Aalto University

UNIVERSITY OF WATERLOO

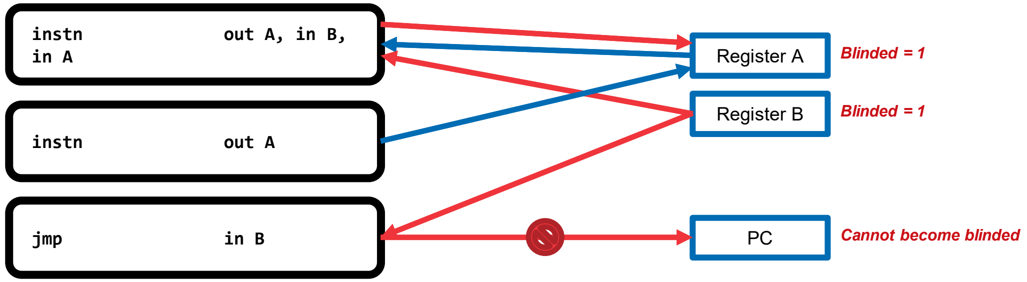
Modelling Blinded Memory with F★

Lachlan J. Gunn
https://lachlan.gunn.ee
@lachlan_gunn

(Joint work with N. Asokan, Hossam ElAtali, Hans Liljestrand)

Goals

In the previous part, we introduced the Blinded Memory extensions



How do we know the design is secure?

Solution: formal verification of the model

Basic methodology

1. **Write a function *f* simulating BliMe**
2. **Express the security property as a predicate *P*(.) on a function**
3. **Prove the assertion *P*(*f*)**

The F* language

F* (F-star) is a functional, dependent-typed language in the ML family

Dependent typing: types can depend on values

e.g. the function prototype

```
val some_function (x:int{x % 256 = 0}):
    (rv:int{rv % 2 = 0})
```

Why?

- Easy way to properties independently of implementation
- Type checker validates program correctness

Try it out yourself

Interactive editor: http://fstar-lang.org/tutorial/

Replace the code on the right with the following:

```
module Examples
open FStar.Mul

val some_function (x:int{x % 256 = 0}):
          (rv:int{rv % 2 = 0})

let some_function x = [fill this in yourself]
```

Another example

<table>
<tr><td>Reference Implementation</td><td><pre>let ref reference_cumulative_sum x =
    if x = 0
    then 0
    else x + reference_cumulative_sum (x-1)</pre></td></tr>
<tr><td>Prototype</td><td><pre>val cumulative_sum (x:int{x >= 0}):
    (rv:int{rv = reference_cumulative_sum x})</pre></td></tr>
</table>

Another example

Types are checked using an SMT solver

- Essentially, magic box that takes a theorem and outputs yes/no/maybe

Some type checks are too hard for SMT, e.g.

```
let cumulative_sum x = x*(x+1)/2
```

```
Subtyping check failed;
expected type rv: Prims.nat{rv = Examples.reference_cumulative_sum x};
got type Prims.int;
The SMT solver could not prove the query. Use --query_stats for more
details.
```

An F* example

In these cases, we can prove a lemma and invoke it in our implementation:

```
let helpful_lemma (x:nat): Lemma
      (ensures x*(x+1)/2 = reference_cumulative_sum x) =
    admit()

let cumulative_sum x =
   helpful_lemma x;
   x*(x+1)/2
```

```
Verified module: Examples
All verification conditions discharged successfully
```

Proof by hand

Theorem. Let n be a natural number. Then,

$$0 + 1 + 2 + \ldots + n = n(n+1)/2$$

Proof by hand

Theorem. Let n be a natural number. Then,

$$0 + 1 + 2 + \ldots + n = n(n+1)/2$$

Proof. We proceed by induction.

- If n = 0, then this is trivial.
- If the theorem holds for *n*-1, then

$$0 + 1 + \ldots + n-1 + n = n + (n-1)n/2 = (n+1)n/2$$

QED

Proving the lemma in F*

F* is good at reasoning about arithmetic, but needs help with induction

So, we don't need to spell out the whole proof: just the inductive part

```
let rec helpful_lemma (x:nat): Lemma
        (ensures x*(x+1)/2 = reference_cumulative_sum x) =
 if x = 0 then ()            (* Trivial to check x=0 case         *)
 else helpful_lemma (x-1)  (* Trivial to check, knowing x-1 case *)
```

The complete definition

```
let rec helpful_lemma (x:nat): Lemma
        (ensures x*(x+1)/2 = reference_cumulative_sum x) =
   if x = 0 then ()
   else helpful_lemma (x-1)

let cumulative_sum x =
   helpful_lemma x;
   x*(x+1)/2

Verified module: Examples
All verification conditions discharged successfully
```

Other things, no time to discuss

Inductive types (i.e. enums)

```
type maybeBlinded (#t:Type) =
    | Clear   : v:t -> maybeBlinded #t (* Represents a non-blinded value *)
    | Blinded : v:t -> maybeBlinded #t (* Represents a blinded value *)
```

Records (i.e. structs)

```
type foo = { a: int;
             b: int }

let add_fields (v:foo) = (a v) + (b v)
```

Typeclasses

- A generic bundle of types with associated properties

Formal verification of BliMe

We model BliMe in F* code, and prove the security of the model

Goal: changes in blinded state never affect non-blinded state

- If any two states differ only in their blinded values...
- ...after each step, the states differ only in their blinded values.

Formally

- Equivalence relation ≡, state transition $f(.)$
- Prove property Safe(≡,f): $a \equiv b \Rightarrow f(a) \equiv f(b)$

https://blinded-computation.github.io/blime-model/

Refinement of the BliMe model

We prove the correctness of BliMe by refinement

- Start with a generic state transition *f*(.)

Easy to understand

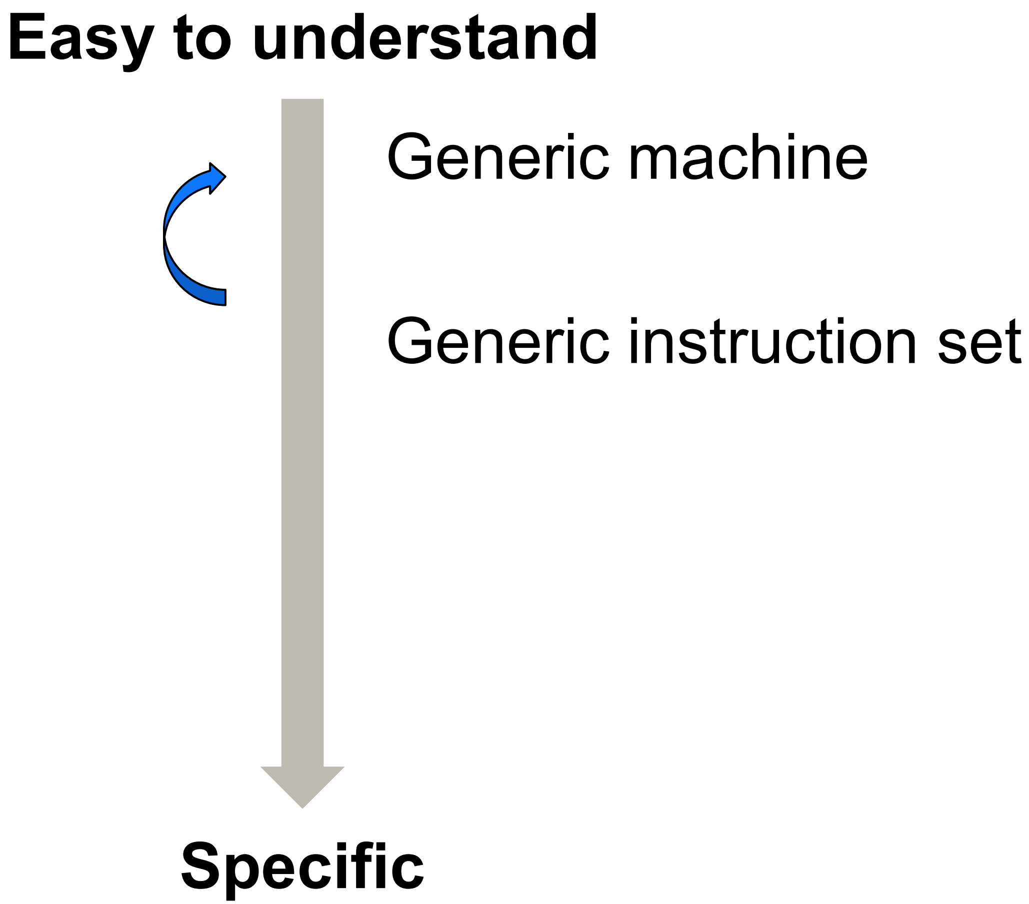
Generic machine

Specific

Refinement of the BliMe model

We prove the correctness of BliMe by refinement

- Start with a generic state transition *f*(.)
- Show that if *g*(.) is safe then *f*(.) is safe

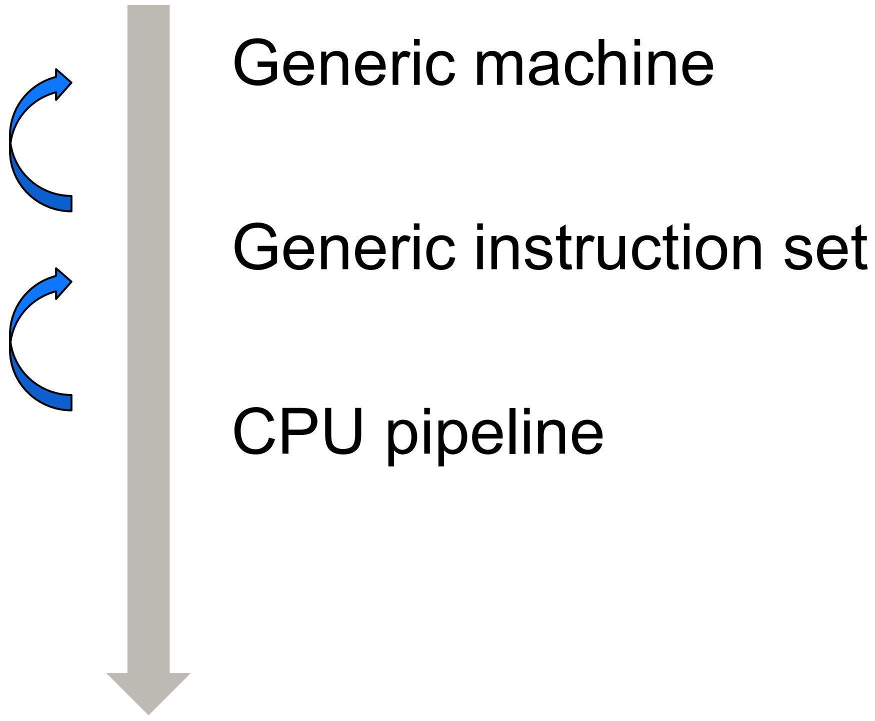


Refinement of the BliMe model

We prove the correctness of BliMe by refinement

- Start with a generic state transition *f*(.)
- Show that if *g*(.) is safe then *f*(.) is safe
- Show that if h(.) is safe then g(.) is safe

Easy to understand

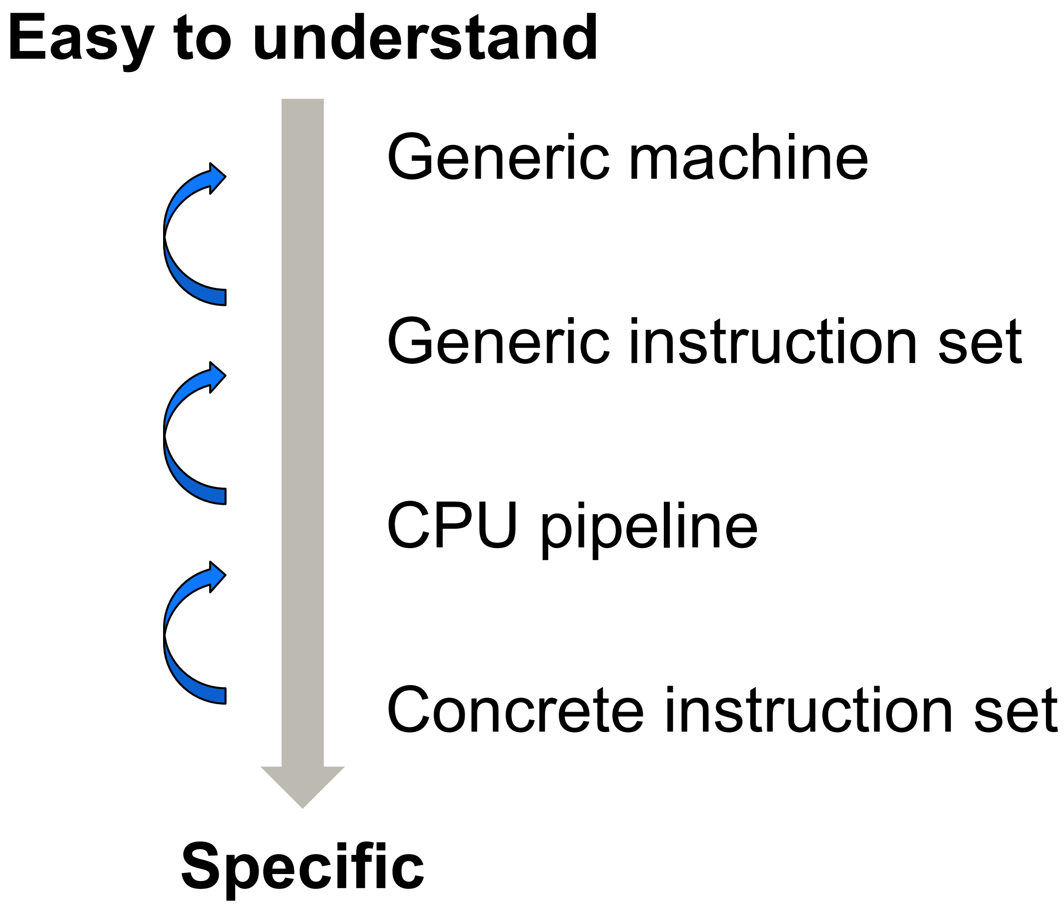


Specific

Refinement of the BliMe model

We prove the correctness of BliMe by refinement

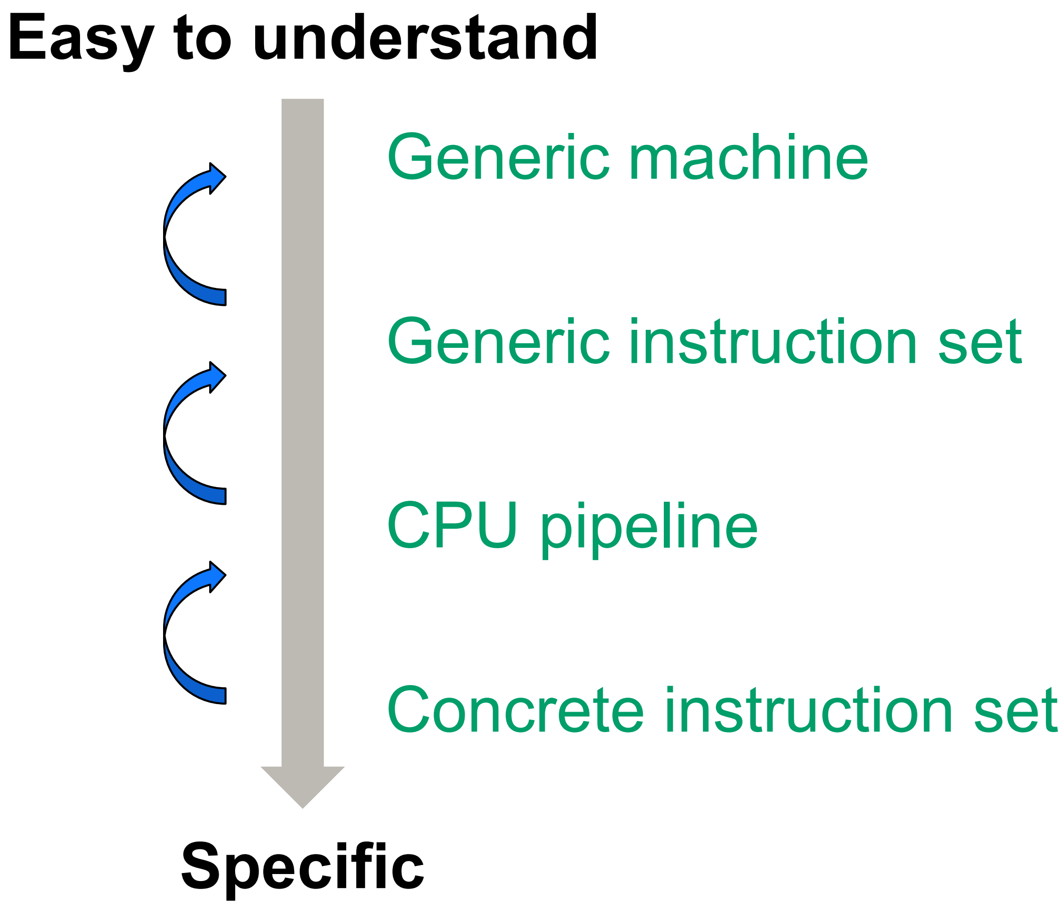
- Start with a generic state transition *f*(.)
- Show that if *g*(.) is safe then *f*(.) is safe
- Show that if h(.) is safe then g(.) is safe
- Show that if i(.) is safe then h(.) is safe



Refinement of the BliMe model

We prove the correctness of BliMe by refinement

- Start with a generic state transition *f*(.)
- Show that if *g*(.) is safe then *f*(.) is safe
- Show that if h(.) is safe then g(.) is safe
- Show that if i(.) is safe then h(.) is safe
- Show that i(.) is safe



Preliminary: Blindable data representation

Blindable state can be Clear or Blinded

- Later, blinded data has a domain tag attached to identify the client

```
type maybeBlinded (#t:Type) =
  | Clear   : v:t -> maybeBlinded #t (* Represents a non-blinded value *)
  | Blinded : v:t -> maybeBlinded #t (* Represents a blinded value *)
```

Preliminary: Blindable data representation

Blindable state can be Clear or Blinded

- Later, blinded data has a domain tag attached to identify the client

```
type maybeBlinded (#t:Type) =
   | Clear   : v:t -> maybeBlinded #t (* Represents a non-blinded value *)
   | Blinded : v:t -> maybeBlinded #t (* Represents a blinded value *)
```

We then define an equivalence class on maybeBlinded

- Equal clear values, or any pair of blinded values

```
let equiv1 lhs rhs
    = match lhs, rhs with
      | Clear x, Clear y -> (x = y)
      | Blinded _, Blinded _ -> true
      | _ -> false
```

Most generic CPU model

CPU model:

1. **Mutate** machine state

Model parameters:

- State mapping: maps machine state to machine state

Goal: Verify that this state transition is safe

Most generic CPU model

CPU model:

1. **Mutate** machine state

Model parameters:

- State mapping: maps machine state to machine state

```
let equivalent_inputs_yield_equivalent_states (exec:execution_unit) (pre1 pre2 : systemState) =
    equiv_system pre1 pre2 ⇒ equiv_system (step exec pre1) (step exec pre2)


let is_safe (exec:execution_unit) =
    ∀ (pre1 pre2 : systemState). equivalent_inputs_yield_equivalent_states exec pre1 pre2
```

Goal: Verify that this state transition is safe

CPU model

CPU model:

1. **Fetch** instruction
2. **Mutate** machine state

Model parameters:

- Execution unit: maps instruction & input values to output values

CPU model

```
type execution_unit (#n #r:memory_size) = word -> systemState #n #r -> systemState #n #r

val step (#n #r:memory_size)
    (exec:execution_unit #n #r)
    (pre_state: systemState #n #r)
    : systemState #n #r

let step exec pre_state =
    let instruction = Memory.nth pre_state.memory pre_state.pc in
        match is_blinded instruction with
        | true -> { pre_state with pc = 0uL }
        | false -> exec (unwrap instruction) pre_state
```

CPU model

```
type execution_unit (#n #r:memory_size) = word -> systemState #n #r -> systemState #n #r

val step (#n #r:memory_size)
    (exec:execution_unit #n #r)
    (pre_state: systemState #n #r)
    : systemState #n #r

let step exec pre_state =
    let instruction = Memory.nth pre_state.memory pre_state.pc in
        match is_blinded instruction with
        | true -> { pre_state with pc = 0uL }
        | false -> exec (unwrap instruction) pre_state
```

Result: Verified that state transition is safe (as in the last slide) if execution unit is safe for every instruction

CPU pipeline model

CPU model:

1. **Fetch** instruction
2. **Decode** instruction
3. **Read** input operands from machine state
4. **Compute** output values
5. **Write** output values to machine state

Model parameters:

- instruction decoder: maps instr word to opcode, lists of in/out operands
- instruction semantics: maps decoded instr & input values to output values, fault status

CPU pipeline model

CPU model:

1. **Fetch** instruction
2. **Decode** instruction
3. **Read** input operands from machine state
4. **Compute** output values
5. **Write** output values to machine state

Model parameters:

- instruction decoder: maps instr word to opcode, lists of in/out operands
- instruction semantics: maps decoded instr & input values to output values, fault status

CPU pipeline model

CPU model:

1. **Fetch** instruction
2. **Decode** instruction
3. **Read** input operands from machine state
4. **Compute** output values
5. **Write** output values to machine state

Model parameters:

- instruction decoder: maps instr word to opcode, lists of in/out operands
- instruction semantics: maps decoded instr & input values to output values, fault status

Result: Verified that this execution unit is safe (as in the last slide), if instruction semantics are safe

CPU pipeline model

What does it mean for instruction semantics to be safe?

If inputs from register file are equivalent, then result is equivalent

- **Fault status** is identical, and if there is no fault, then…
- Values written to **register file** are equivalent
- **Memory operations** have
  - Same addresses
  - Same register source/destination

```
let equiv_result (#di:decodedInstruction) (lhs rhs:(instruction_result di)) = (
       (equiv_list lhs.register_writes rhs.register_writes)
    /\ (equiv_memory_operations lhs.memory_ops rhs.memory_ops)
    /\ lhs.fault = false /\ rhs.fault = false)
  \/ (lhs.fault = true /\ rhs.fault = true)
```

ISA model

Finally, we prove safety for a concrete instruction set

We provide functions to…

- **Parse instruction** for opcode, input/output registers, immediate value
- **Instructions** store, load, conditional branch, add, subtract, multiply, AND, XOR

Too much code to cover in detail here

- Highlight: x AND 0 = Clear 0, even if x is blinded

ISA model

Finally, we prove safety for a concrete instruction set

We provide functions to…

- **Parse instruction** for opcode, input/output registers, immediate value
- **Instructions** store, load, conditional branch, add, subtract, multiply, AND, XOR

Too much code to cover in detail here

- Highlight: x AND 0 = Clear 0, even if x is blinded

Result: Verified that these instruction semantics are safe

Next steps for formal verification

Verified executable simulation

- C or OCaml code can be extracted from F*

Un-blindable registers/memory

- Currently PC is a special case

Microarchitectural side channels

- Effectiveness of enforcing rules during transient execution

Summary

F* is a useful modelling tool

Lots of useful things to prove

- BliMe’s taint propagation rule doesn’t leak information
- Model ISA implements taint propagation rule correctly
- “Special cases” like `x AND 0` are implemented correctly