Formalizing Hardware Security Mechanisms, Using SMT Solvers

Work in progress

Pierre Wilke, Matthieu Baty, Guillaume Hiet, Alix Trieu

SUSHI, CentraleSupélec Rennes, Inria, ANSSI

April 3rd, 2024

Overview

Goal: implement and prove hardware security mechanisms, at the micro-architectural level.

Earlier work[1]:

- extend an existing RISC-V processor written in the Kôika Hardware Description Language with a shadow stack
- propose a framework for proving properties about Kôika designs
- long and fragile Coq proofs of shadow stack correctness

This talk:

- Let's use a SMT solver to do the long and boring proof for us.

[1] Matthieu Baty et al. "A Generic Framework to Develop and Verify Security Mechanisms at the Microarchitectural Level: Application to Control-Flow Integrity". In: *CSF 2023*. IEEE, 2023, pp. 372–387.

Outline

1. Our previous work: proofs on Kôika designs

2. Proofs using a SMT solver

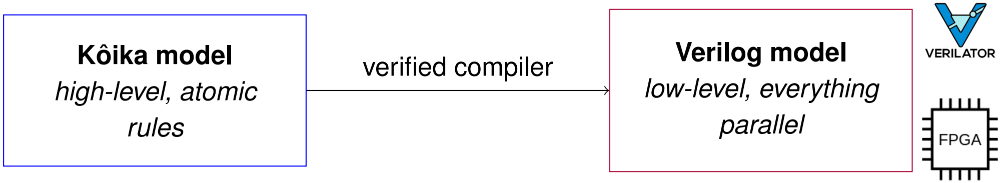
Outline

1 Our previous work: proofs on Kôika designs

2 Proofs using a SMT solver

Kôika[2]

A **Hardware Description Language** embedded in Coq.



Rules describe how the registers are updated **at each cycle.**

Conflicts occur e.g. when the same register is updated by two different rules.
$\rightsquigarrow$ **Complex semantics**

[2] *The Essence of BlueSpec*, PLDI'20, Thomas Bourgeat *et al.*, `https://github.com/mit-plv/koika`

A RISC-V processor in Kôika

Kôika developers provide an example model of a RISC-V processor

- 4-stage processor (Fetch, Decode, Execute, Writeback)
- RV32I
- unprivileged specification, no interrupts
- under 1000 lines of Kôika code
- runs on an actual FPGA board (TinyFPGA, LambdaConcept ECPIX-5)

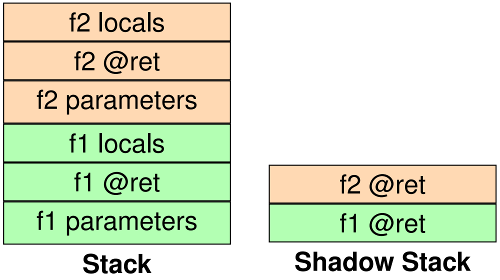
We implemented and proved a **hardware shadow stack**.

Shadow stacks

- Protection against buffer overflows that overwrite the return address
- Enforces (part of) control-flow integrity (only backward edges)
  - i.e., when we execute a `ret` instruction, we always jump back to (just after) our call site

Principle:

- when a `call` instruction is encountered, push `next(pc)` on the shadow stack
- when a `ret` instruction is encountered, pop `addr_ss` from the shadow stack and pop `addr` from the normal stack
  - If `addr_ss == addr`, continue
  - Else, we detect a violation



CentraleSupélec

Shadow stack

Implementation:

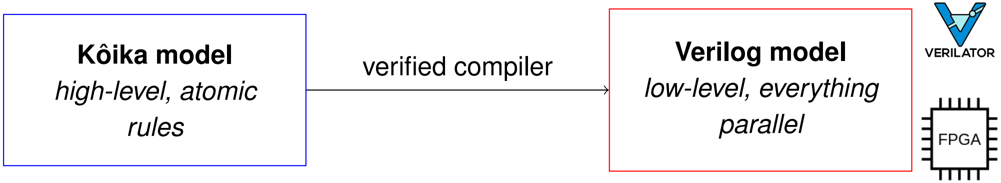
- new memory region for our shadow stack
- instrument the `Execute` stage to push onto and pop from the shadow stack when needed
- when a violation is detected, we **halt** the processor

What we want to prove

- Return to a **modified return address** $\Rightarrow$ halt processor
  - A bit more precisely :
    If the instruction about to be executed in the pipeline is a `ret`[3],
    and the address stored at the top of the shadow stack is different from the address to which we are about to jump,
    then the processor should be put in a *halting state*.
- **Underflow or overflow** of the shadow stack $\Rightarrow$ halt processor
- Otherwise, behaviour **preserved**

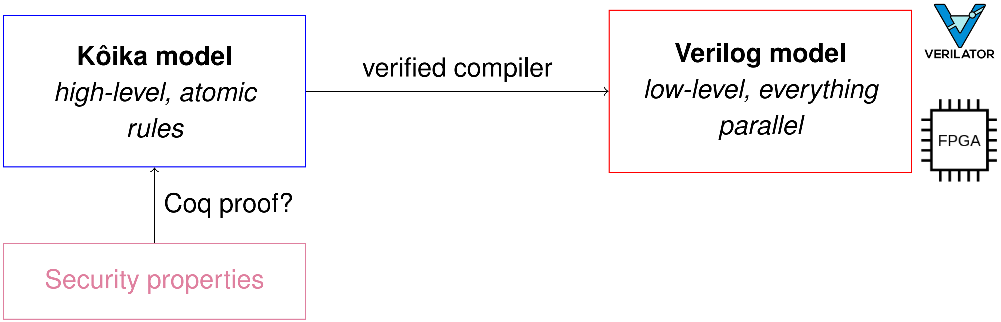
[3]In RISC-V, `ret` is actually `jr ra`, i.e. jump to address contained in register `ra`.

Proving properties on Kôika models

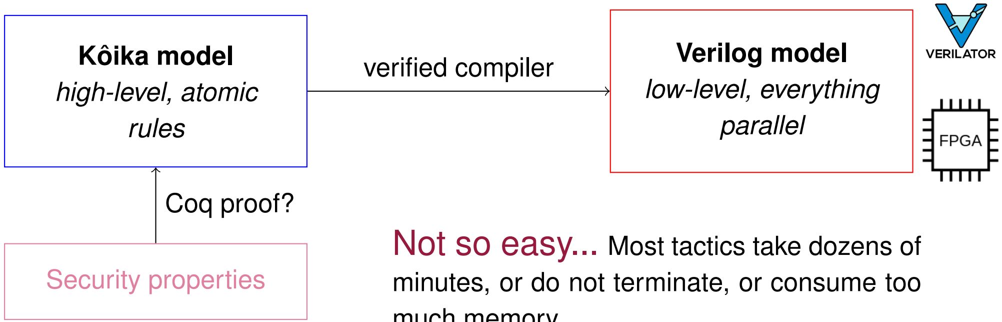


CentraleSupélec

Proving properties on Kôika models



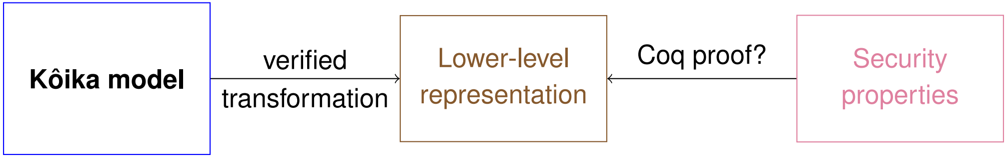
Proving properties on Kôika models



Not so easy... Most tactics take dozens of minutes, or do not terminate, or consume too much memory.

Proofs on Kôika models using Low-Level Representations

Our solution: we compile high-level Kôika models into **lower-level representations** (LLR), more amenable to proofs.



$$e ::= v \mid cst \mid \texttt{reg}(r) \mid \triangleright e \mid e_1 \bowtie e_2 \mid \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3$$
$$llr ::= \{vars : V \to e \; ; \; final\_values : \texttt{Reg} \to V\}$$

The **LLR** is a map $\texttt{Reg} \to \texttt{Expr}$, which gives the value of each register at the end of a cycle, depending on the values of registers at the beginning of the cycle.
In particular, the **conflict detection** logic is embedded into these expressions.

Lower-level representation (LLR)

Computing a LLR is quick; but produces a large number of quite deep expressions.

We developed a range of **program transformations** akin to compiler optimizations on LLRs :

- constant folding ($3 + 4 \rightsquigarrow 7$)
- replace variable v with constant c (with a manual proof obligation that $[\![v]\!] \rightsquigarrow c$)
- replace sub-expression e with e' (with a manual proof obligation that $e \equiv e'$)
- replace register r with its value at the beginning of the cycle (with a manual proof obligation)
- exploit partial information about register values (e.g. bits `6:0` of register `inst` are `0001101`) (with a manual proof obligation)

It's up to the (human) prover to apply each program transformation manually and prove the obligations.

Correctness proofs of the shadow stack

Proof	No. lines of proof	`Time Qed.`
Underflow $\Rightarrow$ halt	150	1m10s
Overflow $\Rightarrow$ halt	270	2m30s
Wrong address $\Rightarrow$ halt	900	3m10s

Lots of lines of boring proofs, very fragile (numbering of variables).

Our proofs are mainly case studies about bitvectors. What if we discharged our proofs to SMT solvers?

Outline

1 Our previous work: proofs on Kôika designs

2 Proofs using a SMT solver

CentraleSupélec

Example proof: shadow stack overflow implies halt

```
Definition sstack_full ctx : Prop :=
  ctx (ShadowStack.size) = ShadowStack.capacity.

Definition sstack_push ctx : Prop :=
  forall s b,
   ctx (d2e.data) = Struct s ->
   get_field_struct s "inst" = Some (Bits b) ->
   is_call_instruction b = true.

Lemma overflow_halt: forall ctx,
  sstack_full ctx -> sstack_push ctx ->
  (cycle ctx) halt = Bits [true].
```

Example proof: shadow stack overflow implies halt

```
Lemma overflow_halt: forall ctx,
  sstack_full ctx -> sstack_push ctx ->
  (cycle ctx) halt = Bits [true].
```

Instead of reasoning about the Kôika semantics of a clock cycle, we compute the LLR corresponding to the circuit. Our goal becomes:

```
Lemma overflow_halt: forall ctx,
  sstack_full ctx -> sstack_push ctx ->
  llr.final_values halt = v_halt ->
  ⟦v_halt⟧^ctx_llr = Bits [true].
```

We then encode the hypotheses about the current context (`sstack_full`, `sstack_push`) as LLR expressions.

We just need to convert LLR expressions into SMTLIB expressions about bitvectors!

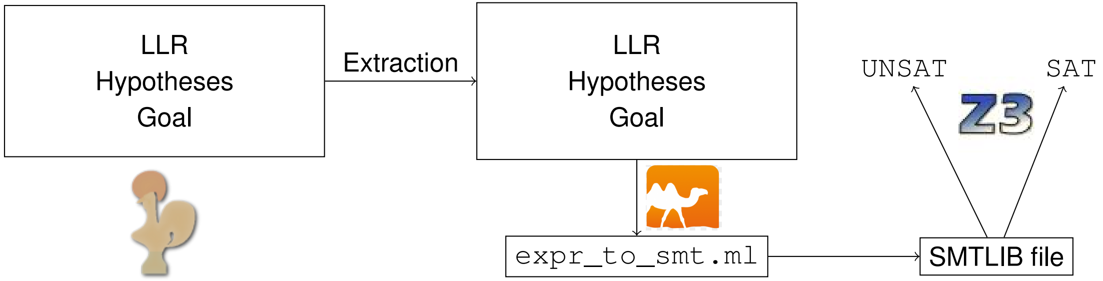
CentraleSupélec

SMT encoding

```
; all variables in LLR
(assert (= v_1 (encode_expr e_1)))
...
(assert (= v_n (encode_expr e_n)))

; hypotheses
; sstack_full
(assert (= reg_shadow_stack_size capacity))
; about to push
(assert ...)
; assert negation of goal
(assert (not (= #b1 final_reg_halt)))
(check-sat) ; expect unsat
(get-model)
```

Proofs



Proof	No. lines of proof	Time Qed.	Time z3 (s)
Underflow ⇒ halt	150	1m10s	0.08
Overflow ⇒ halt	270	2m30s	0.07
Wrong address ⇒ halt	900	3m10s	6.14
No problem			1.70
Addition correct			0.07

CentraleSupélec

Better integration, with SMTCoq ?

OK but we're leaving the Coq world... Can we keep all the formal guarantees of Coq **and** the automation provided by SMT solvers?

SMTCoq[4] sounds like a possible solution

- transforms current goal into a SMT formula
- if `unsat`: solver generates an unsat core, translated back into a Coq proof
- if `sat`: solver generates an (counter-)example model, given back to user

This is still work-in-progress...

[4] `https://github.com/smtcoq/smtcoq`

CentraleSupélec

Conclusion

We automated the proof methodology for Kôika designs using SMT solvers

- makes proof maintenance much easier
- enables exploration of larger, more complex designs (WIP privilege levels, interrupts...)
- TODO: integration with SMTCoq
- TODO: functional correctness wrt. an ISA specification

Hiring **PhDs** and **post-docs** in CentraleSupélec, Rennes!
SUSHI Inria team - SecUrity at the Software Hardware Interface

Topics: formal models of processors, binary analysis,

Contact:

✉ `pierre.wilke@centralesupelec.fr`

✉ `guillaume.hiet@centralesupelec.fr`