

FORMAL SPECIFICATION AND VERIFICATION OF SECURITY MECHANISMS FOR RISC-V PROCESSORS

PhD defence

Matthieu Baty



CentraleSupélec



UMR

IRISA



Director: Guillaume HIET

Advisors: Pierre WILKE, Alix TRIEU

Cybersecurity

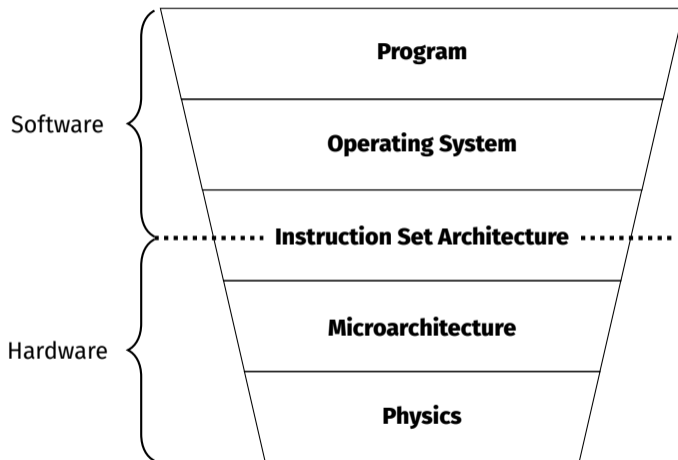
Cybersecurity is about **enforcing security properties** for information systems:

- **Confidentiality**
- **Integrity**
- Availability

Not obvious, as illustrated by many recent incidents, including:

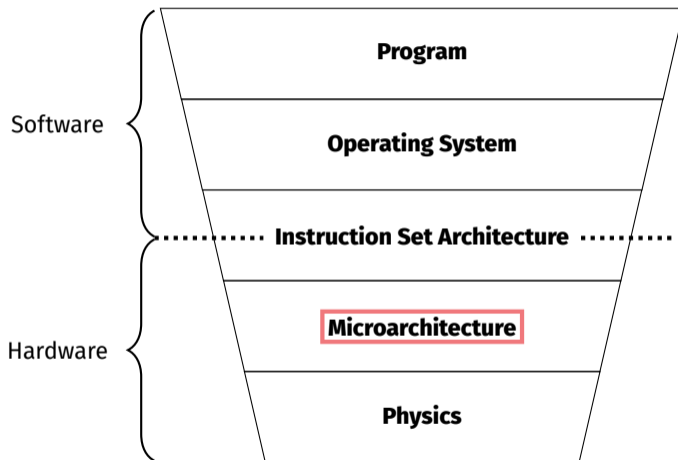
- Pegasus (2020)
- Spectre/Meltdown (2018)

Cybersecurity and hardware



Hardware security is a prerequisite for software security

Cybersecurity and hardware



Hardware security is a prerequisite for software security

Cybersecurity and trust

Cyber Resilience Act (2024): “Rebalance responsibility towards manufacturers”

=> How to prove that security claims are substantiated?

Cybersecurity and trust

Cyber Resilience Act (2024): “Rebalance responsibility towards manufacturers”

=> How to prove that security claims are substantiated?

Common Criteria for Information Technology Security Evaluation:

- **International standard** (ISO/IEC 15408)
- French certification authority: ANSSI
- Highest assurance level: **formal methods**

Topic

“Formal specification and verification of security mechanisms for RISC-V processors”

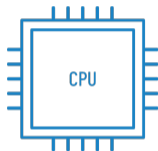
Topic

“Formal specification and verification of security mechanisms for RISC-V processors”

We want to:

Topic

“**Formal specification** and **verification** of **security mechanisms** for **RISC-V processors**”



We want to:

- Develop **RISC-V processors**...

Topic

“**Formal specification** and **verification** of **security mechanisms** for **RISC-V processors**”

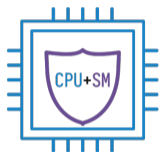


We want to:

- Develop **RISC-V processors**... with **security mechanisms**

Topic

“**Formal specification** and **verification** of **security mechanisms** for **RISC-V processors**”

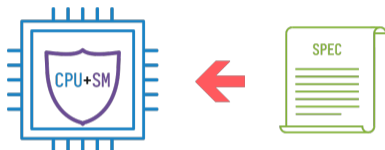


We want to:

- Develop **RISC-V processors**... with **security mechanisms**
- Describe the properties that our mechanisms enforce (**formal specification**)

Topic

“**Formal specification** and **verification** of **security mechanisms** for **RISC-V processors**”



We want to:

- Develop **RISC-V processors**... with **security mechanisms**
- Describe the properties that our mechanisms enforce (**formal specification**)
- Verify that our implementation is correct w.r.t. the specification (**formal proof**)

Research question

“How to formally verify implementations of security mechanisms for processors at the Register-Transfer Level (RTL) of description within a proof assistant?”

Research question

“How to formally verify implementations of security mechanisms for processors at the **Register-Transfer Level** (RTL) of description within a proof assistant?”

Note:

- We do **RTL verification**:
 - Hardware = registers + rules describing how they are updated
 - We handle the **hardware's definition directly**

Research question

“How to formally verify implementations of security mechanisms for processors at the **Register-Transfer Level** (RTL) of description within a **proof assistant**?”

Note:

- We do **RTL verification**:
 - Hardware = registers + rules describing how they are updated
 - We handle the **hardware's definition directly**
- We use **proof assistants**:
 - + Versatile formal methods tools
 - + **Small trusted computing base => high-assurance guarantees**
 - Manual workflow

Plan

- 1 State of the art

Formal verification of hardware

Formal verification is **common in the industry**:

- **Multiple approaches**, including: assertion-based verification, model checking (bounded or unbounded), equivalence checking
- **Many commercial tools**, e.g., OneSpin 360 DV¹

¹<https://onespin.com/products/360-dv-verify/>

Formal verification of hardware

Formal verification is **common in the industry**:

- **Multiple approaches**, including: assertion-based verification, model checking (bounded or unbounded), equivalence checking
- **Many commercial tools**, e.g., OneSpin 360 DV¹

There are **active research communities** on such verification methods:

- Work on formal verification tools²
- Verification of concrete systems³

¹<https://onespin.com/products/360-dv-verify/>

²*AVR: abstractly verifying reachability*, TACAS'20, A. Goel et al.

³*Fault-Resistant Partitioning of Secure CPUs for System Co-Verification against Faults*, CHES'24, S. Tollec et al.

Formal verification of hardware

Formal verification is **common in the industry**:

- **Multiple approaches**, including: assertion-based verification, model checking (bounded or unbounded), equivalence checking
- **Many commercial tools**, e.g., OneSpin 360 DV¹

There are **active research communities** on such verification methods:

- Work on formal verification tools²
- Verification of concrete systems³

However, these approaches **do not rely on proof assistants**

¹<https://onespin.com/products/360-dv-verify/>

²*AVR: abstractly verifying reachability*, TACAS'20, A. Goel et al.

³*Fault-Resistant Partitioning of Secure CPUs for System Co-Verification against Faults*, CHES'24, S. Tollec et al.

Proof assistant-based hardware verification — 1/2

Instruction Set Architecture (ISA) level verification:

- Architectural reasoning
- E.g., Sail⁴
- However, **we are concerned with microarchitectural security properties**

⁴*ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS*, POPL'19, A. Armstrong *et al.*

Proof assistant-based hardware verification — 1/2

Instruction Set Architecture (ISA) level verification:

- Architectural reasoning
- E.g., Sail⁴
- However, **we are concerned with microarchitectural security properties**

Model-based verification:

- The hardware's definition is not verified directly: a model is built first
- E.g., work on DRAM controllers⁵
- However, **the model may not be faithful**

⁴ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS, POPL'19, A. Armstrong et al.

⁵A Coq framework for more trustworthy DRAM controllers, RTNS '22, F. Lisboa Malaquias et al.

Proof assistant-based hardware verification – 2/2

Formal **Hardware Description Languages** (HDLs):

- HDLs equipped with a formal semantics
- E.g., **Kôika**⁶
- However, they **do not include facilities for reasoning about hardware behavior**

⁶*The Essence of BlueSpec*, PLDI'20, T. Bourgeat *et al.*

Proof assistant-based hardware verification — 2/2

Formal **Hardware Description Languages** (HDLs):

- HDLs equipped with a formal semantics
- E.g., **Kôika**⁶
- However, they **do not include facilities for reasoning about hardware behavior**

Verified stacks:

- Cover **both hardware and software** verification
- E.g., CakeML stack⁷
- However, they are **not focused on security properties**

⁶*The Essence of BlueSpec*, PLDI'20, T. Bourgeat *et al.*

⁷*Verified compilation on a verified processor*, PLDI'19, A. Löow *et al.*

Kôika: a formal HDL

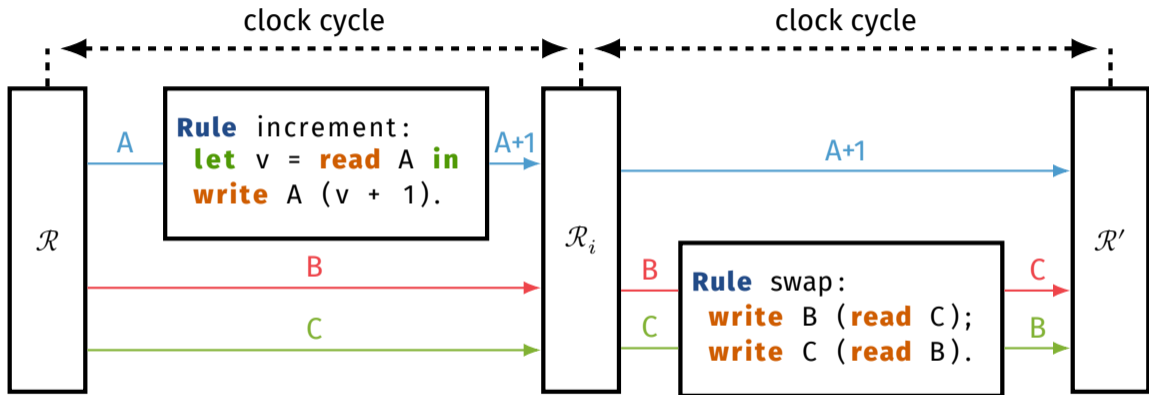
Kôika is a **formal HDL embedded within Coq**.



Kôika is:

- A **RTL language**
- A **rule-based language**
 - Designs are described as a **set of rules**
 - The compiler **infers the control logic** automatically

Kôika semantics, intuitively – 1/3



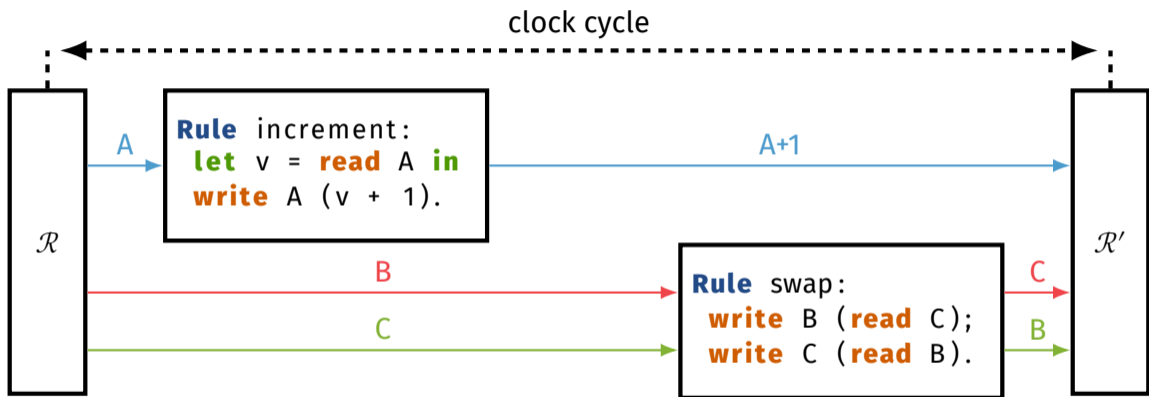
Reads and writes:

- **Reads:** access value at the **cycle's start**
- **Writes:** visible only at the **cycle's end**

One Rule At A Time semantics:

- **As if rules were executed sequentially**

Kôika semantics, intuitively – 1/3



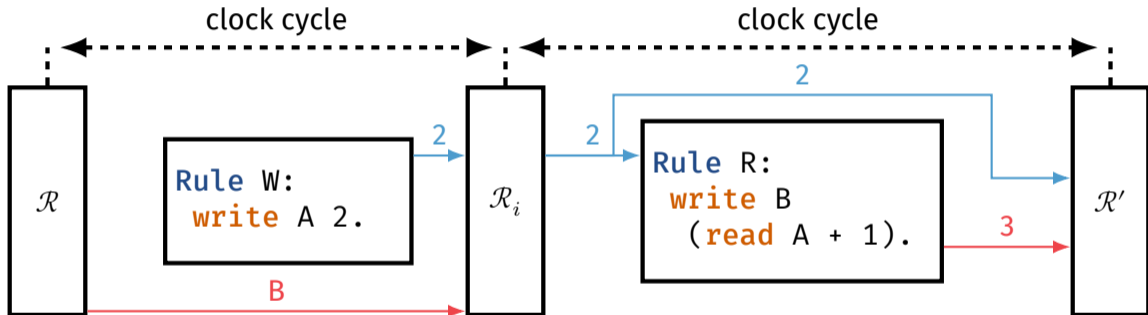
Reads and writes:

- **Reads:** access value at the **cycle's start**
- **Writes:** visible only at the **cycle's end**

One Rule At A Time semantics:

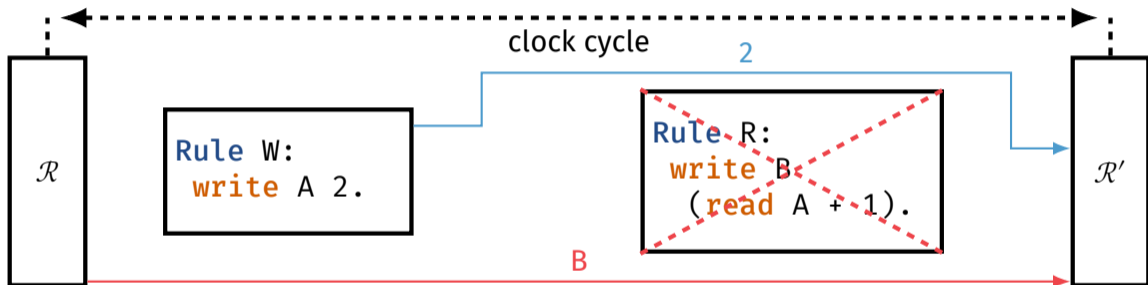
- **As if rules were executed sequentially**
- Actually run in **parallel**, if possible

Kôika semantics, intuitively – 2/3



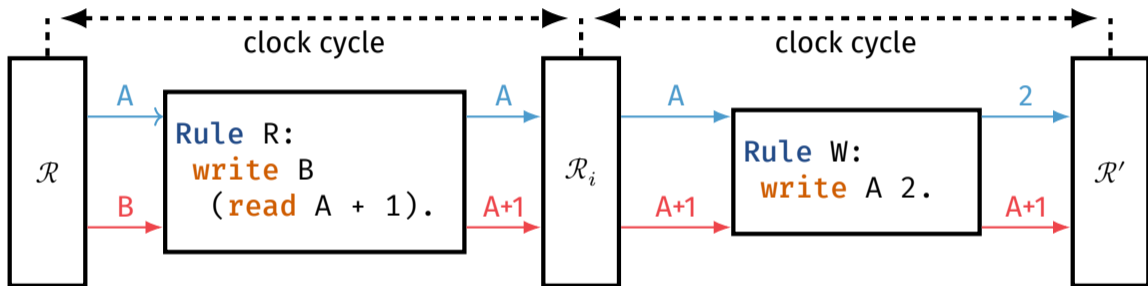
Read after write in the same cycle: conflict.

Kôika semantics, intuitively – 2/3



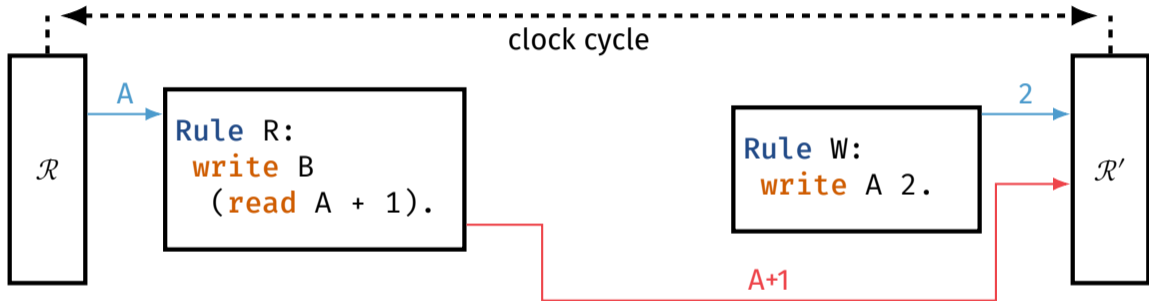
Read after write in the same cycle: conflict.

Kôika semantics, intuitively – 3/3



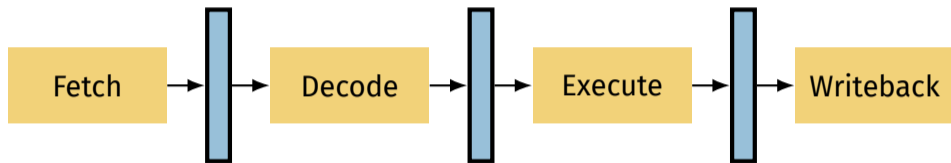
The user-defined **schedule** determines the order in which rules appear to execute.

Kôika semantics, intuitively – 3/3



The user-defined **schedule** determines the order in which rules appear to execute.

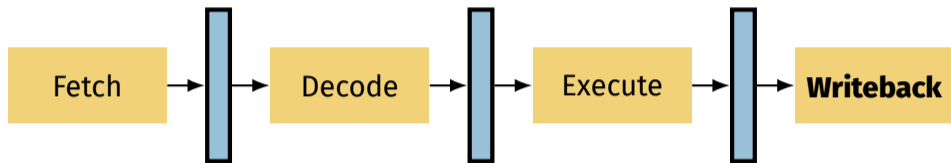
Kôika scheduling and pipelines



For a **pipelined processor**:

- One stage = one rule
- The stages communicate through FIFOs of size 1
- Writing to a full FIFO is considered a conflict
- Consequence: **the stalling behavior is implicit!**

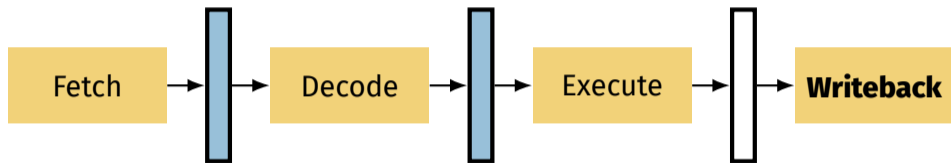
Kôika scheduling and pipelines



For a **pipelined processor**:

- One stage = one rule
- The stages communicate through FIFOs of size 1
- Writing to a full FIFO is considered a conflict
- Consequence: **the stalling behavior is implicit!**

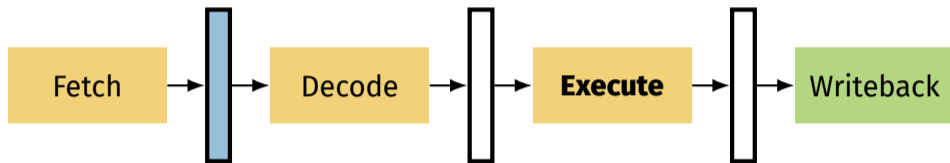
Kôika scheduling and pipelines



For a **pipelined processor**:

- One stage = one rule
- The stages communicate through FIFOs of size 1
- Writing to a full FIFO is considered a conflict
- Consequence: **the stalling behavior is implicit!**

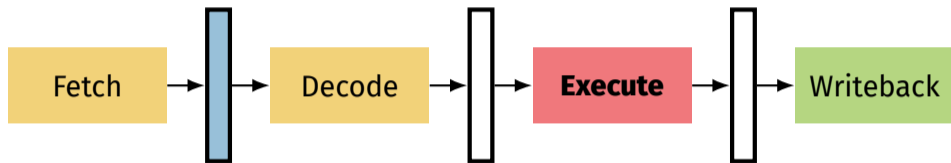
Kôika scheduling and pipelines



For a **pipelined processor**:

- One stage = one rule
- The stages communicate through FIFOs of size 1
- Writing to a full FIFO is considered a conflict
- Consequence: **the stalling behavior is implicit!**

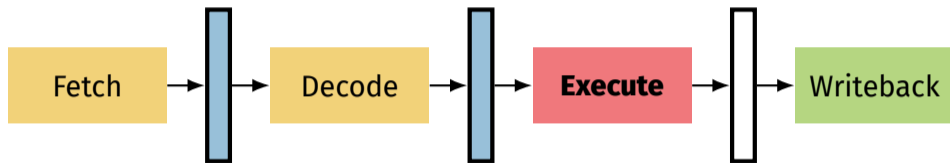
Kôika scheduling and pipelines



For a **pipelined processor**:

- One stage = one rule
- The stages communicate through FIFOs of size 1
- Writing to a full FIFO is considered a conflict
- Consequence: **the stalling behavior is implicit!**

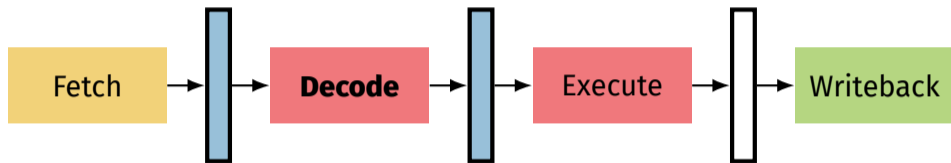
Kôika scheduling and pipelines



For a **pipelined processor**:

- One stage = one rule
- The stages communicate through FIFOs of size 1
- Writing to a full FIFO is considered a conflict
- Consequence: **the stalling behavior is implicit!**

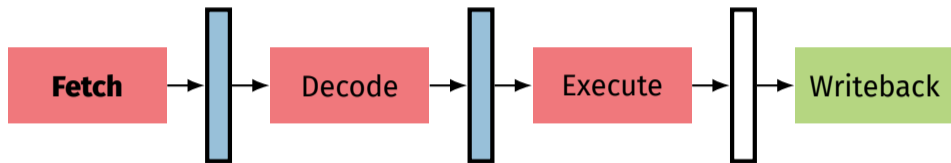
Kôika scheduling and pipelines



For a **pipelined processor**:

- One stage = one rule
- The stages communicate through FIFOs of size 1
- Writing to a full FIFO is considered a conflict
- Consequence: **the stalling behavior is implicit!**

Kôika scheduling and pipelines



For a **pipelined processor**:

- One stage = one rule
- The stages communicate through FIFOs of size 1
- Writing to a full FIFO is considered a conflict
- Consequence: **the stalling behavior is implicit!**

Our contributions

2

A **verification framework**
built around the Kôika
HDL

3

An extension of this
framework that allows
delegation to an **SMT**
solver

4

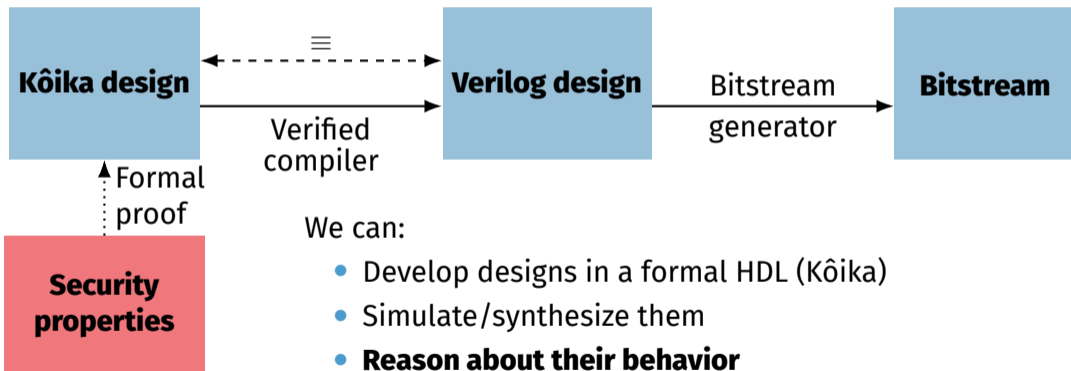
COQQTl, a formalization
of the **industrial-grade**
HDL FIRRTL within Coq

Plan

- 1 State of the art

Overview

We develop a **verification framework** for Kôika.



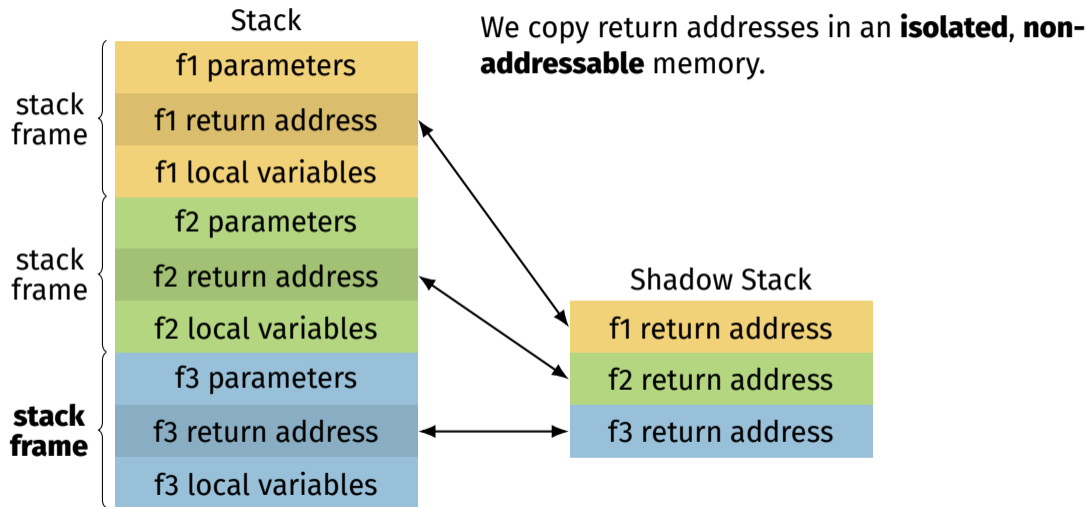
The processor

The Kôika developers wrote **a processor**:

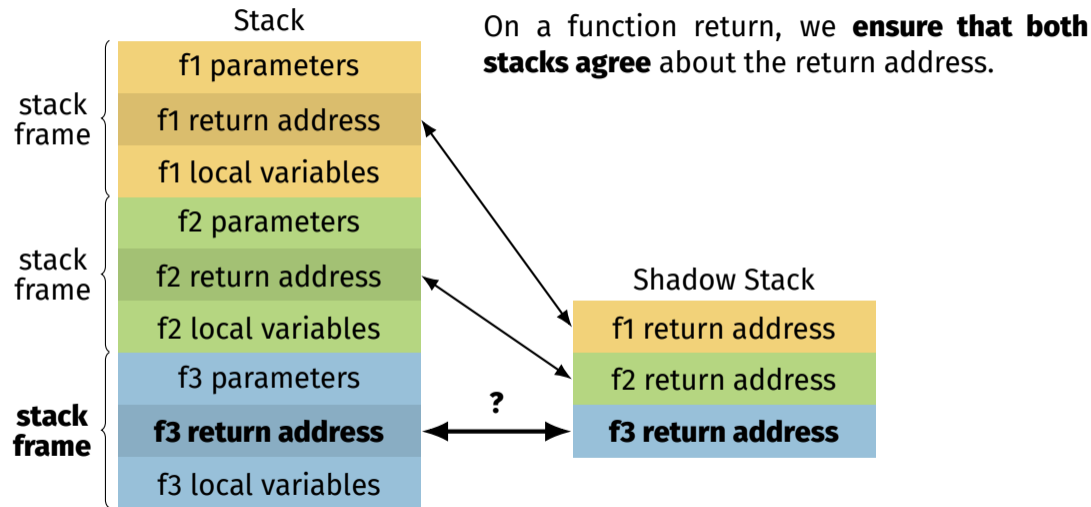
- Embedded-class (4-stage pipeline, RV32I, unprivileged, no interrupts)
- 1000 lines of code
- Synthesizable: can run on FPGAs
- Not formally verified (but passes test suites)

We extend **this processor** with a **verified shadow stack**.

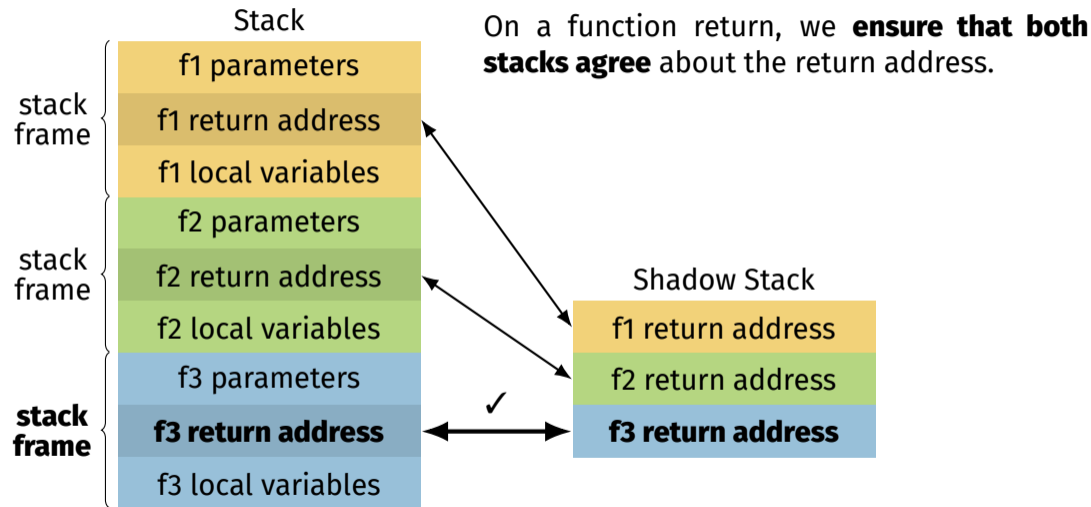
The shadow stack security mechanism



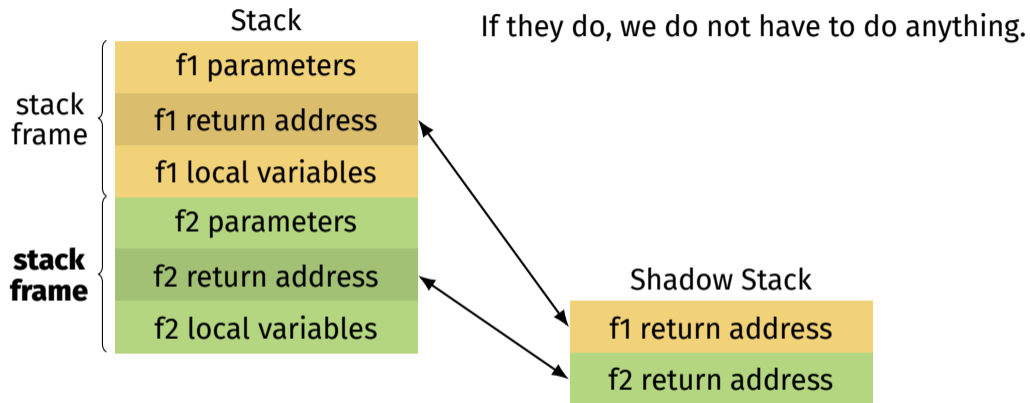
The shadow stack security mechanism



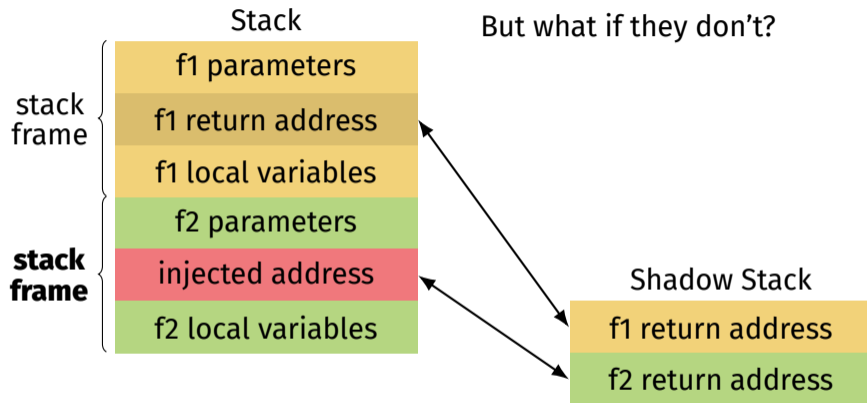
The shadow stack security mechanism



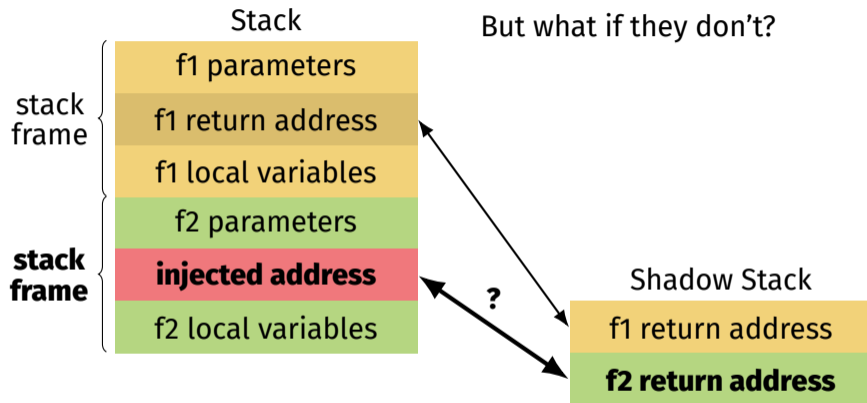
The shadow stack security mechanism



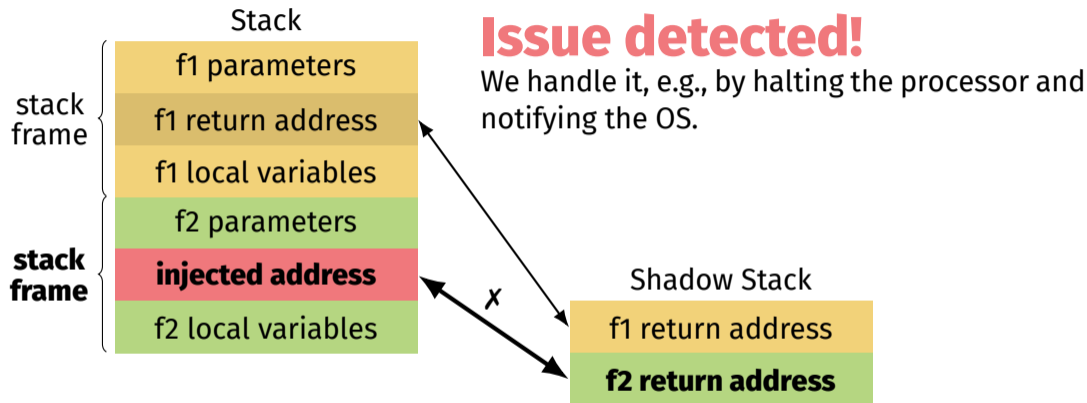
The shadow stack security mechanism



The shadow stack security mechanism



The shadow stack security mechanism



Shadow stack — specification

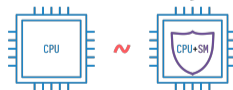
The properties we want to prove:

Shadow stack buffer **overflow**
⇒ the processor **halts**

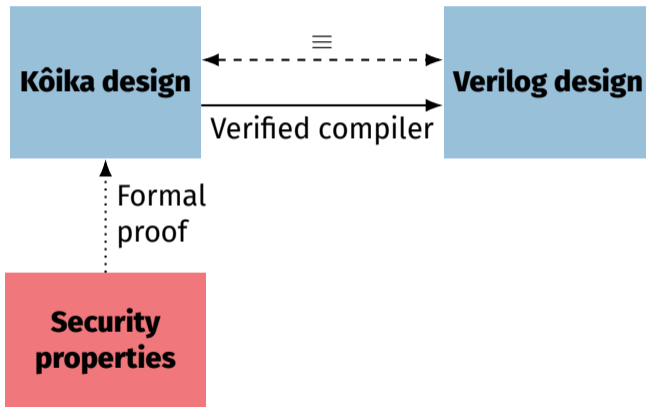
Shadow stack buffer **underflow**
⇒ the processor **halts**

Return to a **modified return address**
⇒ the processor **halts**

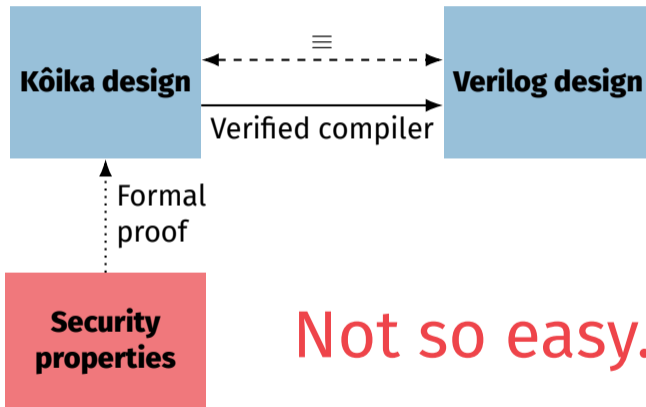
None of the above
⇒ the **behavior is preserved**



Proving properties of Kôika designs



Proving properties of Kôika designs



Performance issues with vanilla Kôika

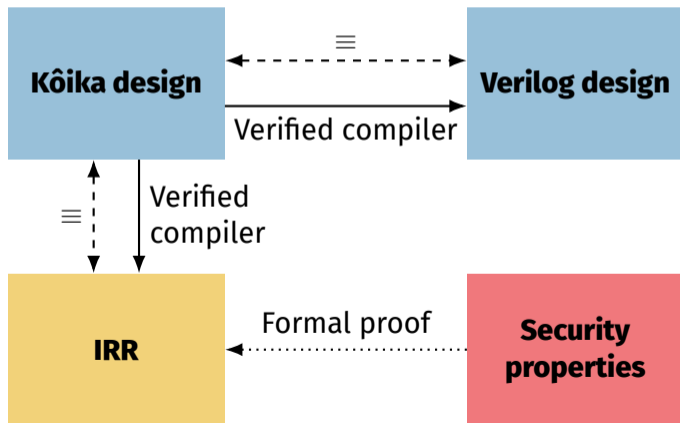
For non-trivial designs, most tactics take minutes to hours and consume large amounts of RAM.

No single cause, some factors are:

- Rigidity of Coq's evaluation tactics
- Complexity of Kôika's semantics

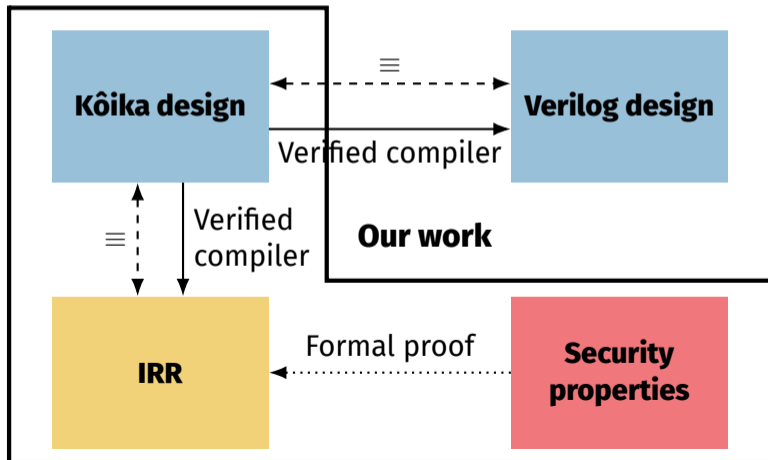
Proofs on Kôika designs

Somewhat counter-intuitively, **compiling** high-level Kôika designs into a lower-level **Intermediate Representation for Reasoning** (IRR) facilitates proofs.



Proofs on Kôika designs

Somewhat counter-intuitively, **compiling** high-level Kôika designs into a lower-level **Intermediate Representation for Reasoning** (IRR) facilitates proofs.



Building IRRs — 1/2

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 s : [r1].

Building IRRs — 1/2

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 s : [r1].

v_1	a
v_2	b
v_3	0
v_4	1
v_5	2

Building IRRs – 1/2

Registers : {a, b}.

Rule r1 :

```
if v1 == v3 then
  write b (v2 - v2)
else
  (write b v4; write a v5).
```

v_1	a
v_2	b
v_3	0
v_4	1
v_5	2

Schedule s : [r1].

Building IRRs – 1/2

Registers : {a, b}.

Rule r1 :

```

if v1 == v3 then
  write b (v2 - v2)
else
  (write b v4; write a v5).
  
```

v_1	a
v_2	b
v_3	0
v_4	1
v_5	2
v_6	$v_1 == v_3$
v_7	$v_2 - v_2$

Schedule s : [r1].

Building IRRs – 1/2

Registers : {a, b}.

Rule r1 :

```

if v6 then
  write b v7
else
  (write b v4; write a v5).
  
```

v_1	a
v_2	b
v_3	0
v_4	1
v_5	2
v_6	$v_1 == v_3$
v_7	$v_2 - v_2$

Schedule s : [r1].

Building IRRs — 1/2

Registers : {a, b}.

Rule r1 :

```
write a (if v6 then a else v5);
write b (if v6 then v7 else v4).
```

v_1	a
v_2	b
v_3	0
v_4	1
v_5	2
v_6	$v_1 == v_3$
v_7	$v_2 - v_2$

Schedule s : [r1].

Building IRRs – 1/2

Registers : {a, b}.

Rule r1 :

```
write a (if v6 then a else v5);
write b (if v6 then v7 else v4).
```

Schedule s : [r1].

v_1	a
v_2	b
v_3	0
v_4	1
v_5	2
v_6	$v_1 == v_3$
v_7	$v_2 - v_2$
v_8 (a)	if v_5 then a else v_4
v_9 (b)	if v_5 then v_6 else v_3

Building IRRs — 1/2

Registers : {a, b}.

Rule r1 :

write a v8;

write b v9.

Schedule s : [r1].

v_1	a
v_2	b
v_3	0
v_4	1
v_5	2
v_6	$v_1 == v_3$
v_7	$v_2 - v_2$
v_8 (a)	if v_5 then a else v_4
v_9 (b)	if v_5 then v_6 else v_3

Building IRRs – 2/2

An IRR:

- Is an **explicit representation** of how register values are updated during a cycle
- **Conflicts management** is encoded **explicitly** in these expressions

We prove the Kôika to IRR compiler correct.

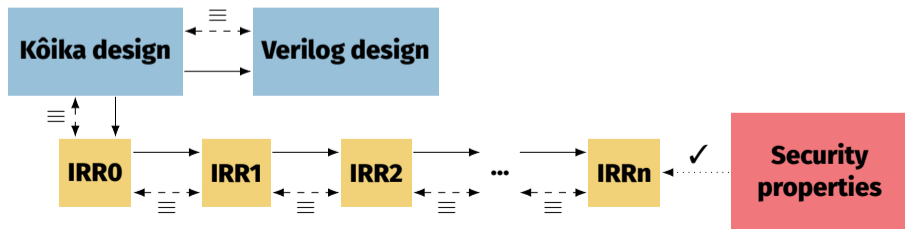
How do we actually reason on an IRR?

Verification framework

We implement a collection of **verified transformations** on IRRs, e.g.:

- Prune
- ExploitReg
- Collapse
- Simplify
- SimplifyTargeted
- ExploitRegPartial
- ReplaceVar
- ReplaceSubact

These transformations can be applied **manually by the user** or **using automatic tactics**.



Reasoning on IRRs — an example

Assume that we want to show that when a 's initial value is 0, then the final value of b is 0.

Reasoning on IRRs – an example

Assume that we want to show that when a 's initial value is 0, then the final value of b is 0.

	Initial
v_1	a
v_2	b
v_3	0
v_4	1
v_5	2
v_6	$v_1 == 0$
v_7	$v_2 - v_2$
v_8 (a)	if v_6 then a else v_5
v_9 (b)	if v_6 then v_7 else v_4

Reasoning on IRRs – an example

Assume that we want to show that when a 's initial value is 0, then the final value of b is 0.

	Initial	Prune (reg b)
v_1	a	
v_2	b	
v_3	0	
v_4	1	
v_5	2	
v_6	$v_1 == 0$	
v_7	$v_2 - v_2$	
v_8 (a)	if v_6 then a else v_5	
v_9 (b)	if v_6 then v_7 else v_4	

Reasoning on IRRs – an example

Assume that we want to show that when a 's initial value is 0, then the final value of b is 0.

	Initial	Prune (reg b)	ExploitReg
v_1	a		○
v_2	b		
v_3	0		
v_4	1		
v_5	2		
v_6	$v_1 == 0$		
v_7	$v_2 - v_2$		
v_8 (a)	if v_6 then a else v_5		
v_9 (b)	if v_6 then v_7 else v_4		

Reasoning on IRRs – an example

Assume that we want to show that when a 's initial value is 0, then the final value of b is 0.

	Initial	Prune (reg b)	ExploitReg	Collapse
v_1	a		0	
v_2	b			
v_3	0			
v_4	1			
v_5	2			
v_6	$v_1 == 0$			$0 == 0$
v_7	$v_2 - v_2$			$b - b$
v_8 (a)	if v_6 then a else v_5			
v_9 (b)	if v_6 then v_7 else v_4			if v_6 then v_7 else 1

Reasoning on IRRs – an example

Assume that we want to show that when a 's initial value is 0, then the final value of b is 0.

	Initial	Prune (reg b)	ExploitReg	Collapse	Simplify
v_1	a		0		
v_2	b				
v_3	0				
v_4	1				
v_5	2				
v_6	$v_1 == 0$			$0 == 0$	1
v_7	$v_2 - v_2$			$b - b$	
v_8 (a)	if v_6 then a else v_5				
v_9 (b)	if v_6 then v_7 else v_4			if v_6 then v_7 else 1	

Reasoning on IRRs – an example

Assume that we want to show that when a 's initial value is 0, then the final value of b is 0.

	Initial	Prune (reg b)	ExploitReg	Collapse	Simplify	Collapse + Simplify
v_1	a		0			
v_2	b					
v_3	0					
v_4	1					
v_5	2					
v_6	$v_1 == 0$			$0 == 0$	1	
v_7	$v_2 - v_2$			$b - b$		
v_8 (a)	if v_6 then a else v_5					
v_9 (b)	if v_6 then v_7 else v_4			if v_6 then v_7 else 1		b - b

Quantitative summary

We successfully verified our shadow stack example using this method!

Furthermore, we **synthesized** our design and ensured that it behaves as expected on real programs.

Verification duration: 10m35s, RAM usage: > 16GiB

Summary

We **extend Kôika**, a formal HDL built within Coq, with a **verification framework**:

- We build a **verified compiler** from Kôika designs to IRR, a **custom, explicit representation**
- We define a set of **verified transformation passes** to progressively simplify designs

We **verify a shadow stack mechanism** built into a **synthesizable** RISC-V processor.

We presented this work at CSF'23⁸

Artifacts available⁹

⁸A *generic framework to develop and verify security mechanisms at the microarchitectural level: application to control-flow integrity*, CSF'23, M. Baty et al.

⁹https://gitlab.inria.fr/SUSHI-public/FMH/koika/-/tree/CSF_2023

Plan

- 1 State of the art

Overview

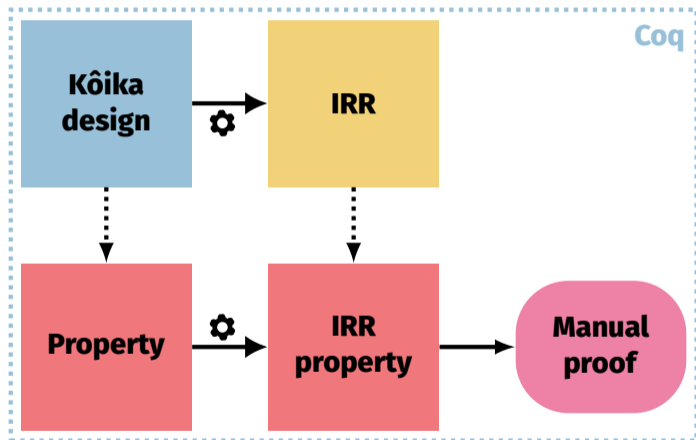
We built a **verification framework for Kôika**. We **successfully verified our target** with it. This was mostly a **manual process** and it was **sensitive to design changes**.

We extend this framework through the inclusion of a highly efficient **SMT solver** (z3¹⁰):

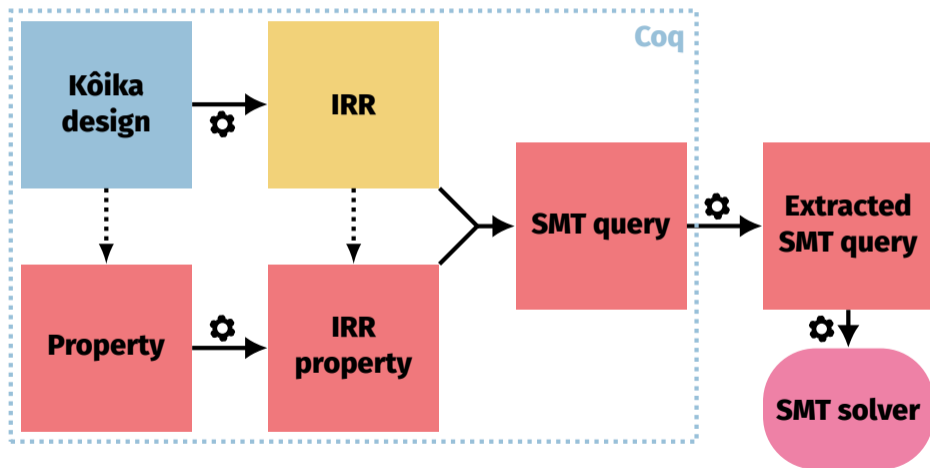
- We can discharge most goals to the solver
- Coq facilities remain available for more complex ones
- More importantly, Coq certifies that the IRR generation is correct

¹⁰<https://github.com/Z3Prover/z3>

Workflow



Workflow



IRR to SMT conversion

SMT queries are:

- **Written manually**
- Very **close to the original statements**
- We prove that the new queries are equivalent to the previous ones

We rely heavily on our **IRR**:

- IRR is an **explicit representation** of Kôika designs
- We **automatically generate an SMT variable per IRR variable**
- References to registers are resolved to IRR variables

Application to an extended processor

We **extend the processor** (work of Gabriel Defresne, Summer 2024 intern):

- We now support **interruptions** and **exceptions**
- Almost a full rewrite

Previously, we would have needed to **rewrite the entire proof**.
Now, the **SMT solver handles the changes on its own**.

Quantitative summary and comparison

Previous duration	(base processor)	10m35.57s
New duration	(base processor)	4.30s

For the base processor, this corresponds to a **speedup of about 150**.

New duration (extended processor) **41.22s**

More importantly, **producing this proof is much easier**: we do not search for a proof manually.

Summary

We **extended our verification framework**, following a **hybrid Coq/SMT** approach to enable **more automatic** forms of verification.

Artifacts available¹¹

However:



- Kôika is an **academic language** with no industrial use
- Kôika **lacks major features** for scalability: no modules

¹¹<https://gitlab.inria.fr/SUSHI-public/FMH/koika> (framework),
<https://gitlab.inria.fr/SUSHI-public/FMH/herve> (processor)

Plan

- 1 State of the art

Overview

We introduce  **COQQTL**, a Coq formalization of the  **FIRRTL** HDL.

FIRRTL:

- Originates as an **intermediate representation for the Chisel HDL**
- Can be used as a **standalone HDL**: human-readable
- Integrated into the LLVM CIRCT compiler

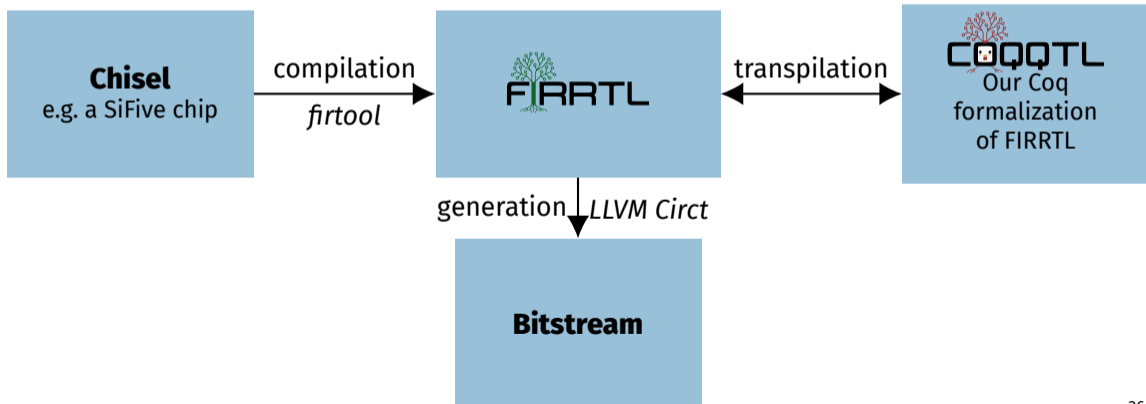
Chisel:

- A **modern HDL** embedded within Scala
- Used in academia (e.g., Rocket Chip RISC-V processor generator)
- Used in the industry (e.g., Google Tensor Processing Unit)

Workflow

There are two ways of using COQQTL:

- **Import Chisel** designs compiled to FIRRTL into Coq
- **Build COQQTL designs directly** in Coq and export them



COQQTL vs. Kôika

```
circuit c:
  module transform {...}
  module m:
    input clk: Clock
    input a: {
      w: UInt<32>, v: UInt<1>
    }
    reg b: UInt<32>, clk

  instance t of transform
  when a.v:
    t.in1 <= a.w
    t.in2 <= b
    b <= t.out
```

In comparison with Kôika:

- **Not rule-based:**
 - More explicit
 - No single, implicit clock
- **Native modules:**
 - Less duplicated code
 - Interesting consequences for verification
- **Rich type system:**
 - Features such as native structs and enums
 - Good for expressivity

Specification and semantics

We start from **FIRRTL's specification** v3.2.0¹²:

- 81 pages long document
- Not a formal specification

The formalization of this semantics comes with a **set of challenges**:

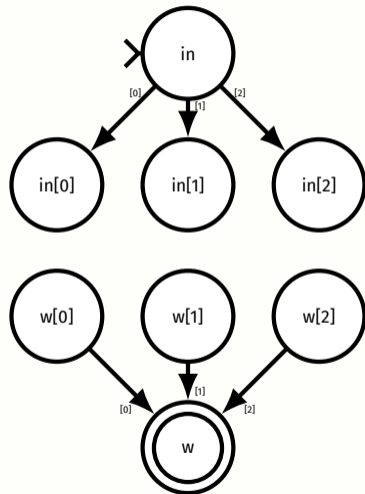
- **Discrepancies** between the specification and the implementation
- Overall **complexity**:
 - Rich type system
 - Modules
 - Detection of combinational loops

¹²<https://github.com/chipsalliance/firrtl-spec/releases/download/v3.2.0/spec.pdf>

Challenge – detection of combinational loops

```
input in: UInt<8>[3]  
wire w: UInt<8>[3]
```

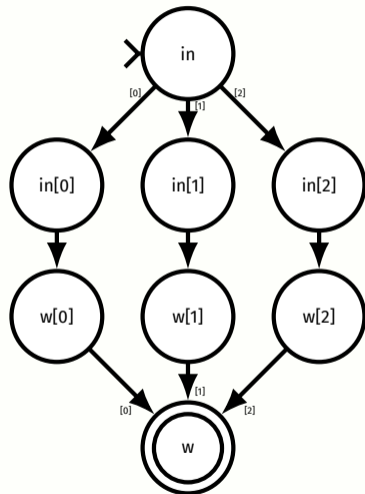
- Two vector-typed components:
 - **in**: input, read-only
 - **w**: wire (**combinational type**)
- We split these registers into their atomic components



Challenge – detection of combinational loops

```
input in: UInt<8>[3]
wire w: UInt<8>[3]
w    <= in
```

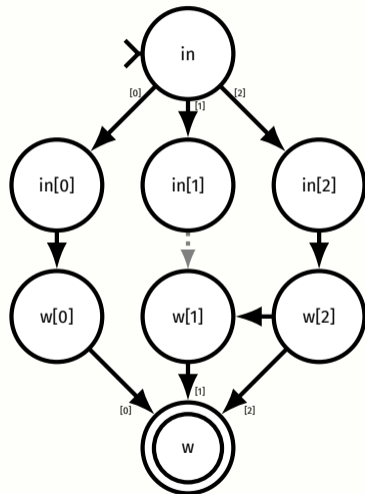
- We connect each subcomponent of **w** to the matching subcomponent of **in**



Challenge – detection of combinational loops

```
input in: UInt<8>[3]
wire w: UInt<8>[3]
w    <= in
w[1] <= w[2]
```

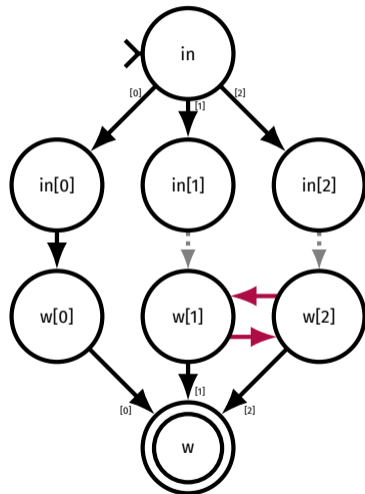
- The connection from **in[1]** to **w[1]** gets overridden



Challenge – detection of combinational loops

```
input in: UInt<8>[3]
wire w: UInt<8>[3]
w    <= in
w[1] <= w[2]
w[2] <= w[1]
```

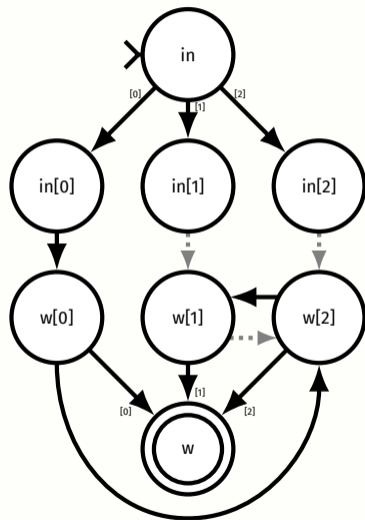
- The connection from **in[2]** to **w[2]** gets overridden
- **w[1]** is defined by **w[2]**, **w[2]** is defined by **w[1]**
- **Loop!**



Challenge – detection of combinational loops

```
input in: UInt<8>[3]
wire w: UInt<8>[3]
w    <= in
w[1] <= w[2]
w[2] <= w[1]
w[2] <= w[0]
```

- The connection from **w[1]** to **w[2]** gets overridden
- **No more loop**



Challenge – detection of combinational loops

```
input in: UInt<8>[3]
```

```
wire w: UInt<8>[3]
```

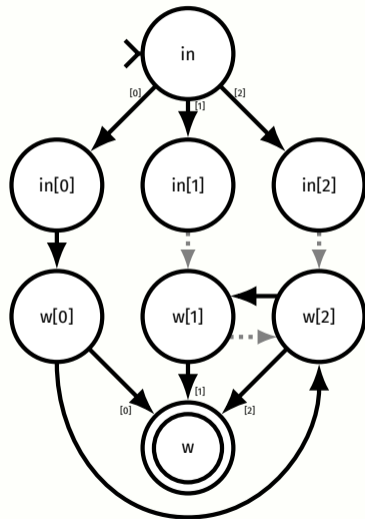
```
w    <= in
```

```
w[1] <= w[2]
```

```
w[2] <= w[1]
```

```
w[2] <= w[0]
```

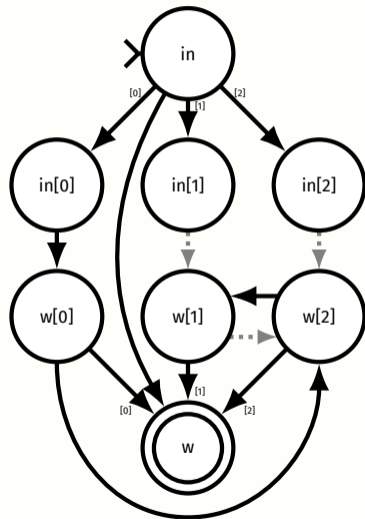
- Here, we split everything into atomic components (naive solution):
 - + Simple
 - Performance impact



Challenge – detection of combinational loops

```
input in: UInt<8>[3000]
wire w: UInt<8>[3000]
w    <= in
w[1] <= w[2]
w[2] <= w[1]
w[2] <= w[0]
```

- In practice, we only introduce subcomponents when strictly necessary



A COQQTTL IRR

We express  COQQTTL's semantics **operationally** by **compiling designs to an IRR**:

- Conceptually **similar to the Kôika IRR**
- We add special constructs to, e.g., update the n-th value of a vector
- We **check that designs are well-formed** while building the IRR

Summary

We formalize  **FIRRTL** within Coq as  **COQQTL**:

- Unlike Kôika, FIRRTL is an **industrial-grade HDL**
- We **compile COQQTL designs to an IRR**
- No verification framework yet: future work

Artifacts available¹³

¹³<https://gitlab.inria.fr/SUSHI-public/FMH/coqqtl>

Plan

- 1 State of the art

Summary

We considered the **formal verification** of **security mechanisms** for processors at the **register transfer level** of description within **proof assistants**.

We presented our **three main contributions**:

2

A **verification framework**
built around the Kôika
HDL

Presented at **CSF'23**
Artifacts available

3

An extension of this
framework that allows
delegation to an **SMT**
solver

Artifacts available

4

COQQTL, a formalization
of the **industrial-grade**
HDL FIRRTL within Coq

Artifacts available

Perspectives – 1/2

Making verification **more accessible**:

- Port our Kôika verification framework for COQQTl
- Automatically deriving proof obligations from annotations

Verifying **more complex examples**:

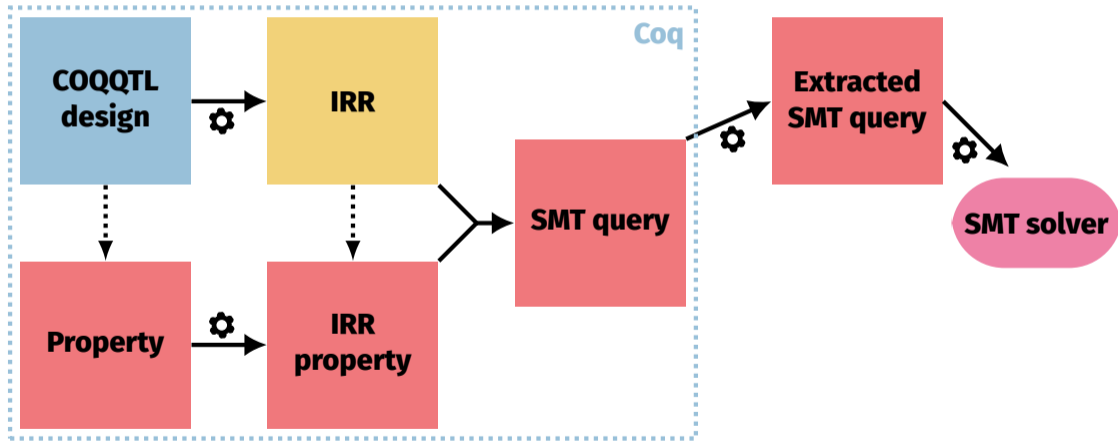
- Building up from processors like the Kôika one to more complex ones¹⁴
- Considering more complex security mechanisms, such as capabilities¹⁵

¹⁴<https://github.com/chipsalliance/rocket-chip>

¹⁵<https://github.com/CTSRD-CHERI/sail-cheri-riscv>

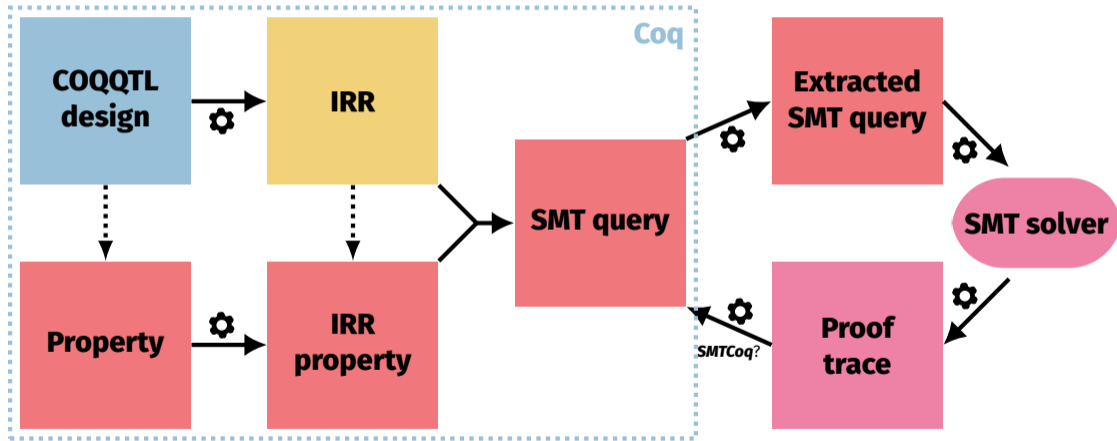
Perspectives – 2/2

Build a verified **SMT binding** to preserve the trusted computing base.



Perspectives – 2/2

Build a verified **SMT binding** to preserve the trusted computing base.



Thank you!