CentraleSupélec

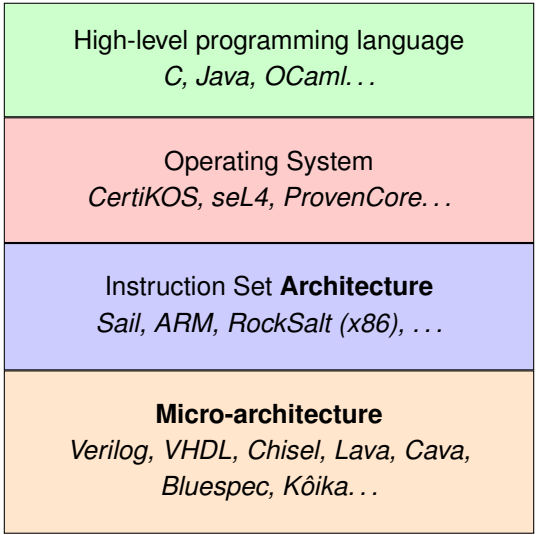
Formalizing Hardware Security Mechanisms

Pierre Wilke, Matthieu Baty, Guillaume Hiet, Arnaud Fontaine, Alix Trieu

CIDRE, CentraleSupélec Rennes, Inria, ANSSI

March 29th, 2023

Stack of abstractions



Goal : implement and prove hardware security mechanisms

Examples :

- shadow stack
- memory protection
- privilege levels
- ...

Work mainly done by Matthieu Baty, Ph.D. student in CIDRE

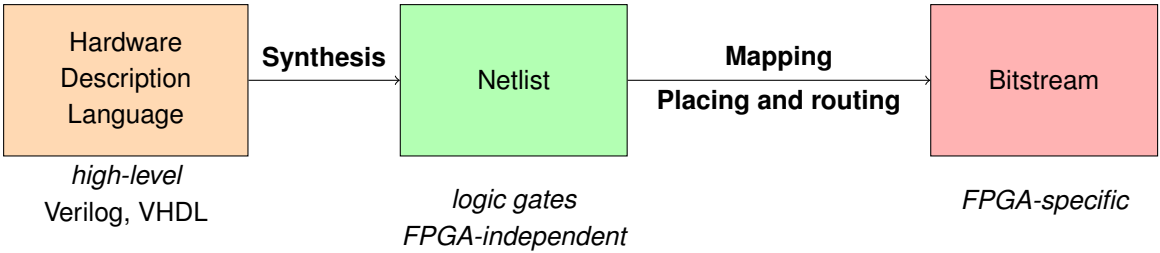
CentraleSupélec

Outline

CentraleSupélec

Outline

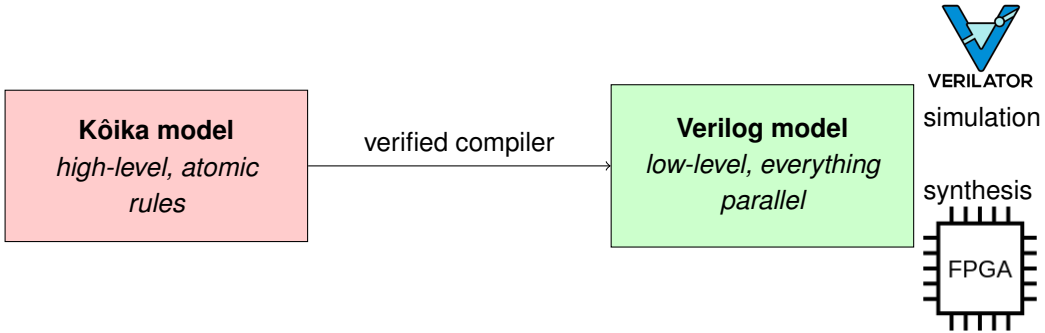
Hardware Design Workflow



CentraleSupélec

Kôika [1]

A Hardware Description Language embedded in Coq.



`https://github.com/mit-plv/koika`

Based on Bluespec (MEMOCODE'04, notion of **atomic rules**, compiles into Verilog)

1. *The Essence of BlueSpec*, PLDI'20, Thomas Bourgeat *et al.*

Kôika syntax and semantics

$$
\begin{array}{lrcl}
\text{Actions} & a & ::= & \vec{b} \mid x \mid \texttt{skip} \\
 & & \mid & \texttt{read}\ r \mid \texttt{write}\ r\ a \\
 & & \mid & \texttt{let}\ x = a\ \texttt{in}\ a \\
 & & \mid & \texttt{if}\ a\ \texttt{then}\ a\ \texttt{else}\ a \\
 & & \mid & f(a, \ldots, a) \mid \texttt{abort} \\
\text{Registers} & r & & \\
\text{Variables} & x & & \\
\\
\text{Program} & P & ::= & [\texttt{rule}\ name = a]* \\
 & & + & \texttt{schedule} = \overrightarrow{name}
\end{array}
$$

A *program* is a set of rules.
Rules manipulate registers.

One-rule-at-a-time (ORAAT) semantics : at each cycle, one rule is picked non-deterministically and executed.

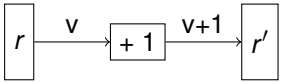
But, in the generated circuit, all rules run **in parallel**.
The compiler introduces **control logic** to :

- determinize the semantics, by following a user-provided **schedule** ;
- rule out parallel behaviors that would violate ORAAT semantics.

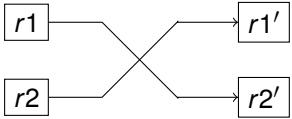
Kôika - examples

Register `read`s see the value of the registers at the beginning of the cycle.
Register `write`s are only committed at the end of the cycle (once all rules have run).

```
rule increment =
  let v = read r in
  write r (v + 1)
```



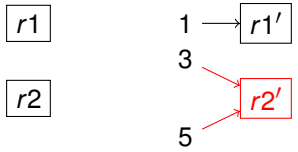

```
rule swap =
  write r1 (read r2);
  write r2 (read r1)
```



Kôika - examples : double write

A rule that writes twice to the same register leads to a conflict.

```
rule doublewrite =
  write r1 1;
  write r2 3;
  write r2 5
```



Conflict !

All effects of that rule are discarded.

In particular, the write to `r1` is also discarded.

Conflict detected at run-time (may depend on conditions).

- in Kôika semantics : keep track of all writes performed within this cycle, detect if double-write
- in the compiled circuit : add control logic that detects these cases

Kôika - examples with multiple rules

```
rule W =
  write r1 2
```

```
rule R =
  write r2 (read r1 + 1)
```

If our schedule is $[W; R]$:

- rule R will be discarded, i.e. will not fire in the same cycle as rule W
- because according to ORAAT, rule R should read the value written by rule W
- that's impossible because register writes are only made visible at the end of a cycle

within the same cycle, a read may not follow a write on the same register.

If our schedule is $[R; W]$:

- Running rules R and W *in parallel*, using the values at the beginning of the cycle for reads, is equivalent to running rule R, followed by rule W.
- Both rules will fire at each cycle.

CentraleSupélec

Kôika – conflict summary

In summary, within one cycle :

- there cannot be two writes on the same register. The rule performing the second write is **entirely discarded**.

  (could be the same rule)

- there cannot be a read on a register that has been written to by a *previous rule*. The rule performing the read is **entirely discarded**.

That's not entirely true : for performance, Kôika actually allows these situations where data written in registers may flow from one rule to the next within the same cycle, using *ports*. For simplicity, we ignore ports in that presentation.

Kôika semantics, formally

The semantics of a rule, i.e. an action a is given by :

$$[\![a]\!](\mathscr{R}, L) = \begin{cases} \texttt{Log}\ l & \text{if } a \text{ succeeds and produces a log } l \\ \texttt{Fail} & \text{otherwise} \end{cases}$$

A *log* is a list of read/write events on registers, e.g. [`write r1 2;read r2;write r2 3`]

- L is the log of events that occurred within the same cycle, in all previous rules.
- l is the log of events produced by **this rule**

The semantics of a schedule is given by $(L, sch) \Downarrow L'$:

$$\frac{[\![a]\!](\mathscr{R}, L) = \texttt{Log}\ l \qquad (L{+}{+}l, sch) \Downarrow L'}{(L, a :: sch) \Downarrow L'} \qquad \frac{[\![a]\!](\mathscr{R}, L) = \texttt{Fail} \qquad (L, sch) \Downarrow L'}{(L, a :: sch) \Downarrow L'}$$

Kôika semantics, formally, cont'd

The semantics of actions is given by :

$$\Gamma \vdash (l, a) \downarrow_{(L,\mathscr{R})} (l', v)$$

- Γ : environment for variables bound by `let ... in`
- l, l' : initial and final *action logs*
- L : previous rules' log
- $\mathscr{R}$: register values at the beginning of the cycle
- v : value computed by the action

$$\frac{(wr, r, *) \notin L}{\Gamma \vdash (l, \textbf{read}\ r) \downarrow (l ++ [(read, r)], \mathscr{R}(r))}$$

$$\frac{\Gamma \vdash (l, a) \downarrow (l', v) \qquad (*, r, *) \notin (L ++ l')}{\Gamma \vdash (l, \textbf{write}\ r\ a) \downarrow (l' ++ [(write, r, v)], tt)}$$

(Semantics of other types of actions are far less surprising for PL people)

Kôika semantics, summary

- Kôika programs are sets of rules, together with a scheduler
- Rules update **registers**
- All rules execute during each cycle, however :
  - each rule may or may not contribute to the next state of registers, depending on whether conflicts appear
- Conflicts appear when :
  - reading a register *r* after a write on that register has occurred in a previous rule
  - writing on register *r* after a read or write has occurred

- Quite hard to predict whether a conflict will happen, hence whether a rule will succeed or fail...

CentraleSupélec

Outline

CentraleSupélec

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

Back to what we wanted to do : **hardware security mechanisms**

- this RISC-V processor looks promising
- we can modify it and implement our security mechanism
- it seems that we have all we need to verify security properties

CentraleSupélec

Hardware security mechanisms

A hardware security mechanism is

- a hardware component (e.g. memory protection unit, shadow stack, privilege levels)
- that enforces a **security property** (confidentiality, integrity, availability)

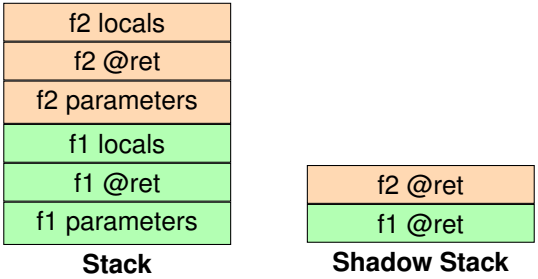
So far, we focused on implementing a **shadow stack** (à la Intel CET)

- protects 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

Shadow stacks

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



Shadow stack

Implementation :

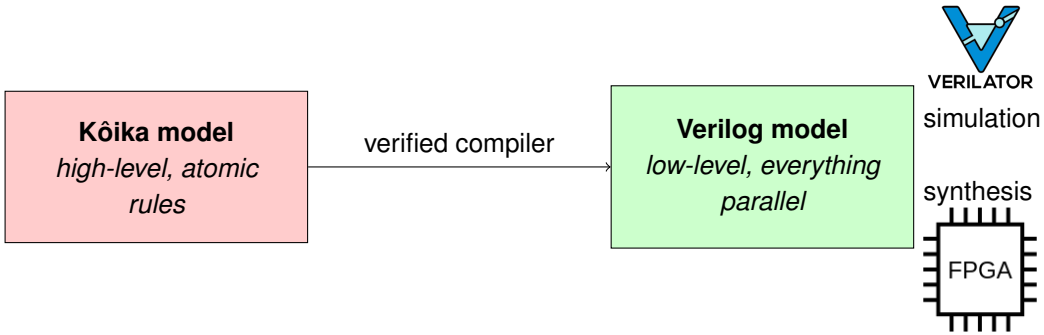
- new memory region for our shadow stack
- instrument the `Execute` stage to push onto and pop from the shadow stack when needed
- when a violation is detected, we **halt** the processor

What we want to prove

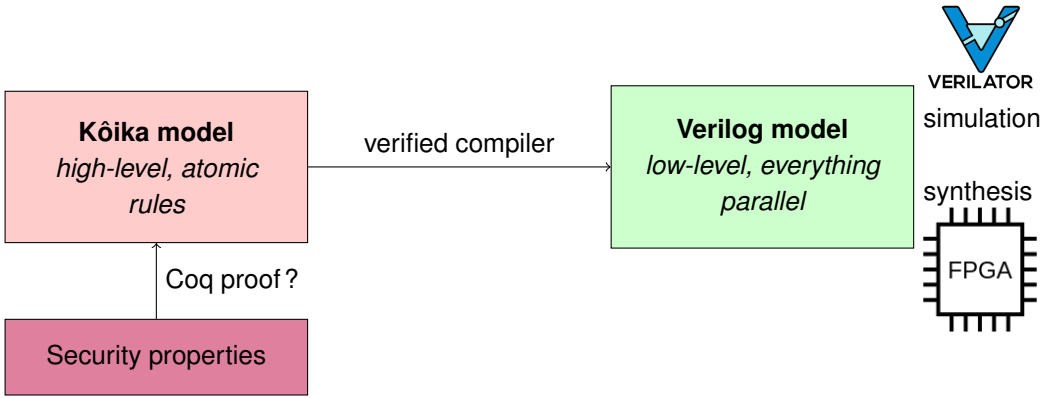
- Return to a **modified return address** $\Rightarrow$ halt processor
  - A bit more precisely :
    If the instruction about to be executed in the pipeline is a `ret` [2],
    and the address stored at the top of the shadow stack is different from the address to which we are about to jump,
    then the processor should be put in a *halting state*.
- **Underflow or overflow** of the shadow stack $\Rightarrow$ halt processor
- Otherwise, behaviour **preserved**

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

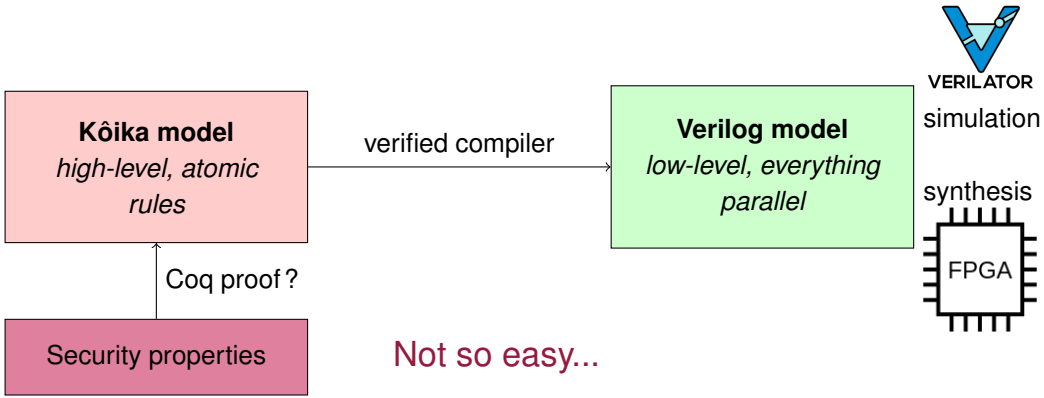
Proving properties on Kôika models



Proving properties on Kôika models



Proving properties on Kôika models



Not so easy...

Problems with proofs on Kôika models

Most tactics take minutes, hours, or do not terminate, or consume all my (32GB) RAM.
Not sure exactly why, probably a combination of :

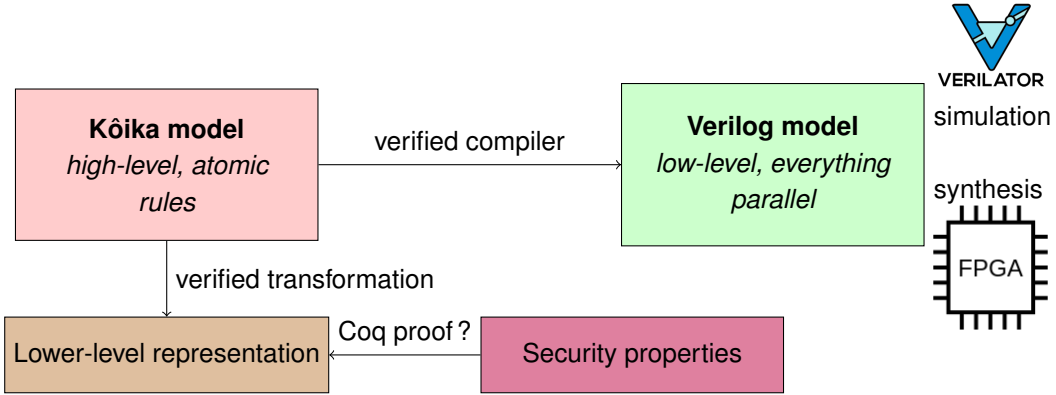
- heavy use of dependent types
- type class resolution
- the processor model is a quite large Coq term
- as we saw before, Kôika semantics are quite complex
- problem of partial evaluation

Attempts :

- write an inductive semantics instead of the existing executable semantics
  now `inversion` *is slow*
- write an alternative semantics with fewer dependent types *did not seem to help much*
- modular reasoning on smaller Kôika actions is not straightforward
  - the semantics is really about interactions between rules
  - the rules we write undergo typing and desugaring before we execute them

Proofs on Kôika models : a new approach

Kôika's semantics is very different from mainstream programming languages (especially with conflicts and rule cancellation).
Somewhat counter-intuitively, perhaps transforming our high-level Kôika model into a lower-level representation would facilitate our proofs.



CentraleSupélec

Outline

Lower-level representation

Our target is a representation of how register values are updated during a cycle, i.e. a mapping from each register in our model to an expression that describes its value at the end of a cycle.

$$
\begin{aligned}
e ::= \quad & v \quad \text{variable}(\in V) \\
\mid \; & c \quad \text{constant} \\
\mid \; & r \quad \text{register} \\
\mid \; & \textbf{if } e \textbf{ then } e \textbf{ else } e \\
\mid \; & f(\vec{e}) \\
\mathit{llr} ::= \quad & \{\mathit{vars} : V \to e \; ; \; \mathit{final\_values} : \texttt{reg\_t} \to V\}
\end{aligned}
$$

The compilation from a Kôika model to this lower-level representation encodes all conflict detection inside these expressions.
Should be easier to reason about.

CentraleSupélec

Lower-level representation (LLR)

Now the Coq interpretation of a cycle of the processor quickly produces a large set of variables.

Because all control logic (conflict detection, data forwarding) is explicit, the expressions associated to variables are quite deep and cannot be directly *computed* within Coq in reasonable time.

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 another sub-expression e' (with a manual proof obligation that $e \equiv e'$)
- replace register r with its value at the beginning of the cycle
- exploit partial information about register values (e.g. bits `6:0` of register `inst` are `0001101`)
- ...

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

Current state of our work

Proofs

- ✓ The halt state is a sink state
- ✓ Shadow stack **underflows** ⇒ halt
- ✓ Shadow stack **overflows** ⇒ halt
- ✓ Shadow stack **violation** ⇒ halt
- ✓ **No violation** ⇒ identical behavior

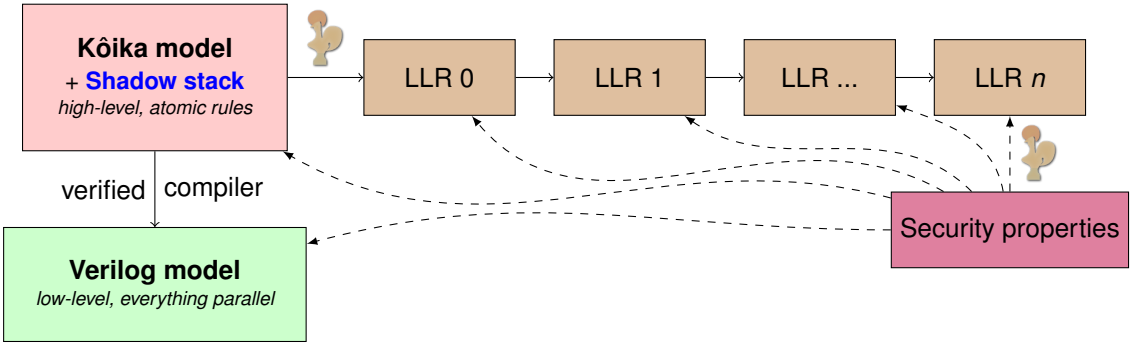
Simulation (verilator, cuttlesim) : we observe that the shadow stack works as expected on a few hand-written examples

Synthesis : the resulting processor runs on an actual FPGA board

Submitted to CSF'23

CentraleSupélec

Conclusion



Perspectives (us... and you!)

Next :

- other hardware security mechanism (e.g. memory protection, privilege levels)
- functional correctness wrt. Sail semantics
- try to make proofs more modular (how ?)

Hiring **PhDs** and **post-docs** in CentraleSupélec, Rennes !
SUSHI team - SecUrity at the Software Hardware Interface (starting June-Sep 2023)

Topics : formal models of processors, security mechanisms, proof methodology, ...

Contact :

pierre.wilke@centralesupelec.fr

guillaume.hiet@centralesupelec.fr