). https://doi.org/10.1145/3485494, https://doi.org/10.1145/3485494

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

[18] Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. TOCS **32**(1) (2014)

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

[20] Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM **52**(7), 107–115 (2009), http://xavierleroy.org/publi/compcert-CACM.pdf

[21] Letan, T., Chifflier, P., Hiet, G., Néron, P., Morin, B.: SpecCert: Specifying and verifying hardware-based security enforcement. In: Fitzgerald, J.S., Heitmeyer, C.L., Gnesi, S., Philippou, A. (eds.) FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9995, pp. 496–512 (2016). https://doi.org/10.1007/978-3-319-48989-6_30, https://doi.org/10.1007/978-3-319-48989-6_30

[22] Li, X., Tiwari, M., Oberg, J., Kashyap, V., Chong, F.T., Sherwood, T., Hardekopf, B.: Caisson: a hardware description language for secure information flow. In: Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 109–120. ACM (2011). https://doi.org/10.1145/1993498.1993512, https://doi.org/10.1145/1993498.1993512

[23] Lööw, A., Myreen, M.O.: A proof-producing translator for verilog development in HOL. In: Gnesi, S., Plat, N., Day, N.A., Rossi, M. (eds.) Proceedings of the 7th International Workshop on Formal Methods in Software Engineering, FormaliSE@ICSE 2019, Montreal, QC, Canada, May 27, 2019. pp. 99–108. IEEE / ACM (2019). https://doi.org/10.1109/FormaliSE.2019.00020, https://doi.org/10.1109/FormaliSE.2019.00020

[24] Nienhuis, K., Joannou, A., Bauereiss, T., Fox, A.C.J., Roe, M., Campbell, B., Naylor, M., Norton, R.M., Moore, S.W., Neumann, P.G., Stark, I., Watson, R.N.M., Sewell, P.: 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. pp. 1003–1020. IEEE (2020). https://doi.org/10.1109/SP40000.2020.00055, https://doi.org/10.1109/SP40000.2020.00055

[25] Nikhil, R.: 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. pp. 69–70 (2004). https://doi.org/10.1109/MEMCOD.2004.1459818

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

[27] Shanbhogue, V., Gupta, D., Sahita, R.: 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. pp. 8:1–8:11. ACM (2019). https://doi.org/10.1145/3337167.3337175, https://doi.org/10.1145/3337167.3337175

[28] Skorstengaard, L., Devriese, D., Birkedal, L.: Reasoning about a machine with local capabilities: Provably safe stack and return pointer management. ACM Trans. Program. Lang. Syst. **42**(1), 5:1–5:53 (2020). https://doi.org/10.1145/3363519, https://doi.org/10.1145/3363519

[29] Skorstengaard, L., Devriese, D., Birkedal, L.: Stktokens: Enforcing well-bracketed control flow and stack encapsulation using linear capabilities. J. Funct. Program. **31**, e9 (2021). https://doi.org/10.1017/S095679682100006X, https://doi.org/10.1017/S095679682100006X

[30] Strydonck, T.V., Georges, A.L., Guéneau, A., Trieu, A., Timany, A., Piessens, F., Birkedal, L., Devriese, D.: Proving full-system security properties under multiple attacker models on capability machines. In: 35th IEEE Computer Security Foundations Symposium, CSF 2022, Haifa, Israel, August 7-10, 2022. pp. 80–95. IEEE (2022). https://doi.org/10.1109/CSF54842.2022.9919645, https://doi.org/10.1109/CSF54842.2022.9919645

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

[32] Woodruff, J., Watson, R.N.M., Chisnall, D., Moore, S.W., Anderson, J., Davis, B., Laurie, B., Neumann, P.G., Norton, R.M., Roe, M.: The CHERI capability model: Revisiting RISC in an age of risk. In: ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014. pp. 457–468. IEEE Computer Society (2014). https://doi.org/10.1109/ISCA.2014.6853201, https://doi.org/10.1109/ISCA.2014.6853201

[33] Zhang, D., Wang, Y., Suh, G.E., Myers, A.C.: A hardware design language for timing-sensitive information-flow security. In: Özturk, Ö., Ebcioglu, K., Dwarkadas, S. (eds.) Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015. pp. 503–516. ACM (2015), https://doi.org/10.1145/2694344.2694372