Formalizing Hardware Security Mechanisms

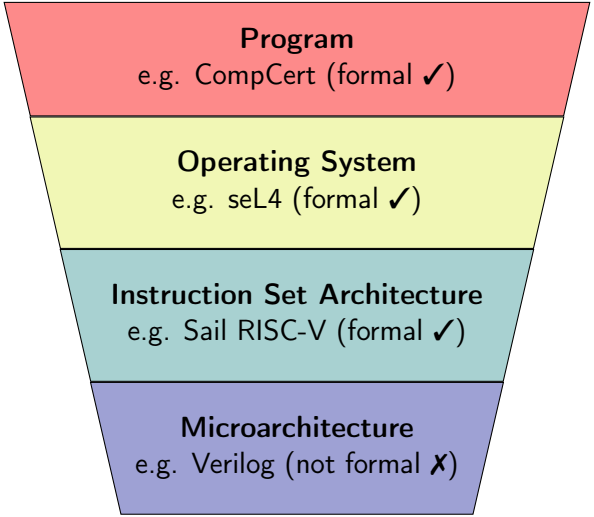
or "A Generic Framework to Develop and Verify Security Mechanisms at the Microarchitectural Level: Application to Control-Flow Integrity"

Matthieu Baty[1,3], Pierre Wilke[1], Guillaume Hiet[1], Arnaud Fontaine[2], Alix Trieu[2]

[1]CIDRE, CentraleSupélec Rennes, Inria, [2]ANSSI
[3]Currently visiting student at the SystemF lab (EPFL)

July 12th, 2023

Motivation: a stack of abstractions



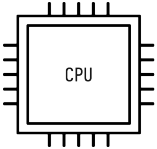
The **security of a layer** depends on the correctness of **all those below** it.

The **microarchitecture** is the lowest common denominator.

Context

We want to:

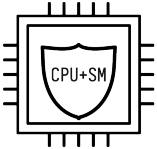
Context



We want to:

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

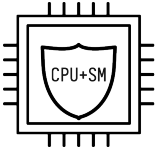
Context



We want to:

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

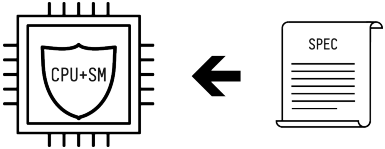
Context



We want to:

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

Context



We want to:

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

In short

We build a **formal verification framework** for a Hardware Description Language.

In short

We build a **formal verification framework** for a Hardware Description Language.

We are able to:

In short

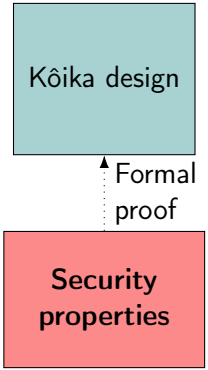
We build a **formal verification framework** for a Hardware Description Language.

We are able to:

- Develop hardware with a formal HDL

In short

We build a **formal verification framework** for a Hardware Description Language.

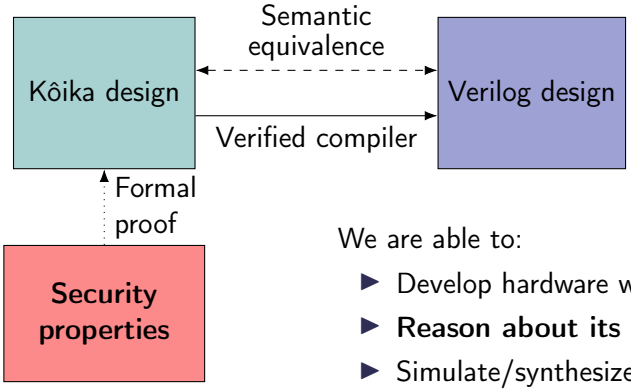


We are able to:

- Develop hardware with a formal HDL
- **Reason about its behavior**

In short

We build a **formal verification framework** for a Hardware Description Language.

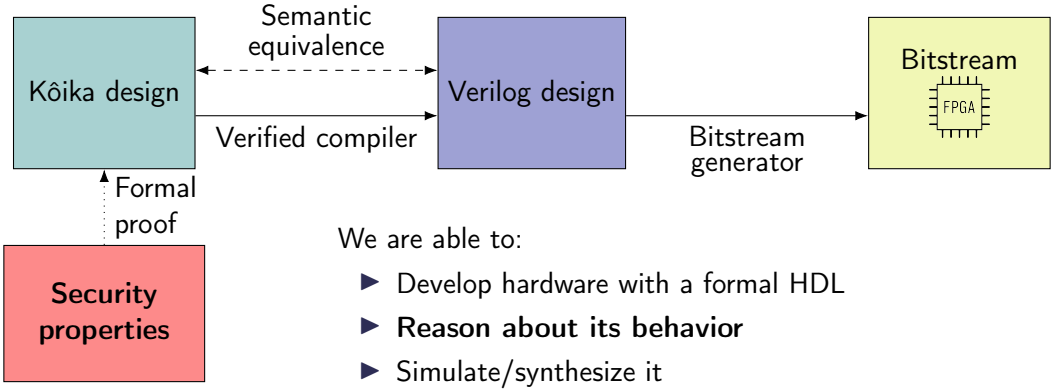


We are able to:

- Develop hardware with a formal HDL
- **Reason about its behavior**
- Simulate/synthesize it

In short

We build a **formal verification framework** for a Hardware Description Language.



We are able to:

- Develop hardware with a formal HDL
- **Reason about its behavior**
- Simulate/synthesize it

Outline

1. The Kôika language
2. Implementing and specifying a shadow stack in Kôika
3. A verification framework

Outline

The Kôika project

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

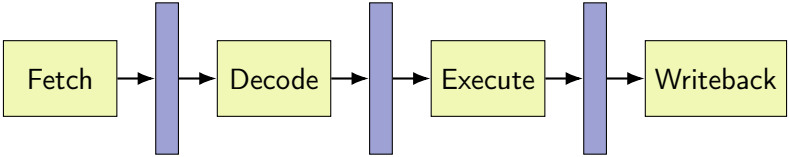
A **Hardware Description Language** embedded in Coq.

The Kôika HDL

Kôika is a **rule-based** register-transfer level language:

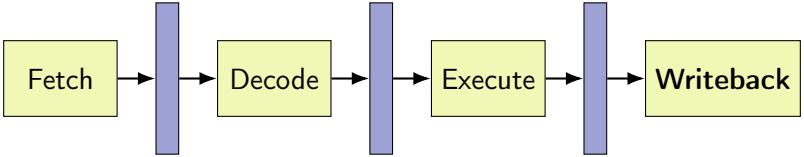
- Hardware is described as a **set of rules**
- During each cycle, all rules are scheduled to be executed
- The semantics treats the **rules sequentially** ...
- ... but the compiler ensures that everything **runs in parallel** in the end
- A rule may be **cancelled** in the presence of **conflicts**

High-level example: a pipelined processor



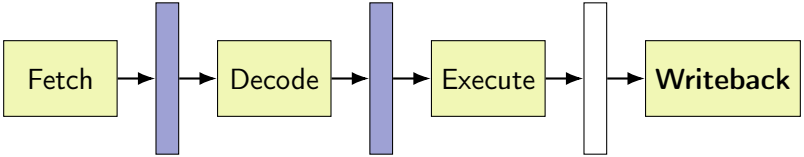
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit**!

High-level example: a pipelined processor



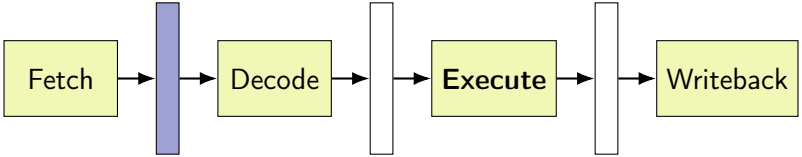
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit**!

High-level example: a pipelined processor



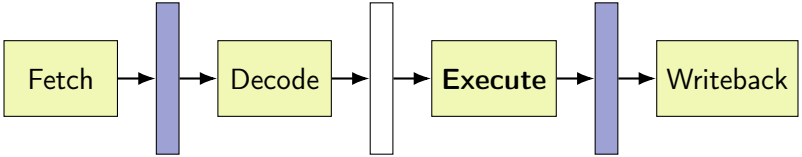
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit**!

High-level example: a pipelined processor



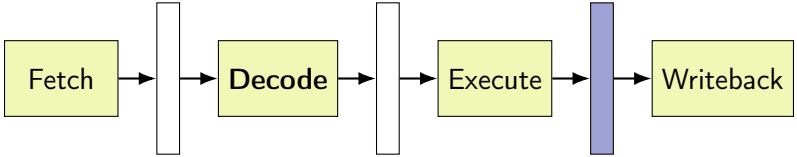
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit**!

High-level example: a pipelined processor



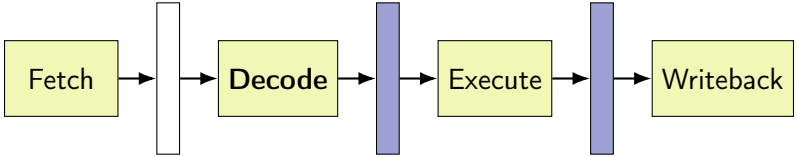
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit**!

High-level example: a pipelined processor



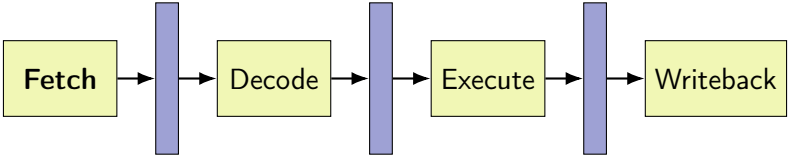
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit**!

High-level example: a pipelined processor



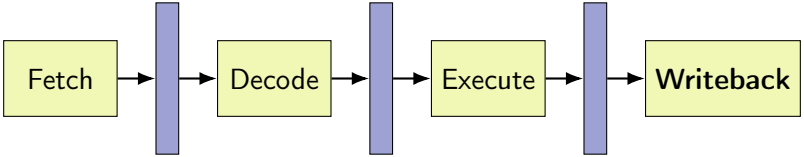
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit**!

High-level example: a pipelined processor



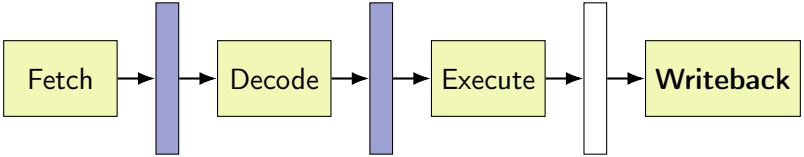
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit**!

High-level example: a pipelined processor



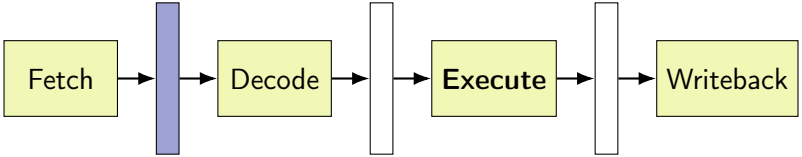
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit**!

High-level example: a pipelined processor



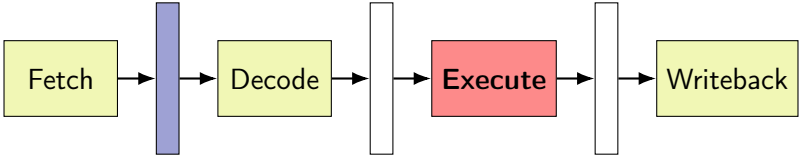
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit**!

High-level example: a pipelined processor



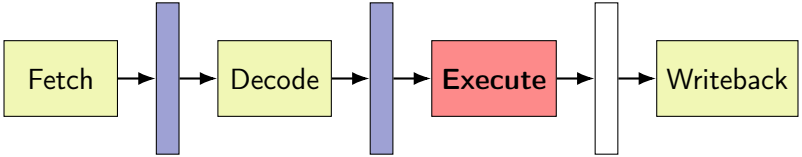
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit**!

High-level example: a pipelined processor



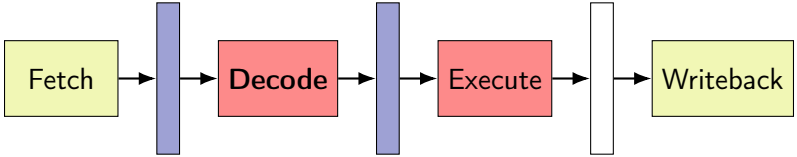
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit**!

High-level example: a pipelined processor



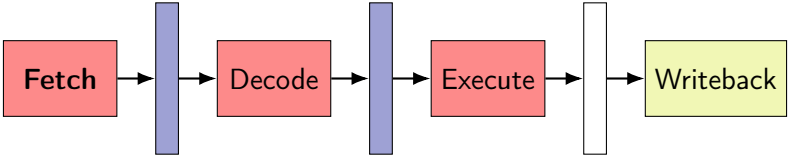
- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit**!

High-level example: a pipelined processor



- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit**!

High-level example: a pipelined processor



- One stage = one rule
- The stages communicate through buffers (FIFOs of size 1)
- Writing to a full FIFO is considered a conflict
- Rules are skipped on cycles on which their inclusion would lead to conflicts
- Consequence: **the stalling behavior is implicit**!

The processor

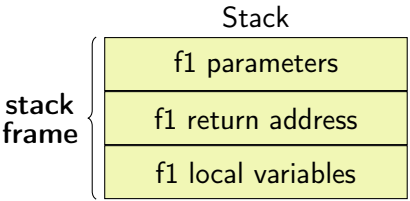
The Kôika developers actually provide such a processor (RISC-V):

- Embedded-class (4-stage pipeline, RV32I, unprivileged, no interrupts)
- 1000 lines of code
- Can run on FPGAs
- Not formally verified

We will **extend** this processor with a **certified security mechanism**.

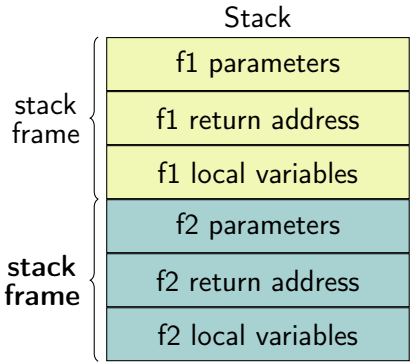
Outline

Motivation: return-oriented programming



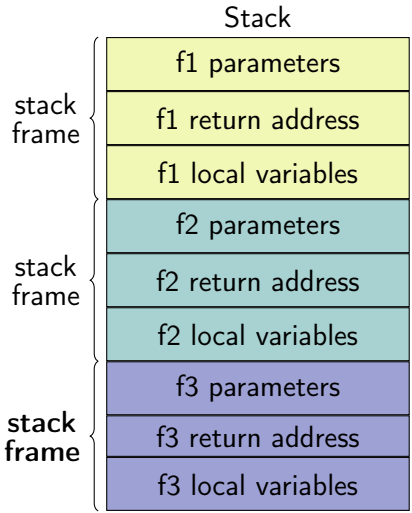
The stack stores information about the active functions.

Motivation: return-oriented programming



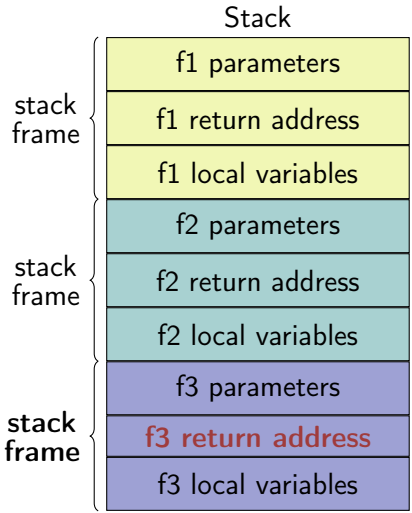
Function calls push the appropriate data on the stack.

Motivation: return-oriented programming



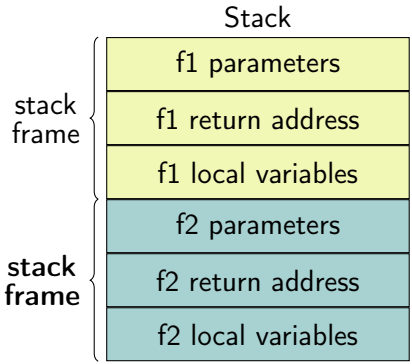
Function calls push the appropriate data on the stack.

Motivation: return-oriented programming



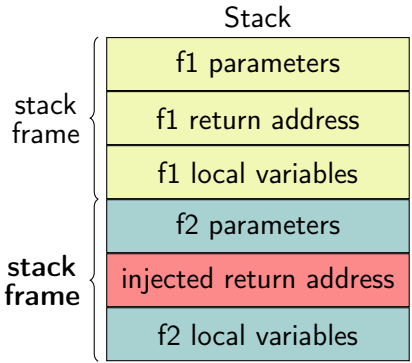
When a function returns, we jump back to **the address stored on the stack**...

Motivation: return-oriented programming



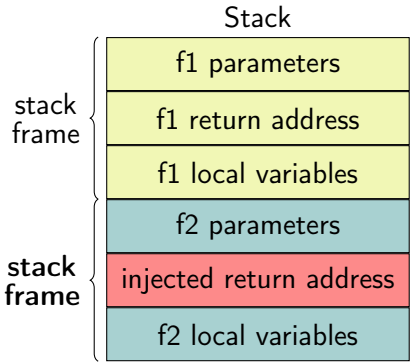
. . . and the stack is popped.

Motivation: return-oriented programming



Code can overwrite the stack, sometimes with malicious intents.
What happens when a return is reached?

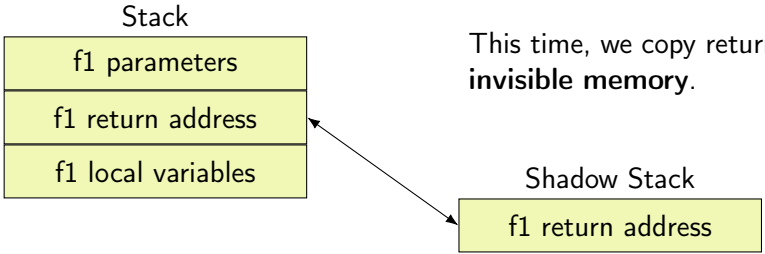
Motivation: return-oriented programming



System compromised!

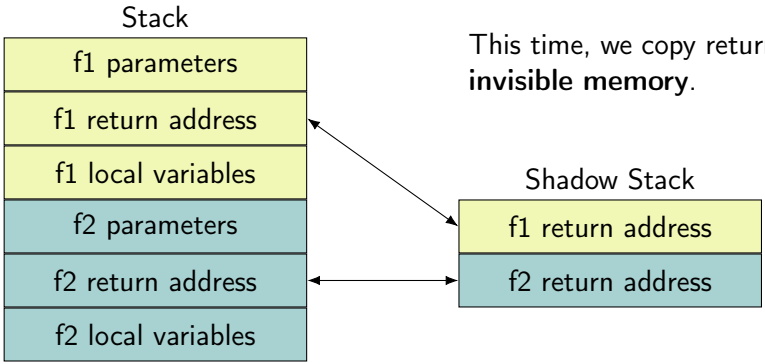
Let's start again but with a shadow stack (*à la* Intel CET).

Shadow stack — principle



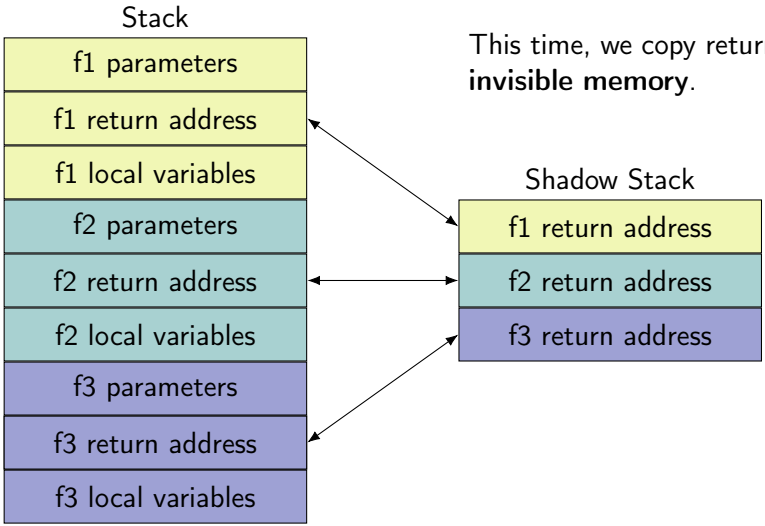
This time, we copy return addresses in a **program invisible memory**.

Shadow stack — principle



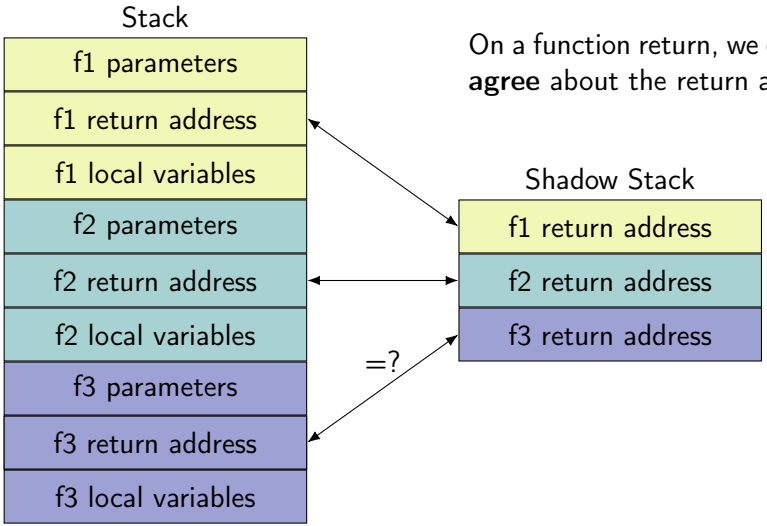
This time, we copy return addresses in a **program invisible memory**.

Shadow stack — principle



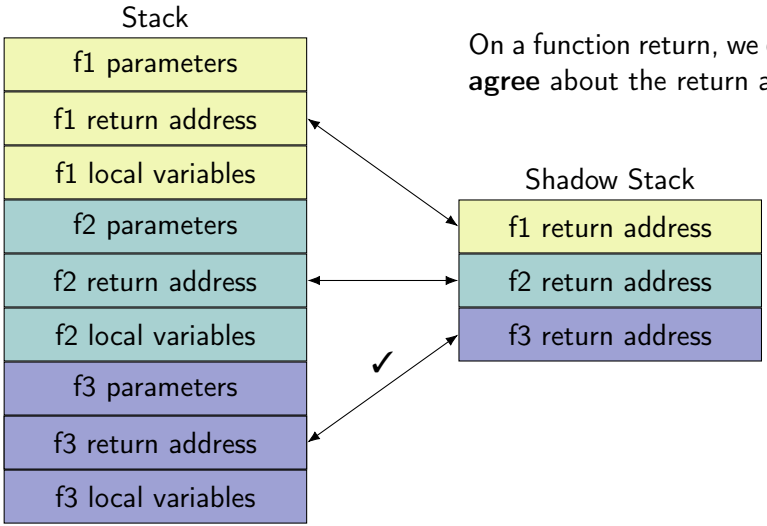
This time, we copy return addresses in a **program invisible memory**.

Shadow stack — principle



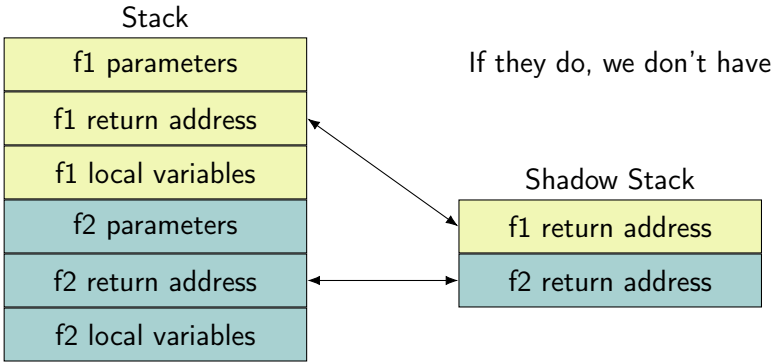
On a function return, we **ensure that both stacks agree** about the return address.

Shadow stack — principle



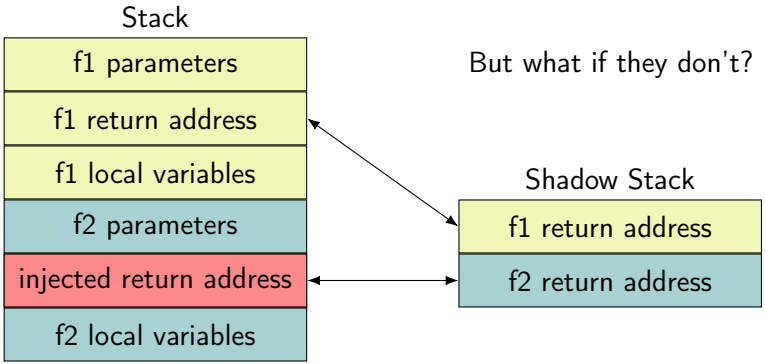
On a function return, we **ensure that both stacks agree** about the return address.

Shadow stack — principle



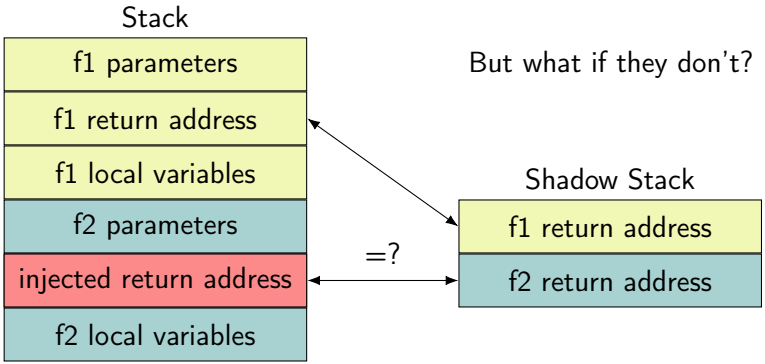
If they do, we don't have to do anything.

Shadow stack — principle



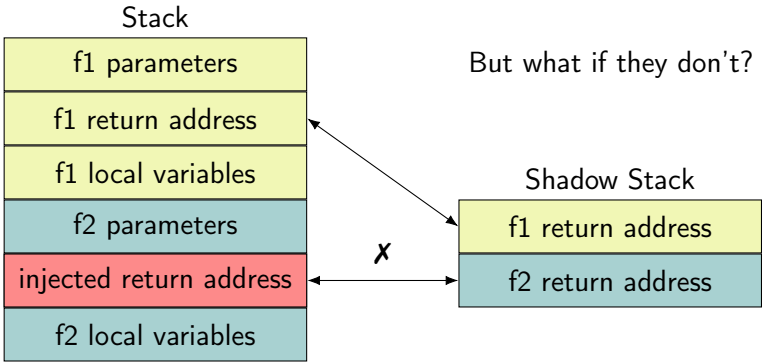
But what if they don't?

Shadow stack — principle



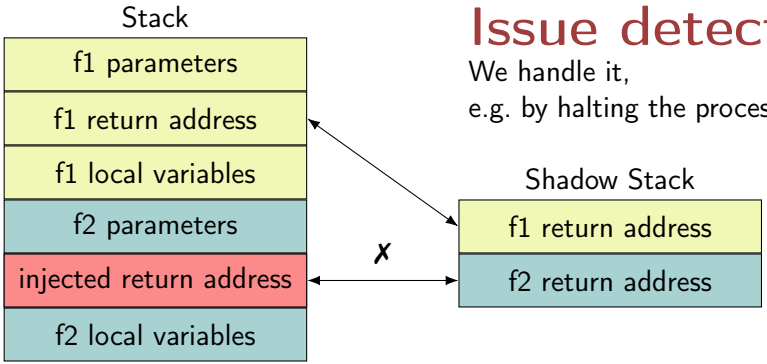
But what if they don't?

Shadow stack — principle



But what if they don't?

Shadow stack — principle



Issue detected!

We handle it,
e.g. by halting the processor and notifying the OS.

Shadow stack — implementation

- Our shadow stack lives in a **program-invisible** secondary memory of **limited size**
- All accesses happen **implicity** on function calls/returns
- When a violation is detected, we **halt** the processor (remember, no interrupts)
- No support for context switching
- Less than **100 additional lines of code**

Shadow stack — specification

The properties we want to prove

Shadow stack — specification

The properties we want to prove

Shadow stack buffer **overflow**
$\Longrightarrow$ the processor **halts**

Shadow stack — specification

The properties we want to prove

Shadow stack buffer **overflow**
$\Longrightarrow$ the processor **halts**

Shadow stack buffer **underflow**
$\Longrightarrow$ the processor **halts**

Shadow stack — specification

The properties we want to prove

Shadow stack buffer **overflow**
$\Longrightarrow$ the processor **halts**

Shadow stack buffer **underflow**
$\Longrightarrow$ the processor **halts**

Return to a **modified return address**
$\Longrightarrow$ the processor **halts**

Shadow stack — specification

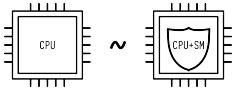
The properties we want to prove

Shadow stack buffer **overflow**
$\Longrightarrow$ the processor **halts**

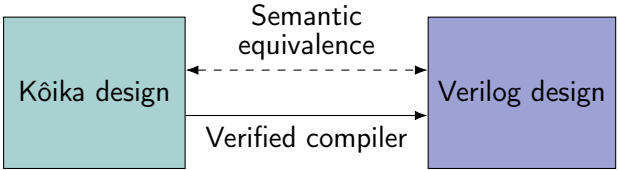
Shadow stack buffer **underflow**
$\Longrightarrow$ the processor **halts**

Return to a **modified return address**
$\Longrightarrow$ the processor **halts**

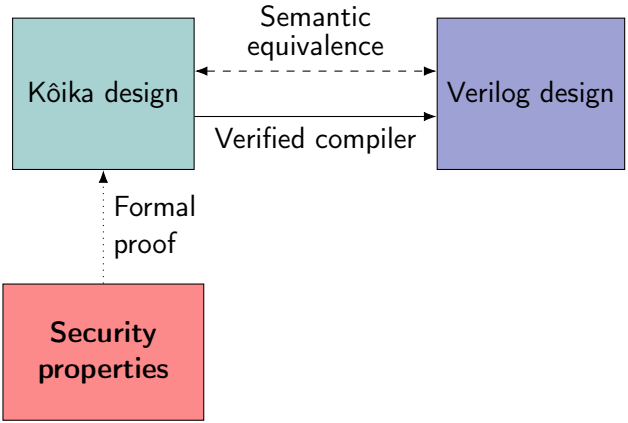
None of the above
$\Longrightarrow$ the **behavior is preserved**



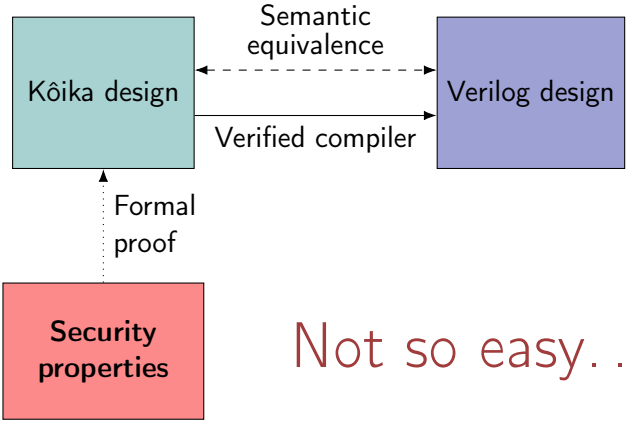
Proving properties of Kôika designs



Proving properties of Kôika designs



Proving properties of Kôika designs



Not so easy. . .

Problems with proofs on Kôika designs

Performance! For non-trivial designs, most tactics take minutes to hours and consume an absurd amount of RAM.

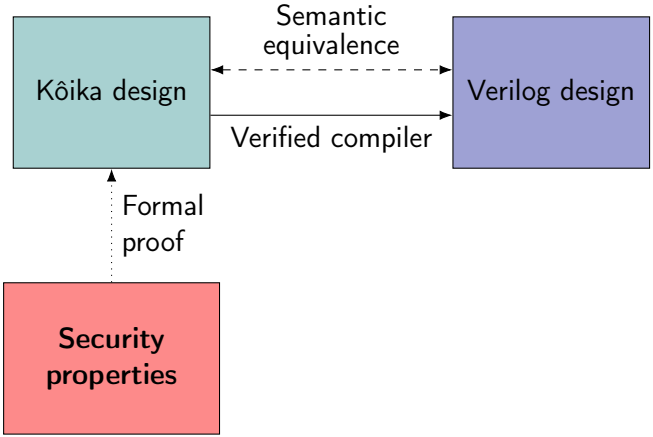
No single cause, **combination of**:

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

Outline

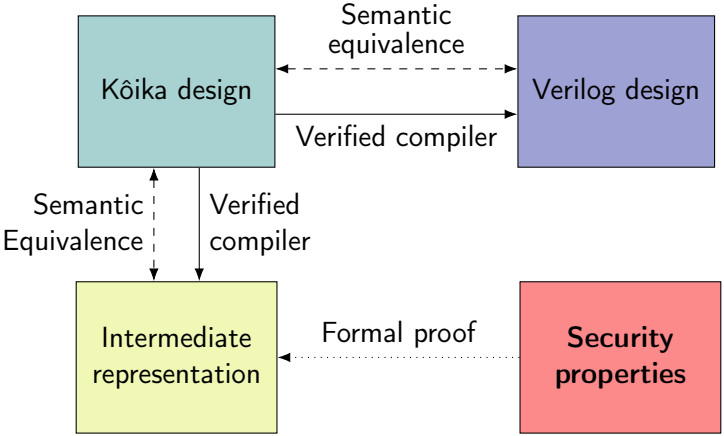
Proofs on Kôika designs

Somewhat counter-intuitively, **compiling** high-level Kôika designs into a **lower-level representation** facilitates proofs.



Proofs on Kôika designs

Somewhat counter-intuitively, **compiling** high-level Kôika designs into a **lower-level representation** facilitates proofs.



Intermediate Reasoning Representation (IRR)

IRR:

- Is a **representation of how register values are updated during a cycle**.
- Consists of:
  - A **list of variables**.
  - A **mapping from registers to the variables which describe their values** at the end of a cycle.
- **Conflicts management** is encoded **explicitly** in these expressions.

We prove the compiler from Kôika to IRR correct.

IRR isn't enough

The compiler is efficient, however it tends to produce **large sets** of deep expressions:

- Control logic is explicit
- The model is large

Furthermore, the values of the registers is usually symbolic.

Formal reasoning is still impractical...

Efficient proofs with IRR

We implement a **generic collection of verified transformations** on IRRs, akin to standard compilers passes:

- Constants propagation
- Replacement of variables with an arbitrary expression proven equivalent
- Hypothesis application
- ...

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

Example

We want to prove that when the initial value of a is 0, the final value of b is 0.

```
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].
```

	Initial
1	a
2	b
3	v_1 == 0
4	v_2 - v_2
5	1
6	2
7 (a)	if v_3 then a else v_6
8 (b)	if v_3 then v_4 else v_5

Example

We want to prove that when the initial value of a is 0, the final value of b is 0.

```
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].
```

	Initial	Prune (reg b)
1	a	
2	b	
3	v_1 == 0	
4	v_2 - v_2	
5	1	
6	2	
7 (a)	if v_3 then a else v_6	
8 (b)	if v_3 then v_4 else v_5	

Example

We want to prove that when the initial value of a is 0, the final value of b is 0.

```
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].
```

	Initial	Prune (reg b)	ExploitReg
1	a		0
2	b		
3	v_1 == 0		
4	v_2 - v_2		
5	1		
6	2		
7 (a)	if v_3 then a else v_6		
8 (b)	if v_3 then v_4 else v_5		

Example

We want to prove that when the initial value of a is 0, the final value of b is 0.

```
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].
```

	Initial	Prune (reg b)	ExploitReg	Collapse
1	a		0	
2	b			
3	v_1 == 0			0 == 0
4	v_2 - v_2			b - b
5	1			
6	2			
7 (a)	if v_3 then a else v_6			
8 (b)	if v_3 then v_4 else v_5			if v_3 then v_4 else 1

Example

We want to prove that when the initial value of a is 0, the final value of b is 0.

```
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].
```

	Initial	Prune (reg b)	ExploitReg	Collapse	Simplify
1	a		0		
2	b				
3	v_1 == 0			0 == 0	1
4	v_2 - v_2			b - b	
5	1				
6	2				
7 (a)	if v_3 then a else v_6				
8 (b)	if v_3 then v_4 else v_5			if v_3 then v_4 else 1	

Example

We want to prove that when the initial value of a is 0, the final value of b is 0.

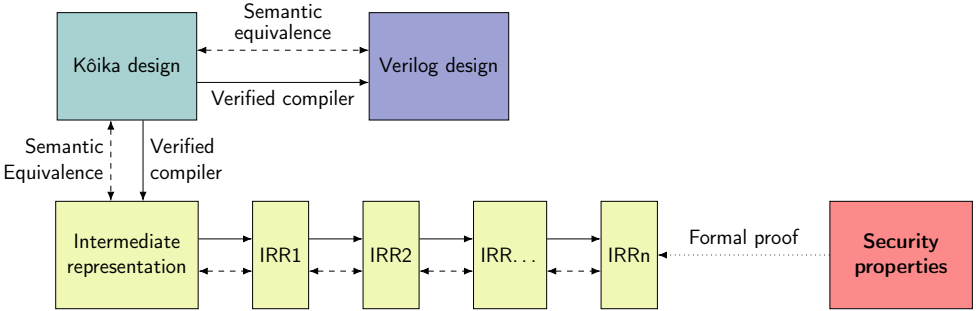
```
Registers : {a, b}.

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].
```

	Initial	Prune (reg b)	ExploitReg	Collapse	Simplify	Collapse + Simplify
1	a		0			
2	b					
3	v_1 == 0			0 == 0	1	
4	v_2 - v_2			b - b		
5	1					
6	2					
7 (a)	if v_3 then a else v_6					
8 (b)	if v_3 then v_4 else v_5			if v_3 then v_4 else 1		b - b

Conclusion



We are able to:

- Develop hardware with the Kôika HDL
- **Reason about its behavior** (in particular, we proved our security mechanism correct!)
- Simulate it
- Synthesize it (the resulting processor runs on an actual FPGA board)

Future work

- **Other security mechanisms** (e.g. memory protection, privilege levels)
- **Functional correctness** wrt. Sail semantics
- Try to **improve modularity**
- Generalization of the IRR

Thank you!