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 Kôïka Hardware Description Language with a shadow stack
- propose a framework for proving properties about Kôïka 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.

---

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
A Hardware Description Language embedded in Coq.

Kôika model
high-level, atomic
rules

Verilog model
low-level, everything parallel

Rules describe how the registers are updated at each cycle.

Conflicts occur e.g. when the same register is updated by two different rules.

Complex semantics

2 The Essence of BlueSpec, PLDI’20, Thomas Bourgeat et al., https://github.com/mit-plv/koika
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

```
Stack
f1 parameters  f2 parameters
f1 @ret        f2 @ret
f1 locals      f2 locals

Shadow Stack
f1 @ret        f2 @ret
```
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** ⇒ 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 ⇒ halt processor
- Otherwise, behaviour **preserved**

^3In RISC-V, `ret` is actually `jr ra`, i.e. jump to address contained in register `ra`. 
Proving properties on Kôika models

Kôika model

*high-level, atomic rules*

Verilog model

*low-level, everything parallel*

verified compiler

Our previous work: proofs on Kôika designs

April 3rd, 2024
Proving properties on Kôika models

Kôika model
high-level, atomic rules

verified compiler

Verilog model
low-level, everything parallel

Coq proof?

Security properties
Proving properties on Kôika models

Kôika model
high-level, atomic rules

Verilog model
low-level, everything parallel

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.

\[ e ::= v \mid \text{cst} \mid \text{reg}(r) \mid e_1 \Downarrow e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \]

\[ llr ::= \{ \text{vars} : V \to e ; \text{final_values} : \text{Reg} \to V \} \]

The LLR is a map \( \text{Reg} \to \text{Expr} \), 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 \((3 + 4 \rightarrow 7)\)
• replace variable \(v\) with constant \(c\) (with a manual proof obligation that \(\llbracket v \rrbracket \rightarrow c\))
• 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 \(\text{inst}\) 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

<table>
<thead>
<tr>
<th>Proof</th>
<th>No. lines of proof</th>
<th>Time Qed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Underflow ⇒ halt</td>
<td>150</td>
<td>1m10s</td>
</tr>
<tr>
<td>Overflow ⇒ halt</td>
<td>270</td>
<td>2m30s</td>
</tr>
<tr>
<td>Wrong address ⇒ halt</td>
<td>900</td>
<td>3m10s</td>
</tr>
</tbody>
</table>

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
Example proof: shadow stack overflow implies halt

**Definition** \( sstack\_full \) \( \text{ctx} : \text{Prop} \) :=
\[ \text{ctx} (\text{ShadowStack}.\text{size}) = \text{ShadowStack}.\text{capacity} . \]

**Definition** \( sstack\_push \) \( \text{ctx} : \text{Prop} \) :=
\[
\text{forall } s \ b , \ \\
\text{ctx} (d2e.\text{data}) = \text{Struct } s \rightarrow \\
\text{get\_field\_struct } s \ "\text{inst}" = \text{Some } (\text{Bits } b) \rightarrow \\
\text{is\_call\_instruction } b = \text{true} . \]

**Lemma** \( \text{overflow\_halt} : \text{forall } \text{ctx}, \ \\
\text{sstack\_full } \text{ctx} \rightarrow \text{sstack\_push } \text{ctx} \rightarrow \\
(\text{cycle } \text{ctx}) \text{ halt } = \text{Bits } [\text{true}] . \)
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!
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

**Proofs**

### LLR Hypotheses

- **Goal**

### Extraction

- **LLR Hypotheses**

### SMTLIB file

- **expr_to_smt.ml**

### Z3

- **UNSAT**
- **SAT**

### Formalizing Hardware Security Mechanisms, Using SMT Solvers

<table>
<thead>
<tr>
<th>Proof</th>
<th>No. lines of proof</th>
<th>Time Qed.</th>
<th>Time z3 (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Underflow ⇒ halt</td>
<td>150</td>
<td>1m10s</td>
<td>0.08</td>
</tr>
<tr>
<td>Overflow ⇒ halt</td>
<td>270</td>
<td>2m30s</td>
<td>0.07</td>
</tr>
<tr>
<td>Wrong address ⇒ halt</td>
<td>900</td>
<td>3m10s</td>
<td>6.14</td>
</tr>
<tr>
<td>No problem</td>
<td></td>
<td></td>
<td>1.70</td>
</tr>
<tr>
<td>Addition correct</td>
<td></td>
<td></td>
<td>0.07</td>
</tr>
</tbody>
</table>
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 \texttt{unsat}: solver generates an unsat core, translated back into a Coq proof
• if \texttt{sat}: solver generates an (counter-)example model, given back to user

This is still work-in-progress...

\(^4\)https://github.com/smtcoq/smtcoq
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 SMTCooq
- 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