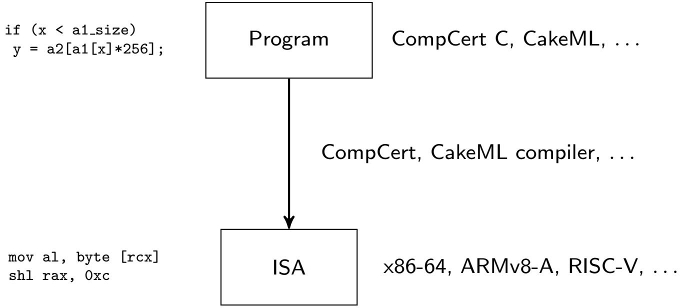
Foundations and Tools in HOL4 for Analysis of Microarchitectural Out-of-Order Execution

Karl Palmskog, Xiaomo Yao, Ning Dong,
Roberto Guanciale, and Mads Dam

KTH Royal Institute of Technology, Sweden

Instruction Set Architectures (ISAs) and Verified Programs

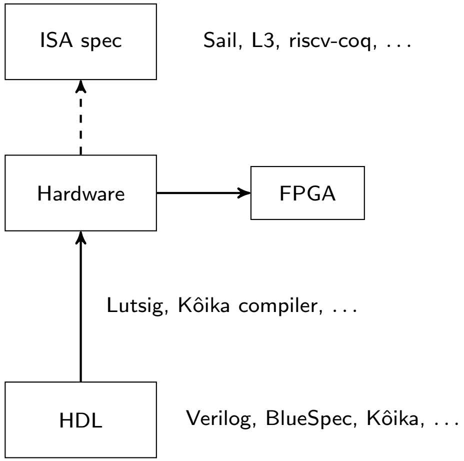

Program
CompCert C, CakeML, . . .
CompCert, CakeML compiler, . . .
ISA
x86-64, ARMv8-A, RISC-V, . . .

```
if (x < a1_size)
 y = a2[a1[x]*256];
```

```
mov al, byte [rcx]
shl rax, 0xc
```

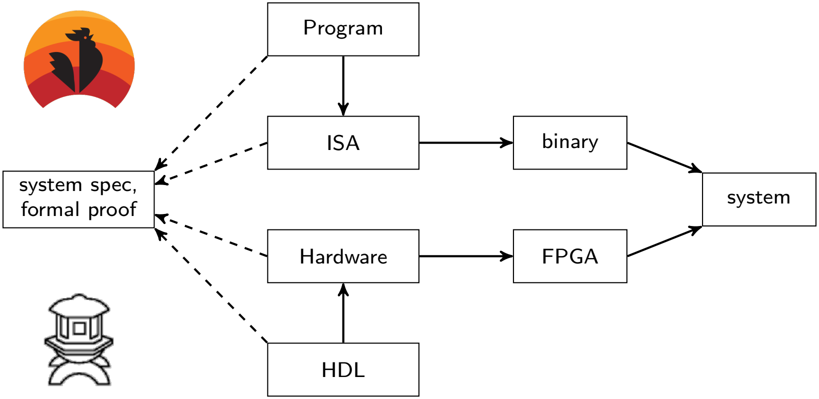
- compiler correctness composes with program correctness
- program correctness reduces to formal ISA specification

Hardware Description Languages and Hardware


ISA spec
Sail, L3, riscv-coq, . . .
Hardware
FPGA
Lutsig, Kôika compiler, . . .
HDL
Verilog, BlueSpec, Kôika, . . .

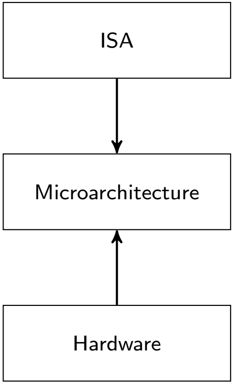
- HDL verified against ISA specification
- Compiler guarantees hardware (and FPGA) properties

End-to-End Functional System Verification



- Lightbulb system using Bluespec and RISC-V (Coq)
- Silver processor and CakeML compiler (HOL4)
- So far: only functional correctness

Missing Formal Abstraction: Microarchitectures



SPECTRE MELTDOWN FORESHADOW

- abstracts over hardware features (pipelines, cores)
- source of information flow vulnerabilities such as Spectre
- *mostly* hidden at ISA level

A Machine Independent Language (MIL)

- MIL: proposed language for describing microarchitectural program execution and reasoning about information flow
- abstracts behavior of single core, pipelined processor with out-of-order and speculative execution

	formal abstraction	implementation
ISA	BAP, BIR, ...	ARMv8-A
Microarchitecture	MIL	Cortex-A53
Hardware	HOL circuits, Bluespec	SC2A11 SoC

InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis.
R. Guanciale, M. Balliu, M. Dam. CCS ’20.

Our Contributions

Language Formalization

- deep embedding of MIL in HOL4 theorem prover
- in-order (IO) and out-of-order (OoO) dynamic MIL semantics

Our Contributions

Language Formalization

- deep embedding of MIL in HOL4 theorem prover
- in-order (IO) and out-of-order (OoO) dynamic MIL semantics

Formal Metatheory

- proof of memory consistency between IO and OoO semantics
- definition of conditional noninterference for MIL programs
- rules out side channels from OoO execution compared to IO

Our Contributions

Language Formalization

- deep embedding of MIL in HOL4 theorem prover
- in-order (IO) and out-of-order (OoO) dynamic MIL semantics

Formal Metatheory

- proof of memory consistency between IO and OoO semantics
- definition of conditional noninterference for MIL programs
- rules out side channels from OoO execution compared to IO

Tools

- execution of MIL programs inside HOL4 and via CakeML
- strategy for proving conditional noninterference

MIL Example Program: Conditional Memory Store

Parameters: `r1`, `r2`, `z` (registers), `b1`, `b2` (memory addresses)

```
tc00 := 0; tc01 := r1; tc02 := r2; // name <- value
tc03 := z; tc04 := b1; tc05 := b2;
```

MIL Example Program: Conditional Memory Store

Parameters: `r1`, `r2`, `z` (registers), `b1`, `b2` (memory addresses)

```
tc00 := 0; tc01 := r1; tc02 := r2; // name <- value
tc03 := z; tc04 := b1; tc05 := b2;
tc11 := load(MEM,tc04);            // r1 := *b1
tc12 := store(REG,tc01,tc11);
```

MIL Example Program: Conditional Memory Store

Parameters: `r1`, `r2`, `z` (registers), `b1`, `b2` (memory addresses)

```
tc00 := 0; tc01 := r1; tc02 := r2; // name <- value
tc03 := z; tc04 := b1; tc05 := b2;

tc11 := load(MEM,tc04);            // r1 := *b1
tc12 := store(REG,tc01,tc11);

tc21 := load(REG,tc03);            // cmov z,r2,r1
tc22 := tc21 == 1 ? load(REG,tc01);
tc23 := tc21 == 1 ? store(REG,tc02,tc22);
```

MIL Example Program: Conditional Memory Store

Parameters: `r1`, `r2`, `z` (registers), `b1`, `b2` (memory addresses)

```
tc00 := 0; tc01 := r1; tc02 := r2; // name <- value
tc03 := z; tc04 := b1; tc05 := b2;
tc11 := load(MEM,tc04);            // r1 := *b1
tc12 := store(REG,tc01,tc11);
tc21 := load(REG,tc03);            // cmov z,r2,r1
tc22 := tc21 == 1 ? load(REG,tc01);
tc23 := tc21 == 1 ? store(REG,tc02,tc22);
tc31 := load(REG,tc02);            // *b2 := r2
tc32 := store(MEM,tc05,tc31);
```

MIL Example Program: Conditional Memory Store

Parameters: `r1`, `r2`, `z` (registers), `b1`, `b2` (memory addresses)

```
tc00 := 0; tc01 := r1; tc02 := r2; // name <- value
tc03 := z; tc04 := b1; tc05 := b2;

tc11 := load(MEM,tc04);             // r1 := *b1
tc12 := store(REG,tc01,tc11);

tc21 := load(REG,tc03);             // cmov z,r2,r1
tc22 := tc21 == 1 ? load(REG,tc01);
tc23 := tc21 == 1 ? store(REG,tc02,tc22);

tc31 := load(REG,tc02);             // *b2 := r2
tc32 := store(MEM,tc05,tc31);

tc41 := load(PC,tc00);              // pc := pc + 4
tc42 := tc41 + 4;
tc43 := store(PC,tc00,tc42);
```

MIL States, Executions, and Traces

- instructions (guardedly) assign operations to names
- runtime states hold program, name/value map, tracker sets
- during execution, names mapped to values
- traces describe interaction with memory subsystem

$$\begin{array}{rrcl}
\textit{name sets} & C, F & ::= & \{t_1, t_2, \ldots\} \\
\textit{instructions} & \iota & ::= & t \leftarrow c?o \\
\textit{program} & I & ::= & \{\iota_1, \iota_2, \ldots\} \\
\textit{name/value map} & s & ::= & [t_1 \mapsto v_1, t_2 \mapsto v_2, \ldots] \\
\textit{runtime state} & \sigma & ::= & (I, s, C, F) \\
\textit{observations} & obs & ::= & \epsilon \mid dl\ a \mid ds\ a \mid il\ a
\end{array}$$

MIL Program Traces (in order)

```
tc00 := 0; tc01 := r1; tc02 := r2; // name <- value
tc03 := z; tc04 := b1; tc05 := b2;
tc11 := load(MEM,tc04);             // r1 := *b1
tc12 := store(REG,tc01,tc11);
tc21 := load(REG,tc03);             // cmov z,r2,r1
tc22 := tc21 == 1 ? load(REG,tc01);
tc23 := tc21 == 1 ? store(REG,tc02,tc22);
tc31 := load(REG,tc02);             // *b2 := r2
tc32 := store(MEM,tc05,tc31);
tc41 := load(PC,tc00);              // pc := pc + 4
tc42 := tc41 + 4;
tc43 := store(PC,tc00,tc42);
```

MIL Program Traces (in order)

```
tc00 := 0; tc01 := r1; tc02 := r2; // name <- value
tc03 := z; tc04 := b1; tc05 := b2;
tc11 := load(MEM,tc04);             // r1 := *b1
tc12 := store(REG,tc01,tc11);
tc21 := load(REG,tc03);             // cmov z,r2,r1
tc22 := tc21 == 1 ? load(REG,tc01);
tc23 := tc21 == 1 ? store(REG,tc02,tc22);
tc31 := load(REG,tc02);             // *b2 := r2
tc32 := store(MEM,tc05,tc31);
tc41 := load(PC,tc00);              // pc := pc + 4
tc42 := tc41 + 4;
tc43 := store(PC,tc00,tc42);
```

$$[\ dl\ b_1,\ ds\ b_2,\ il\,(pc_0 + 4)\ ]$$

MIL Program Traces (out-of-order)

```
tc00 := 0; tc01 := r1; tc02 := r2; // name <- value
tc03 := z; tc04 := b1; tc05 := b2;
tc11 := load(MEM,tc04);             // r1 := *b1
tc12 := store(REG,tc01,tc11);
tc21 := load(REG,tc03);             // cmov z,r2,r1
tc22 := tc21 == 1 ? load(REG,tc01);
tc23 := tc21 == 1 ? store(REG,tc02,tc22);
tc31 := load(REG,tc02);             // *b2 := r2
tc32 := store(MEM,tc05,tc31);
tc41 := load(PC,tc00);              // pc := pc + 4
tc42 := tc41 + 4;
tc43 := store(PC,tc00,tc42);
```

$$[\ \mathit{dl}\ b_1,\ \mathit{il}\ (pc_0 + 4),\ \mathit{ds}\ b_2\ ]$$

MIL Program Traces (out-of-order)

```
tc00 := 0; tc01 := r1; tc02 := r2; // name <- value
tc03 := z; tc04 := b1; tc05 := b2;
tc11 := load(MEM,tc04);            // r1 := *b1
tc12 := store(REG,tc01,tc11);
tc21 := load(REG,tc03);            // cmov z,r2,r1
tc22 := tc21 == 1 ? load(REG,tc01);
tc23 := tc21 == 1 ? store(REG,tc02,tc22);
tc31 := load(REG,tc02);            // *b2 := r2
tc32 := store(MEM,tc05,tc31);
tc41 := load(PC,tc00);             // pc := pc + 4
tc42 := tc41 + 4;
tc43 := store(PC,tc00,tc42);
```

$$[\ il\,(pc_0 + 4),\ dl\ b_1,\ ds\ b_2\ ]$$

MIL Program Traces (out-of-order, $z \neq 1$)

```
tc00 := 0; tc01 := r1; tc02 := r2; // name <- value
tc03 := z; tc04 := b1; tc05 := b2;
tc11 := load(MEM,tc04);            // r1 := *b1
tc12 := store(REG,tc01,tc11);
tc21 := load(REG,tc03);            // cmov z,r2,r1
tc22 := tc21 == 1 ? load(REG,tc01);
tc23 := tc21 == 1 ? store(REG,tc02,tc22);
tc31 := load(REG,tc02);            // *b2 := r2
tc32 := store(MEM,tc05,tc31);
tc41 := load(PC,tc00);             // pc := pc + 4
tc42 := tc41 + 4;
tc43 := store(PC,tc00,tc42);
```

$$[\ ds\ b_2,\ dl\ b_1,\ il\ (pc_0 + 4)\ ]$$

HOL4 Formalization Approach

- MIL abstract syntax as inductive datatypes
  - instruction names $\rightarrow$ `num`
  - values/addresses $\rightarrow$ `word64`
  - no need for explicit name binders
- OoO and IO dynamic semantics as HOL4 relations
  - rules for instruction execution/commit/fetch in OoO semantics
  - single rule for IO semantics (ordered OoO steps)
  - rules have labels with name, action, observation
- executions π are nonempty lists of state-label-state triplets
- traces are lists of (non-ϵ) observations, e.g., *trace*(π)

Store-to-load dependencies: Store May

str-may(σ, t): for load instruction t and state σ, all store instructions before t to same resource that $\underline{\text{may}}$, by further execution, assign to the load address of t.

Store-to-load dependencies: Store May

str-may(σ, t): for load instruction t and state σ, all store instructions before t to same resource that may, by further execution, assign to the load address of t.

Example: *str-may*(σ, t_{c31})

```
tc00 := 0; tc01 := r1; tc02 := r2; // name <- value
tc03 := z; tc04 := b1; tc05 := b2;
tc11 := load(MEM,tc04);             // r1 := *b1
tc12 := store(REG,tc01,tc11);
tc21 := load(REG,tc03);             // cmov z,r2,r1
tc22 := tc21 == 1 ? load(REG,tc01);
tc23 := tc21 == 1 ? store(REG,tc02,tc22);
tc31 := load(REG,tc02);             // *b2 := r2
```

MIL Dynamic Semantics

$$\frac{\begin{array}{c} t \leftarrow c?o \in I \qquad s(t)\uparrow \qquad [c]s \\ [t \leftarrow c?o](I, s, C, F) = (v, obs) \end{array}}{(I, s, C, F) \xrightarrow{(obs,\, \textsc{Exe},\, t)} (I, s + [t \mapsto v], C, F)} \ \textsc{OoO-Exe}$$

MIL Dynamic Semantics

$$\frac{\begin{array}{c} t \leftarrow c?o \in I \qquad s(t)\uparrow \qquad [c]s \\ [t \leftarrow c?o](I, s, C, F) = (v, obs) \end{array}}{(I, s, C, F) \xrightarrow{(obs,\, \textsc{Exe},\, t)}\!\!\!\!\!\rightarrow (I, s + [t \mapsto v], C, F)} \ \textsc{OoO-Exe}$$

$$\frac{\begin{array}{c} t \leftarrow c?st\, \mathcal{M}\ t_1\ t_2 \in I \qquad t \notin C \\ s(t)\downarrow \qquad s(t_1) = a \qquad s(t_2) = v \\ bn\,(str\text{-}may\,((I, s, C, F), t)) \subseteq C \end{array}}{(I, s, C, F) \xrightarrow{(ds\ a,\, \textsc{Cmt}(a, v),\, t)}\!\!\!\!\!\rightarrow (I, s, C \cup \{t\}, F)} \ \textsc{OoO-Cmt}$$

MIL Dynamic Semantics

$$\dfrac{\begin{array}{c} t \leftarrow c?st\,\mathcal{PC}\ t_1\ t_2 \in I \qquad t \notin F \qquad s(t) = a \\ translate\,(a, max\,(bn\,(I))) = I' \\ bn\,(str\text{-}may\,((I, s, C, F), t)) \subseteq F \end{array}}{(I, s, C, F) \xrightarrow{(il\ a, \textsc{Ftc}(I'), t)} (I \cup I', s, C, F \cup \{t\})}\ \textsc{OoO-Ftc}$$

MIL Dynamic Semantics

$$\frac{\begin{array}{c} t \leftarrow c?st\,\mathcal{PC}\ t_1\ t_2 \in I \qquad t \notin F \qquad s(t) = a \\ translate\,(a, max\,(bn\,(I))) = I' \\ bn\,(str\text{-}may((I, s, C, F), t)) \subseteq F \end{array}}{(I, s, C, F) \xrightarrow{(il\ a, \textsc{Ftc}(I'), t)}\!\!\!\!\rightarrow (I \cup I', s, C, F \cup \{t\})}\ \textsc{OoO-Ftc}$$

$$\frac{\begin{array}{c} \sigma \xrightarrow{(obs, \alpha, t)}\!\!\!\!\rightarrow \sigma' \\ \forall\, \iota \in \sigma.\ \text{if}\ bn\,(\iota) < t\ \text{then}\ \mathcal{C}\,(\sigma, \iota) \end{array}}{\sigma \xrightarrow{(obs, \alpha, t)} \sigma'}\ \textsc{IO-Step}$$

Well-Formed and Resource Initialized MIL States

MIL runtime states can be malformed in several ways:

- dangling instruction names: $t \leftarrow c?st\,\mathcal{PC}\;t_1\;t_{\text{undef}}$
- not respecting name order: $t_{11} \leftarrow t_{55}?o$
- loads of uninitialized resources: $t \leftarrow c?ld\,\mathcal{R}\;t_{\text{uninit}}$

Well-Formed and Resource Initialized MIL States

MIL runtime states can be malformed in several ways:

- dangling instruction names: $t \leftarrow c?st\,\mathcal{PC}\;t_1\;t_{\text{undef}}$
- not respecting name order: $t_{11} \leftarrow t_{55}?o$
- loads of uninitialized resources: $t \leftarrow c?ld\,\mathcal{R}\;t_{\text{uninit}}$

Our solution:

- state <u>well-formedness</u> conditions preserved by semantics
- state <u>resource initialization</u>

Memory Consistency

Theorem

For all well-formed and resource initialized states σ_1 *and OoO executions* $\pi = \sigma_1 \xrightarrow{l_1}\!\!\!\!\twoheadrightarrow \sigma_2 \cdots$, *there exists an IO execution* $\pi' = \sigma_1 \xrightarrow{l_1'} \sigma_2' \cdots$ *such that for all (address) values a, the list of commits for a in* π *is a prefix of the list of commits for a in* π'.

Memory Consistency

Theorem

For all well-formed and resource initialized states σ_1 and OoO executions $\pi = \sigma_1 \xrightarrow{l_1}\!\!\!\!\rightarrow \sigma_2 \cdots$, there exists an IO execution $\pi' = \sigma_1 \xrightarrow{l_1'} \sigma_2' \cdots$ such that for all (address) values a, the list of commits for a in π is a prefix of the list of commits for a in π'.

```
Theorem OoO_IO_well_formed_memory_consistent:
 !pi. well_formed_initialized_state (FST (HD pi)) ==>
  step_execution out_of_order_step pi ==>
  ?pi'. step_execution in_order_step pi' /\
   FST (HD pi') = FST (HD pi) /\
   !a. commits pi a <<= commits pi' a
```

Memory Consistency

Theorem

For all well-formed and resource initialized states σ_1 and OoO executions $\pi = \sigma_1 \xrightarrow{l_1}\!\!\!\rightarrow \sigma_2 \cdots$, there exists an IO execution $\pi' = \sigma_1 \xrightarrow{l'_1} \sigma'_2 \cdots$ such that for all (address) values a, the list of commits for a in π is a prefix of the list of commits for a in π'.

```
Theorem OoO_IO_well_formed_memory_consistent:
 !pi. well_formed_initialized_state (FST (HD pi)) ==>
  step_execution out_of_order_step pi ==>
  ?pi'. step_execution in_order_step pi' /\
   FST (HD pi') = FST (HD pi) /\
   !a. commits pi a <<= commits pi' a
```

Memory Consistency

Theorem

For all well-formed and resource initialized states σ_1 and OoO executions $\pi = \sigma_1 \xrightarrow{l_1}\!\!\!\!\rightarrow \sigma_2 \cdots$, there exists an IO execution $\pi' = \sigma_1 \xrightarrow{l'_1} \sigma'_2 \cdots$ such that for all (address) values a, the list of commits for a in π is a prefix of the list of commits for a in π'.

```
Theorem OoO_IO_well_formed_memory_consistent:
 !pi. well_formed_initialized_state (FST (HD pi)) ==>
  step_execution out_of_order_step pi ==>
  ?pi'. step_execution in_order_step pi' /\
   FST (HD pi') = FST (HD pi) /\
   !a. commits pi a <<= commits pi' a
```

Proof Intuition: Two-Step Reordering Lemma

$$\begin{tikzcd}
 & & \sigma_{k+1} \arrow[dr, "l'"] & \\
\sigma_1 \arrow[r, dashed] & \sigma_k \arrow[ur, "l"'] \arrow[dr, "l''"] & & \sigma_{k+2} \\
 & & \sigma'_{k+1} \arrow[ur, "l"] &
\end{tikzcd}$$

- name in l' is less than name in l
- l' and l'' have the same commits

Proof Intuition: Extending to Traces

$$\begin{tikzcd}
 & & \sigma_{k+1} \arrow[dr, "l'"] & & \\
\sigma_1 \arrow[r, dashed, "\hat{\pi}_1"'] \arrow[rrr, dashed, bend left=50, "\pi"] \arrow[rrrr, dashed, bend right=50, "\hat{\pi}_c"'] & \sigma_k \arrow[ur, "l"] \arrow[dr, "l''"] & & \sigma_{k+2} \arrow[r, dashed, "\pi_c"] & \sigma_c \\
 & & \sigma'_{k+1} \arrow[ur, "l"'] & &
\end{tikzcd}$$

Confidentiality: Conditional Noninterference

- IO execution is a *reference execution*
- OoO execution should not leak more than its reference

Confidentiality: Conditional Noninterference

- IO execution is a *reference execution*
- OoO execution should not leak more than its reference

Definition

States σ_1 and σ_2 are *trace-indistinguishable* for a labeled state transition relation T, written $\sigma_1 \simeq_T \sigma_2$, if for every T-execution π_1 starting in σ_1, there exists a T-execution π_2 starting in σ_2 such that $\mathit{trace}(\pi_1) = \mathit{trace}(\pi_2)$, and $\simeq_T$ is symmetric.

Confidentiality: Conditional Noninterference

- IO execution is a *reference execution*
- OoO execution should not leak more than its reference

Definition

States σ_1 and σ_2 are *trace-indistinguishable* for a labeled state transition relation T, written $\sigma_1 \simeq_T \sigma_2$, if for every T-execution π_1 starting in σ_1, there exists a T-execution π_2 starting in σ_2 such that $\mathit{trace}(\pi_1) = \mathit{trace}(\pi_2)$, and $\simeq_T$ is symmetric.

Definition

A system is *conditionally noninterferent* with respect to the security policy $\sim_\ell$, written $\mathit{CNI}(\sim_\ell)$, if it holds that $\sim_\ell \cap \simeq_{\mathsf{IO}} \subseteq \simeq_{\mathsf{OoO}}$.

Violating Conditional Noninterference

```
tc00 := 0; tc01 := r1; tc02 := r2; // name <- value
tc03 := z; tc04 := b1; tc05 := b2;
tc11 := load(MEM,tc04);            // r1 := *b1
tc12 := store(REG,tc01,tc11);
tc21 := load(REG,tc03);            // cmov z,r2,r1
tc22 := tc21 == 1 ? load(REG,tc01);
tc23 := tc21 == 1 ? store(REG,tc02,tc22);
tc31 := load(REG,tc02);            // *b2 := r2
tc32 := store(MEM,tc05,tc31);
tc41 := load(PC,tc00);             // pc := pc + 4
tc42 := tc41 + 4;
tc43 := store(PC,tc00,tc42);
```

Violating Conditional Noninterference

```
tc00 := 0; tc01 := r1; tc02 := r2; // name <- value
tc03 := z; tc04 := b1; tc05 := b2;
tc11 := load(MEM,tc04);            // r1 := *b1
tc12 := store(REG,tc01,tc11);
tc21 := load(REG,tc03);            // cmov z,r2,r1
tc22 := tc21 == 1 ? load(REG,tc01);
tc23 := tc21 == 1 ? store(REG,tc02,tc22);
tc31 := load(REG,tc02);            // *b2 := r2
tc32 := store(MEM,tc05,tc31);
tc41 := load(PC,tc00);             // pc := pc + 4
tc42 := tc41 + 4;
tc43 := store(PC,tc00,tc42);
```

- Assume value of z is classified by the security policy
- IO trace: $[dl\ b_1, ds\ b_2, il\ (pc_0 + 4)]$
- OoO trace if z $= 1$: $[dl\ b_1, ds\ b_2, il\ (pc_0 + 4)]$, $[dl\ b_1, il\ (pc_0 + 4), ds\ b_2]$, or $[il\ (pc_0 + 4), dl\ b_1, ds\ b_2]$.
- OoO trace if z $\neq 1$: $[ds\ b_2, dl\ b_1, il\ (pc_0 + 4)]$

Proving Conditional Noninterference

Lemma

If there exists:

1. *a relation* **L** *such that* $\sim_\ell \cap \simeq_{\mathsf{IO}} \subseteq \mathbf{L}$, *that is,* **L** *underapproximates information leakage for IO execution,*
2. *a bisimulation* **R** *for the OoO semantics, that is,* **R** *overapproximates program information leakage for OoO execution, and*
3. $\sim_\ell \cap \mathbf{L} \subseteq \mathbf{R}$, *that is initial attacker knowledge and IO information leakage "are not less than" the OoO leakage,*

then $CNI(\sim_\ell)$.

Automating the Conditional Noninterference Lemma Steps

1. finding a relation $\mathbf{L}$ s.t. $\sim_\ell \cap \simeq_{\mathsf{IO}} \subseteq \mathbf{L}$: mostly automatic by executing by the MIL semantics and using self composition
2. finding a bisimulation $\mathbf{R}$: currently the least automated, heuristics and examples in paper
3. $\sim_\ell \cap \mathbf{L} \subseteq \mathbf{R}$: mostly automatic

Proof Engineering

- two-step reordering lemma the most difficult to prove
- unverified translation from BIR to MIL useful for testing
- CakeML execution orders of magnitude faster than inside HOL4
- 30,000 lines of code, 20+ person months

Proof Engineering

- two-step reordering lemma the most difficult to prove
- unverified translation from BIR to MIL useful for testing
- CakeML execution orders of magnitude faster than inside HOL4
- 30,000 lines of code, 20+ person months

Longer-term vision:

- tool workflow based on HolBA, BIR, and MIL
- input: ARMv8-A and RISCV binary programs
- automatically prove absence of OoO/speculation side channels
- validated hardware model (via Scam-V, Revizor)

Conclusion

- MIL formalization corrects and validates CCS '20 paper
- MIL HOL4 proofs, CakeML code, BIR translation on Zenodo
- MIL can be integrated with other tools/workflows/ITPs
  - memory consistency "for free"
  - workflow for proving conditional noninterference