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, Stockholm, Sweden
Email: {palmskog, xiaomoy, dongn, robertog, mfd}@kth.se

***Abstract*—Program analyses based on Instruction Set Architecture (ISA) abstractions can be circumvented using microarchitectural vulnerabilities, permitting unwanted program information flows even when proven ISA-level properties ostensibly rule them out. However, the low abstraction levels found below ISAs, e.g., in microarchitectures defined in hardware description languages, may obscure information flow and hinder analysis tool development. We present a machine-checked formalization in the HOL4 theorem prover of a language, MIL, that abstractly describes microarchitectural in-order and out-of-order program execution and enables reasoning about low-level program information flows. In particular, MIL programs can exhibit information flow side channels when executed out-of-order, as compared to a reference in-order execution. We prove memory consistency between MIL's out-of-order and in-order dynamic semantics in HOL4, and define a notion of conditional noninterference for MIL programs which rules out trace-driven cache side channels. We then demonstrate how to establish conditional noninterference for programs via a novel semi-automated bisimulation based verification strategy inside HOL4 that we apply to several examples. Based on our results, we believe MIL is suitable as a translation target for ISA code to enable information flow analyses.**

***Index Terms*—information flow, interactive theorem proving, HOL4, microarchitectures, out-of-order execution**

I. Introduction

Vulnerabilities such as Spectre, Meltdown, and Foreshadow [1]–[3] demonstrate that program analyses based on Instruction Set Architecture (ISA) abstractions cannot guarantee important program properties such as freedom from unwanted information flows. Consequently, microarchitectures (residing below the ISA level) are important to understand and take into account by developers of compilers and program analysis tools. However, the low abstraction level of most hardware description languages (HDLs) obscures important microarchitectural features such as out-of-order execution of program instructions. In particular, HDLs complicate reasoning about *low-level program information flows*.

To address this problem, Guanciale et al. [4] proposed the Machine Independent Language (MIL), which abstractly describes microarchitectures and permits analysis of information flows between microinstructions. In this paper, we present a deep embedding of MIL and an encoding of its out-of-order (OoO) and in-order (IO) dynamic semantics in the HOL4 theorem prover [5]. Using our embedding, we formalize two key aspects of the metatheory of MIL. Firstly, we provide, to our knowledge, the first general machine-checked proof of *memory consistency* between in-order and out-of-order execution of microinstructions. Secondly, we define a notion of *conditional noninterference* (CNI) capturing trace-driven cache based information flow [6]. To achieve this, we clarify the assumptions under which MIL programs (1) do not *go wrong* during runtime, and (2) *progress* as expected, which was previously left implicit.

We show that out-of-order execution can introduce information side channels, by exhibiting a violation of conditional noninterference. We then devise a semi-automated bisimulation based strategy to verify conditional noninterference, which we apply to several example MIL programs. To improve automation of conditional noninterference proofs, we developed functions and results for verified execution of MIL instructions inside HOL4 [7]. We also refined our functions to CakeML code [8], which, when compiled to native code, can execute instructions orders of magnitude faster than HOL4 and demonstrate side channels for concrete MIL programs.

In order to make our theory and tools applicable to a range of real-world ISAs such as ARMv8-A and RISC-V, we developed a translator from BIR, an architecture independent binary code representation from the HolBA binary analysis framework [9] that has proof-producing lifters. To validate the MIL formalization, we analyzed both hand-crafted programs and programs translated from BIR. Based on our results, we believe MIL is ready to be used as a form of abstract microcode language, e.g., as a target language for ISA instructions to enable low-level information flow analysis. From the hardware perspective, our memory consistency proof for MIL can be reused across different formalized microarchitectures.

In summary, we make the following contributions:

- **Foundations**: We define MIL and its dynamic OoO and IO semantics in HOL4, including notions of well-formedness and resource initialization for runtime states.
- **Metatheory**: We develop formal metatheory of MIL in HOL4, including a proof of *memory consistency* and a notion of *conditional noninterference* for the semantics.
- **Tools**: We verify functions for executing MIL programs and then refine them to CakeML, yielding trustworthy MIL analysis tools both inside and outside HOL4.
- **Applications**: We devise a semi-automated reasoning strategy for conditional noninterference, which we apply to verify confidentiality of several MIL programs.

This work has been partially supported by the KTH CERCES Center and the Trustfull project funded by the Swedish Foundation for Strategic Research.

https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_19

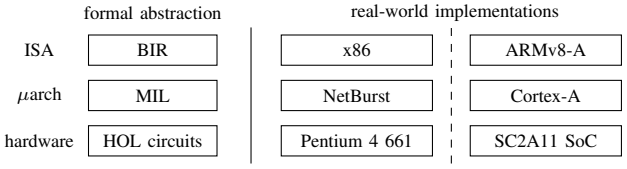

Fig. 1. Comparison of abstraction levels of BIR and MIL.

As supplementary material for the paper [10], we provide the HOL4 definitions and proofs, Standard ML code, CakeML programs, and a technical report that renders key MIL definitions and results into readable mathematical vernacular.

II. Background

A. Instruction pipelining and OoO execution

Pipelined processors divide instruction execution into *stages*, such as fetching and decoding, which are carried out by distinct processing units working independently. However, a programmer or compiler developer typically assumes instructions are processed and completed in sequential order, which may cause pipelines to stall while a processing unit is waiting for instructions to complete the previous stage. By extracting *data and address dependencies* between instructions, microarchitectures can reorder instructions, leading to better pipeline utilization and performance [11], [12].

Instruction reordering, and thus OoO execution, is a fundamental microarchitectural mechanism that can be leveraged in isolation to increase performance of pipelined processors [13]. It is also a prerequisite of *speculative* execution, where instructions are fed into a pipeline even when they are not known to be necessary to execute. Our formalization of the foundations of OoO execution is therefore an important building block towards machine-checked analysis of speculation in MIL using the speculative MIL semantics by Guanciale et al. [4].

B. HolBA and BIR

HolBA is a binary analysis platform based on HOL4 with support for ISAs such as ARMv8-A and RISC-V [9]. HolBA provides proof-producing transformations of binaries to an intermediate HOL4 representation, called BIR. That is, HolBA generates a HOL4 theorem that the BIR representation of an input binary preserves its behavior, as given by a formalization of the corresponding ISA [14]. BIR is also the target language of Scam-V, a toolchain which finds discrepancies between abstract information side-channel models and real microarchitectures [15]. Figure 1 illustrates the intended abstraction levels of BIR and MIL compared to some real-world counterparts [16], [17]. However, MIL elides many microarchitectural features not relevant to information flow.

III. Syntax and Semantics of MIL

In this section, we present the syntax of MIL and its OoO and IO dynamic semantics. The presentation largely follows Guanciale et al. [4], but we highlight key differences and additions due to the formalization in HOL4. Informally, MIL is a single static assignment (SSA) language [18], where variables in an assignment are unique microinstruction names. Ultimately, a MIL program, if it terminates successfully, computes a set of assignments of 64-bit values to such names. We assume that names are totally ordered, which induces an order on instructions via their assigned names that we call the *program order*. A program can thus be given as a linear list of guarded assignments to variables.

Example 1: We use the small parameterized MIL program below as a running example. The program compares the content of the register *reg* to 1 and sets the program counter (PC) to the memory address *adr* if this is the case, or increments the current PC value by 4 otherwise. It thus implements a high-level conditional branch on equal (`beq`) instruction.

```
tb0 := true ? 0; // zeroed name for PC load/store
tb1 := true ? reg; // get register identifier
tb2 := true ? load(REG, tb1); // load register value
tb3 := true ? tb2 == 1; // is the register value 1?
tb4 := true ? load(PC, tb0);  // load PC value
tb5 := true ? adr; // get memory address
tb6 := tb3 ? store(PC, tb0, tb5); // store to PC
tb7 := true ? tb4 + 4; // increment PC value by 4
tb8 := !tb3 ? store(PC, tb0, tb7); // store to PC
```

To obtain a fully defined ("ground") MIL program, the assignment variable names (`tbX`) must be replaced by non-negative integers, and the parameters *reg* and *adr* must be replaced by 64-bit words. We usually use variable name suffixes to indicate desired integer ordering, e.g., `tb0` $<$ `tb6`. Subsequently, we will omit `true` guards, e.g., we will write `tb0 := 0`.

A. Abstract Syntax

In Figure 2(a), we define the abstract syntax of MIL.

Names. We use unbounded HOL4 natural numbers as microinstruction names t, and predicate sets [19] for collections of names N. This approach theoretically permits infinite sets which are not meaningful in our context, but allowed easy transcription of set-related definitions from the original definition of MIL.

Values. Values v (and a) are 64-bit words encoded in the usual way for HOL [20]. The constant values *false*, *true*, and 0 are defined according to conventions of the C language. Besides finiteness and distinctness of *false* and *true*, the MIL metatheory (in contrast to the tools and examples) does not rely on anything specific about the word size.

Expressions. Expressions e (and c) are side-effect free and are assumed to include at least names and values. However, as long as requirements on the semantics of expressions (outlined in Section III-B) are met, expressions can be arbitrarily added to MIL without affecting the metatheory. In our HOL4 encoding, we defined expression syntax and semantics to match the BIR language, streamlining the translation from BIR to MIL.

Resources and operations. MIL operations are defined on a *resource* τ, which is either the PC, memory, or a register. An operation o is either an expression, or a load or store on a resource. Since there is a single PC resource, PC loads and stores are intended to take a name as first argument that is assigned to the value 0; this is implicit for Guanciale et al.

Microinstructions. A MIL microinstruction ι, or *instruction* for short, is an assignment of a name t to (the result of) an operation o, guarded by an expression c. A single higher-level instruction, e.g., at the ISA level, will typically be represented by many MIL instructions, which is why MIL is parameterized on a *translation* function explained in Section III-B.

B. Runtime States and Semantic Definitions

To provide dynamic semantics for MIL, we define runtime states for programs; Figure 2(b) lists the basic syntax we use.

Programs. MIL programs I are predicate sets of instructions. Whenever convenient, we consider instructions in I in program order (using the assigned instruction names).

Stores. Stores s are finite maps from names to values, where $dom\,(s)$ is the set of names that are mapped by s. We write $s(t)\downarrow$ ($s(t)\uparrow$) for $t \in dom\,(s)$ (resp. $t \notin dom\,(s)$).

States. In addition to a program I and store s, a MIL runtime state σ contains two sets of names C and F that respectively track whether an associated instruction has been committed to memory or its successor instructions have been fetched.

Observations. An observation obs is either the silent observation ϵ, a data load dl, a data store ds, or an instruction load il. The three latter include a memory address value.

Actions. Actions α represent transitions. Instructions are first executed (EXE). Then, if an instruction is a memory store, it can be committed (CMT), or, if it is a PC store, it can cause the next instructions to be fetched (FTC).

Labels. In contrast to Guanciale et al., *transition labels* l contain not only observations, but also the action performed by the transition and the name of the instruction for which the action was performed.

In abstract syntax, the program in Example 1, which we abbreviate $I_{\mathsf{beq}}(reg, adr)$, is written:

$$\left\{\begin{array}{l} t_{\mathsf{b0}} \leftarrow 0,\ t_{\mathsf{b1}} \leftarrow reg,\ t_{\mathsf{b2}} \leftarrow ld\,\mathcal{R}\,t_{\mathsf{b1}},\ t_{\mathsf{b3}} \leftarrow t_{\mathsf{b2}} == 1, \\ t_{\mathsf{b4}} \leftarrow ld\,\mathcal{PC}\,t_{\mathsf{b0}},\ t_{\mathsf{b5}} \leftarrow adr,\ t_{\mathsf{b6}} \leftarrow t_{\mathsf{b3}}?st\,\mathcal{PC}\,t_{\mathsf{b0}}\,t_{\mathsf{b5}}, \\ t_{\mathsf{b7}} \leftarrow t_{\mathsf{b4}} + 4,\ t_{\mathsf{b8}} \leftarrow !t_{\mathsf{b3}}?st\,\mathcal{PC}\,t_{\mathsf{b0}}\,t_{\mathsf{b7}} \end{array}\right\}$$

Executing the last instruction in I_{beq} is represented by a label $(il\,(pc_0+4), \text{FTC}\,(I), t_{\mathsf{b8}})$, where pc_0 is the original PC value and I is the translation of the program at pc_0+4.

Bound and free names. For an expression e, its set of names $n\,(e)$ is defined recursively on the structure in the obvious way. An instruction ι has a *bound name*, written $bn\,(\iota)$, and a set of *free names*, written $fn\,(\iota)$; the set of all names in ι is written $n\,(\iota)$. The set of all bound names of instructions in a program I is written $bn\,(I)$. In addition, $n\,(l)$ yields the name in the label l. For example, if $\iota = t_{\mathsf{b6}} \leftarrow t_{\mathsf{b3}}?st\,\mathcal{PC}\,t_{\mathsf{b0}}\,t_{\mathsf{b5}}$, then we have $bn\,(\iota) = t_{\mathsf{b6}}$ and $fn\,(\iota) = n(t_{\mathsf{b3}}) \cup n(st\,\mathcal{PC}\,t_{\mathsf{b0}}\,t_{\mathsf{b5}}) = \{t_{\mathsf{b0}}, t_{\mathsf{b3}}, t_{\mathsf{b5}}\}$, so $n\,(\iota) = \{t_{\mathsf{b0}}, t_{\mathsf{b3}}, t_{\mathsf{b5}}, t_{\mathsf{b6}}\}$.

Semantics of expressions. The semantics of an expression e in store s is given by a partial function returning a value v, which we write $[e]s = v$. If the function is (un-)defined, we write $[e]s\downarrow$ (resp. $[e]s\uparrow$). We do not define an explicit canonical function for the semantics of expressions, since it is microarchitecture dependent. However, in contrast to Guanciale et al., we impose requirements on such functions:

1) $[e]s\downarrow$ if and only if $n\,(e) \subseteq dom\,(s)$.
2) If $s(t) = s'(t)$ holds for all $t \in n\,(e)$, then $[e]s = [e]s'$.
3) For all v and s, $[v]s = v$.

For validation, we implemented a function consistent with BIR semantics, where for example $e + e'$ is evaluated using `word_add` from the HOL4 word theory. Given a store s, an expression c evaluates to a *true guard condition*, written $[c]s$, whenever there exists v such that $[c]s = v$ and $v \neq false$.

Address and resource of store or load. Given the name t of a store or load instruction in a program, we need to be able to obtain the resource and the name of the instruction that computes the *address* that t targets. We therefore define the partial function $addr$ so that $addr(I, t) = (\tau, t')$ if $t \leftarrow c?ld\,\tau\,t' \in I$ or $t \leftarrow c?st\,\tau\,t'\,t'' \in I$. For instance, $addr(I_{\mathsf{beq}}, t_{\mathsf{b2}}) = (\mathcal{R}, t_{\mathsf{b1}})$ for the example program.

Store may and store active. To handle store-to-load dependencies we define two auxiliary functions *str-may*(σ, t) and *str-act*(σ, t) that determine, for a given load instruction t and state σ, respectively, a) the set of stores $\iota = t' \leftarrow c'?st\,\tau\,t_1\,t_2$ that *may* by further instantiation of names smaller than t assign to the (possibly as yet unknown) load address t_0 of t, and b) the set of stores ι in *str-may*(σ, t) that cannot be eliminated due to another store $t'' : t' < t'' < t$ overwriting either the store address t_1 of t' or the load address t_0 of t. Formally:

$$\begin{array}{l} \iota \in \textit{str-may}\,(\sigma, t) \text{ iff } t' < t \wedge ([c']s \vee [c']s\uparrow) \wedge \\ \quad (s(t_1) = s(t_0) \vee\ s(t_1)\uparrow \vee s(t_0)\uparrow) \\ \\ \iota \in \textit{str-act}\,(\sigma, t) \text{ iff } \iota \in \textit{str-may}\,(\sigma, t) \wedge \\ \quad t'' \leftarrow c''?st\,\tau\,t'_1\,t'_2 \in \textit{str-may}\,(\sigma, t)\ \wedge t'' > t' \wedge \\ \quad [c'']s\ \rightarrow s(t'_1) \neq s(t_0) \wedge s(t'_1) \neq s(t_1) \end{array}$$

Example 2: The MIL program below loads the register r_1 from the memory address b_1, copies the value of r_1 into r_2 if the flag in register z is set, saves the result into the memory address b_2, and then increments the PC by 4. At a high level, the program thus implements conditional copying of memory on equal, and we refer to it as $I_{\mathsf{ceq}}(b_1, b_2)$.

```
tc00 := 0; tc01 := r1; tc02 := r2;
tc03 := z; tc04 := b1; tc05 := b2;
tc11 := load(MEM, tc04); // [1of2] r1 := *b1
tc12 := store(REG, tc01, tc11); // [2of2]
tc21 := load(REG, tc03); // [1of3] cmov z, r2, r1
tc22 := tc21 == 1 ? load(REG, tc01); // [2of3]
tc23 := tc21 == 1 ? store(REG, tc02, tc22); // [3of3]
tc31 := load(REG, tc02); // [1of2] *b2 := r2
tc32 := store(MEM, tc05, tc31); // [2of2]
tc41 := load(PC, tc00); // [1of3] pc := pc + 4
tc42 := tc41 + 4; // [2of3]
tc43 := store(PC, tc00, tc42); // [3of3]
```

We assume that $I_{\mathsf{ceq}}(b_1, b_2)$ runs after another initialization program I_0, i.e., that $\sigma = (I_0 \cup I_{\mathsf{ceq}}(b_1, b_2), s, C, F)$ and all instruction names in I_0 are before t_{c00}.

Suppose that in the state σ, we have $s(t_{\mathsf{c00}})\uparrow, \ldots, s(t_{\mathsf{c43}})\uparrow$. Then, *str-may*$\,(\sigma, t_{\mathsf{c31}})$ contains all register stores coming before t_{c31}, since the load address of t_{c31} is undefined and any previous register store instruction can potentially affect the loaded value of t_{c31}.

$$
\begin{array}{rll}
N, C, F ::= \{t_1, t_2, \ldots\} & & \text{names (\texttt{set})} \\
v, a ::= \mathit{false} \mid \mathit{true} \mid 0 \mid \ldots & & \text{value (\texttt{word64})} \\
e, c ::= v \mid t \mid !e \mid e + e' \mid \ldots & & \text{expression} \\
\tau ::= \mathcal{PC} \mid \mathcal{R} \mid \mathcal{M} & & \text{resource} \\
o ::= e \mid \mathit{ld}\ \tau\ t \mid \mathit{st}\ \tau\ t\ t' & & \text{operation} \\
\iota ::= t \leftarrow c?o & & \text{instruction}
\end{array}
$$

(a)

$$
\begin{array}{rll}
I ::= \{\iota_1, \iota_2, \ldots\} & & \text{program (\texttt{set})} \\
s ::= [t_1 \mapsto v_1, t_2 \mapsto v_2, \ldots] & & \text{store (\texttt{fmap})} \\
\sigma ::= (I, s, C, F) & & \text{state} \\
\mathit{obs} ::= \epsilon \mid \mathit{dl}\ a \mid \mathit{ds}\ a \mid \mathit{il}\ a & & \text{observation} \\
\alpha ::= \textsc{Exe} \mid \textsc{Cmt}(a, v) \mid \textsc{Ftc}(I) & & \text{action} \\
l ::= (\mathit{obs}, \alpha, t) & & \text{transition label}
\end{array}
$$

(b)

Fig. 2. MIL abstract syntax (a), and syntax used for MIL runtime state and executions (b).

Suppose σ is the state after the execution of all instructions in I_0 and the instructions on the first line, i.e., $s(t_{\mathsf{c00}}) = 0, \ldots, s(t_{\mathsf{c05}}) = b_2$. Then, *str-may* $(\sigma, t_{\mathsf{c31}})$ contains all stores in I_0 that update r_2, as well as the store t_{c23}. *str-may* $(\sigma, t_{\mathsf{c31}})$ does not contain t_{c12}, since the destination register of t_{c12} (r_1) differs from the source register of t_{c31} (r_2).

Suppose σ is the state after the execution of all instructions until t_{c22}, and let $s(t_{\mathsf{c21}}) = 0$. Then, *str-may* $(\sigma, t_{\mathsf{c31}})$ does not contain t_{c23}, since the guard condition of t_{c23} is false, and therefore the store will not be executed. Hence, *str-act* $(\sigma, t_{\mathsf{c31}})$ contains the last instruction in *str-may* $(\sigma, t_{\mathsf{c31}})$ not overwritten by a subsequent store. However, if $s(t_{\mathsf{c21}}) = 1$, then *str-may* $(\sigma, t_{\mathsf{c31}})$ contains t_{c23} and *str-act* $(\sigma, t_{\mathsf{c31}}) = \{t_{\mathsf{c23}}\}$.

Semantics of instructions. The semantics of instructions is given by a partial function taking an instruction ι and state σ and returning a value and an observation, which we write as $[\iota]\sigma = (v, \mathit{obs})$. We define the function by case analysis on ι.

- $[t \leftarrow c?e]\sigma = (v, \epsilon)$, if $[e]s = v$.
- $[t \leftarrow c?ld\ \tau\ t']\sigma = (v, \mathit{dl}\ a)$, if $\mathit{bn}\,(\textit{str-act}\,(\sigma, t)) = \{t''\}$, $s(t') = a$, $s(t'') = v$, $\tau = \mathcal{M}$, and $t'' \in C$.
- $[t \leftarrow c?ld\ \tau\ t']\sigma = (v, \epsilon)$, if $\mathit{bn}\,(\textit{str-act}\,(\sigma, t)) = \{t''\}$, $s(t') = a$, $s(t'') = v$, and either $\tau \neq \mathcal{M}$ or $t'' \notin C$.
- $[t \leftarrow c?st\ \tau\ t_1\ t_2]\sigma = (v, \epsilon)$, if $s(t_1) = v$ and $s(t_2)\downarrow$.

Completed microinstructions. To guarantee *progress* during execution of MIL programs, we provide a different criterion than Guanciale et al. for instructions to be completed. Specifically, we define ι as *completed* in a state $\sigma = (I, s, C, F)$, written $\mathcal{C}\,(\sigma, \iota)$, whenever

- $\iota = t \leftarrow c?st\ \mathcal{M}\ t_1\ t_2$ and either $[c]s = \mathit{false}$ or $t \in C$
- $\iota = t \leftarrow c?st\ \mathcal{PC}\ t_1\ t_2$ and either $[c]s = \mathit{false}$ or $t \in F$
- $\iota = t \leftarrow c?o$, and either $[c]s = \mathit{false}$ or $t \in \mathit{dom}\,(s)$.

For example, if the value of *reg* is 1 in Example 1, then after $t_{\mathsf{b3}} \leftarrow t_{\mathsf{b2}} == 1$ has been executed (mapping t_{b3} to *true*), instruction t_{b8} becomes completed, since its guard is *false*.

C. Transition Step Relations

We define two dynamic semantics of MIL in the structural operational semantics style: an OoO semantics and an IO semantics. Specifically, we define, by the rules in Figure 3, the labeled OoO transition step relation, $\sigma \xrightarrow{l}\!\!\!\!\rightarrow \sigma'$, and the labeled IO transition step relation, $\sigma \xrightarrow{l} \sigma'$.

OoO-Exe. This rule computes the value v of an instruction with bound name t and records the result in the store by adding the mapping $[t \mapsto v]$. This uses the semantics of instructions, and therefore relies on most functions above, such as *str-act*.

OoO-Ftc. This rule fetches an already-executed PC store instruction, which potentially adds more instructions to the program in the state. Intuitively, the function $\mathit{translate}(a, t)$ used in the rule looks up the code at the data area address a and generates the corresponding MIL instructions using names greater than t. Fetches thus enable MIL programs to have iterative and possibly diverging behavior.

OoO-Cmt. This rule commits an already-executed memory store instruction to memory. Both the memory address a and the new value v are part of the label's action, while only the former is included in the observation.

IO-Step. This rule processes instructions using the OoO rules, but deterministically following the program order.

For instance, in an initial state for I_{beq}, OoO-Exe transitions are enabled for the instructions for t_{b0} and t_{b1}. However, if $t_{\mathsf{b0}} < t_{\mathsf{b1}}$ as expected, only t_{b0} is enabled for an IO-Step transition, i.e., the instruction for t_{b0} must be completed before the instruction for t_{b1}.

The OoO semantics can be viewed as abstracting the behavior of a pipelined single-core microarchitecture which receives CISC-like ordered program instructions, and then translates each such instruction into one or more RISC-like microinstructions which are nondeterministically executed and (possibly) completed. For instance, the OoO semantics is reminiscent of the NetBurst microarchitecture used in Pentium 4 processors [16]. In contrast, the IO semantics is more akin to abstract ISA behavior, where execution must always proceed according to an order specified by a programmer or compiler. Silver is an example where the microarchitecture itself behaves similarly to the MIL IO semantics [17].

IV. Metatheory of MIL

While a MIL program has no canonical initial state at runtime, we define in this section a notion of state well-formedness that, intuitively, ensures program execution does not *go wrong*. However, well-formedness does not by itself guarantee *progress*, e.g., that execution will end up in a state where all instructions are completed. For progress, we define *resource-initialized* states, which prevent instruction execution from getting stuck. By comparison, the MIL semantics of Guanciale et al. [4] did not explicitly account for progress and only ruled out some forms of malformed states.

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

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

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

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

Fig. 3. OoO and IO labeled transition step relation rules.

Assuming well-formedness and resource initialization enabled us to prove in HOL4 the *memory consistency* of the OoO and IO semantics of MIL, fully elaborating the previous pen-and-paper reasoning and filling in all the gaps. We believe this puts our notion of *conditional noninterference* for MIL on firm ground. In turn, this notion allows us to reason about information flow in MIL programs, as described in Section VI.

A. Well-formedness of States

We now define the requirements for a state $\sigma = (I, s, C, F)$ to be *well formed*. Properties 1 to 8 below are the basic sanity properties that, e.g., express the absence of dangling instruction references, that instruction dependencies form a directed acyclic graph, and that instruction execution is properly recorded in the store and elsewhere. For instance, in states including I_{beq}, property 2 requires that $t_{\mathsf{b1}} < t_{\mathsf{b2}}$, and property 3 forbids having $t_{\mathsf{b0}} = t_{\mathsf{b2}}$.

1) I is a finite set such that $C \cup F \subseteq dom(s) \subseteq bn(I)$.
2) If $\iota \in I$, $t \in fn(\iota)$, then $t < bn(\iota)$ and $\exists\, \iota' \in I$ such that $bn(\iota') = t$.
3) If $\iota \in I$, $\iota' \in I$, and $bn(\iota) = bn(\iota')$, then $\iota = \iota'$.
4) If $t \in C$, then $bn(\textit{str-may}(\sigma, t)) \subseteq C$ and $\exists\, \iota \in I$ such that $\iota = t \leftarrow c?st\,\mathcal{M}\,t_1\,t_2$.
5) If $t \in F$, then $bn(\textit{str-may}(\sigma, t)) \subseteq F$ and $\exists\, \iota \in I$ such that $\iota = t \leftarrow c?st\,\mathcal{PC}\,t_1\,t_2$.
6) If $\iota \in I$ for $\iota = t \leftarrow c?st\,\mathcal{PC}\,t_1\,t_2$ or $t \leftarrow c?ld\,\mathcal{PC}\,t_1$, then $t_1 \leftarrow true?0 \in I$.
7) If $\iota \in I$ for $\iota = t \leftarrow c?st\,\tau\,t_1\,t_2$, and $s(t) = v$, then $s(t_1)\downarrow$ and $s(t_2) = v$.
8) If $t \leftarrow c?e \in I$, $s(t) = v$, then $[t \leftarrow c?e]\sigma = (v, \epsilon)$.

Properties 9 to 11 below next ensure that guards behave as expected and do not block execution. For instance, property 9 says that if t_{b6} in I_{beq} has a stored value, then t_{b3} has a value stored which is not equal to *false*.

9) If $t \leftarrow c?o \in I$ and $s(t)\downarrow$, then $[c]s$.
10) If $t \leftarrow c?o \in I$, $t' \leftarrow c'?o' \in I$ and $t' \in n(c)$, then $c' = true$.
11) If $t \leftarrow c?o \in I$, $t' \leftarrow c'?o' \in I$, $t' \in n(o)$, $[c]s'$, and $[c']s' = v'$, then $v' \neq false$.

Finally, we impose analogous properties for output from the *translate* function; motivation and details are in the supplementary material [10]. In lieu of subject reduction for an explicitly typed language, we then proved in HOL4 that well-formedness is preserved by all the OoO and IO transition rules whenever *translate* returns output with the required properties. In particular, the proof relies on that $t < t'$ whenever $\iota \in translate(v, t)$ and $t' \in n(\iota)$. From now on, we always assume that states are well formed.

B. State Resource Initialization

Consider a load instruction in a state, e.g., the instruction for t_{b2} in a state whose program includes I_{beq}. Intuitively, during an Exe transition for t_{b2}, the previous value for the register *reg* is copied to the store, which is done by finding the last completed store instruction on *reg*. However, if there is no such store instruction, t_{b2} can never be completed.

To address this problem in the MIL metatheory of Guanciale et al. [4], we introduce a notion of resource initialization for states. Specifically, we say that the predicate *initialized-resource-set*(σ, τ, V) is true precisely when, for all $v \in V$, there exists a completed store instruction for v and τ in σ such that there is no earlier load instruction in σ for v and τ. We then say that, in *resource initialized* states, *initialized-resource-set* holds for *all* possible values when $\tau = \mathcal{R}$ or $\tau = \mathcal{M}$, and for 0 when $\tau = \mathcal{PC}$.

For example, in a well-formed resource initialized state $\sigma = (I, s, C, F)$ whose program includes I_{beq}, we know there exists an instruction $t \leftarrow c?st\,\mathcal{R}\,t'\,t''$ such that $t \in dom(s)$, $s(t') = z$, and $t < t_{\mathsf{b4}}$, ensuring that we can complete the load instruction t_{b4} with an Exe transition.

C. Executions, Commits, and Traces

We define MIL *executions* formally as (bounded) lists π of state-label-state triples (σ, l, σ'). More specifically, for π to be an *OoO execution*, it must be non-empty and its triples must follow the OoO transition relation, which we write as $\pi = \sigma_1 \xrightarrow{l_1}\!\!\!\!\rightarrow \sigma_2 \xrightarrow{l_2}\!\!\!\!\rightarrow \sigma_3 \cdots$. Analogously, when π is an *IO execution*, we write $\pi = \sigma_1 \xrightarrow{l_1} \sigma_2 \xrightarrow{l_2} \sigma_3 \cdots$. We also write $\pi +\!\!+ \pi'$ for the concatenation of two executions.

For an execution π and a memory address a, the function $commits(\pi, a)$ returns a list with the history of values written (i.e., sent to the memory subsystem) in π to a. We define *commits* by case analysis on the first transition in π so that, e.g., $commits(\sigma_1 \xrightarrow{(obs,\,\textsc{Cmt}(a,v),\,t)} \sigma_2 +\!\!+ \pi', a) = v, commits(\pi', a)$. Finally, the function $trace(\pi)$ returns the trace of the execution π, which is its (possibly empty) list of non-silent observations. As one example, we have $trace(\sigma \xrightarrow{(dl\,a,\,\tau,\,t)} \sigma' +\!\!+ \pi') = dl\,a,\, trace(\pi')$.

D. Functional Correctness: Memory Consistency

Intuitively, two models of program execution are memory consistent when they yield the same sequence of memory updates for each memory address, which ensures that the final result of a program (if there is one) is the same in both models. More formally, *memory consistency* of the OoO and IO semantics requires that writes to the same memory location are always seen in the same order by an observer, which we state and prove as our main theorem.

Theorem 1: For all well-formed and resource initialized states σ_1 and OoO executions $\pi = \sigma_1 \xrightarrow{l_1} \sigma_2 \cdots$, there exists an IO execution $\pi_i = \sigma_1 \xrightarrow{l'_1} \sigma'_2 \cdots$ 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_i (and vice versa for IO and OoO execution).

Proof: The proof relies on a two-step reordering lemma which says that if $\sigma_k \xrightarrow{l} \sigma_{k+1} \xrightarrow{l'} \sigma_{k+2}$ and $n(l') < n(l)$, then there exists σ'_{k+1} and l'' such that $\sigma_k \xrightarrow{l''} \sigma'_{k+1} \xrightarrow{l} \sigma_{k+2}$, where l' and l'' have the same commits.

The key steps of the proof are illustrated in Figure 4, and can be divided into two parts. The first part establishes, by induction on execution length, that for every OoO execution there is a corresponding *ordered* OoO execution, with the same initial and final state, and the same order of commits per address, where transition labels respect the total order on names. In the following, we use $\hat{\ }$ to identify ordered OoO executions. Let $\pi = \pi_0 +\!\!+ \sigma_{k+1} \xrightarrow{l'} \sigma_{k+2}$ be an OoO execution; then by induction there is an ordered OoO execution $\hat{\pi}_0 = \hat{\pi}_1 +\!\!+ \sigma_k \xrightarrow{l} \sigma_{k+1}$ of π_0. Hence, $\pi' = \hat{\pi}_1 +\!\!+ \sigma_k \xrightarrow{l} \sigma_{k+1} \xrightarrow{l'} \sigma_{k+2}$ is also an OoO execution. If $n(l) \leq n(l')$, then π' is an ordered execution of π; otherwise, the two-step reordering lemma guarantees that there exists an OoO execution $\pi'' = \hat{\pi}_1 +\!\!+ \sigma_k \xrightarrow{l''} \sigma'_{k+1} \xrightarrow{l} \sigma_{k+2}$. We use induction again to show that there is an ordered OoO execution $\hat{\pi}_2$ of $\hat{\pi}_1 +\!\!+ \sigma_k \xrightarrow{l''} \sigma'_{k+1}$. Clearly, $\pi''' = \hat{\pi}_2 +\!\!+ \sigma'_{k+1} \xrightarrow{l} \sigma_{k+2}$ is also an OoO execution. Since the label names in $\hat{\pi}_2$ are the union of the label names in $\hat{\pi}_1$ and $n(l')$, the label names in $\hat{\pi}_1$ are less than or equal to $n(l)$, and if $n(l') < n(l)$ then π''' is an ordered execution of π.

In the second part of the proof, we establish that any ordered OoO partial execution π''' can be extended to an ordered OoO execution π_c where all instructions in the last state of π''' have been completed and no other instruction has been completed. We reuse the above reasoning to guarantee that there is an ordered OoO execution $\hat{\pi}_c$ of π_c, with last state σ_c. Finally, we show that if an OoO execution is ordered and its last state has an upper bound t such that all instructions with name smaller than t have been completed and no other instruction has been completed, then this execution is an IO execution. Therefore, for every address a, the commits for a in π are a prefix of the commits for a in the IO execution $\hat{\pi}_c$.

The vice versa case is trivial, since any IO execution is also an OoO execution. ∎

In summary, Guanciale et al. proved a two-step reordering lemma in their paper [4], which we formalized in HOL4 with substantial required effort. However, to complete the memory consistency proof, we also provide novel formal proofs that (1) the two-step reordering lemma implies the existence of an ordered OoO execution, (2) any OoO execution can be extended to complete all currently incomplete instructions, and (3) an ordered and completed OoO execution is an IO execution. These three properties were previously only hinted at and not formally stated or proved.

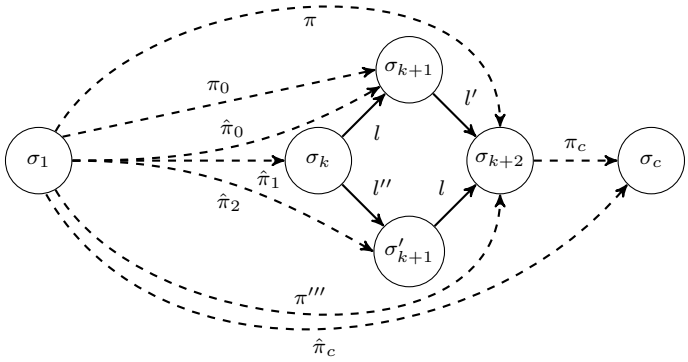

Fig. 4. Illustration of key steps in the memory consistency proof. π is the given OoO execution, σ_1 is the initial state in π, π_c is an extension to a state where all instructions in the last state of π (and no other) have been completed, and $\hat{\pi}_c$ is the IO execution with the commits from π as a prefix.

On one hand, memory consistency for the OoO semantics expresses that, subject to the conditions given by the semantics, executing instructions out-of-order is always correct. On the other hand, memory consistency provides a useful formal verification aid: to show that a real out-of-order processor pipeline satisfies memory consistency, it suffices to show that its design is simulated by the OoO semantics, without any need for dealing explicitly with instruction reordering. In practice, this requires demonstrating that processor scheduling is equally or more restrictive than MIL's conditions on resource loads and memory commits.

E. Confidentiality: Conditional Noninterference

In order to reason about information leaks via cache-based side channels transparently without an explicit cache model, we assume that the attacker can observe the address of a memory load ($dl\ a$), the address of a memory store ($ds\ a$), as well as the value of the program counter ($il\ a$). This approach makes the attacker more powerful than in many real-world scenarios, but is common in analysis of microarchitectural vulnerabilities [21] and for verifying constant time implementations [22]. In particular, the approach allows us to describe in a simple way, devoid of details on caches, when two states are indistinguishable by an attacker according to a given labeled state transition relation (for MIL, the OoO and IO relations).

Definition 1: 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 $trace(\pi_1) = trace(\pi_2)$, and $\simeq_T$ is symmetric.

In the following, we assume a binary relation on states, $\sim_\ell$, which we call the *security policy*. The security policy specifies the parts of the program state that contain sensitive/high information and the parts that contain public/low information; if states are related by $\sim_\ell$, then this means they have the same public information and therefore cannot be distinguished by the attacker prior to execution. We usually assume that the attacker knows the executing program, which means that $\sim_\ell$ also constrains the current set of instructions to execute and future instruction fetches. Moreover, $\sim_\ell$ usually requires states to be initial for the program under analysis: i.e. no instruction of the program has been executed. We take the IO semantics as a *specification* of the permitted information flows, and consider a program secure if an OoO execution of the program does not leak more information than its IO execution. The following definition formalizes this intuition:

Definition 2: A system is *conditionally noninterferent* with respect to the security policy $\sim_\ell$, written $CNI(\sim_\ell)$, if it holds that $\sim_\ell \cap \simeq_{\mathsf{IO}} \subseteq \simeq_{\mathsf{OoO}}$.

Unfortunately, conditional noninterference does not hold in general—execution of a program according to the OoO semantics can introduce new side channels. More specifically, there is a resource-initialized and well-formed state σ and policy $\sim_\ell$ such that $CNI(\sim_\ell)$ is false.

We demonstrate the CNI violation using a state with the program $I_{\mathsf{ceq}}(b_1, b_2)$ from Example 2. IO execution of the state always produces the trace $[dl\, b_1, ds\, b_2, il\, (pc_0 + 4)]$. When the flag in the register z is 1, OoO execution produces one of three traces, due to the possibility of fetching t_{c42} independently of the memory operations and the fact that the memory store must follow the memory load to respect the data dependency introduced by t_{c23}, i.e., $[dl\, b_1, ds\, b_2, il\, (pc_0 + 4)]$, $[dl\, b_1, il\, (pc_0{+}4), ds\, b_2]$, and $[il\, (pc_0{+}4), dl\, b_1, ds\, b_2]$. On the other hand, the trace $[ds\, b_2, dl\, b_1, il\, (pc_0{+}4)]$ is only possible if the flag z is not 1, since then the memory store can be reordered ahead of the memory load. By observing such traces, the attacker learns the flag in z.

For this counterexample, the security policy $\sim_\ell$ requires the programs of the two initial states to have the shape $I_0 \cup I_{\mathsf{ceq}}(b_1, b_2)$ and $I_0' \cup I_{\mathsf{ceq}}(b_1', b_2')$, where I_0 and I_0' set the initial values of the resources accessed by I_{ceq}, and requires the instructions in I_{ceq} to be undefined in the initial stores. In this case, $CNI(\sim_\ell)$ can be proved only if $\sim_\ell$ also constrains I_0 and I_0' to set the same initial value for z. Intuitively, this corresponds to considering z to be known by the attacker before program execution or to declassifying its value.

Due to the possibility of confidentiality violations, we develop a semi-automated strategy in Section VI to verify conditional noninterference of a given program.

V. Tools for Analysis of MIL Programs

A. Computing Executions and Traces Inside HOL4

Formalizing sets of instructions and names as HOL4 predicate sets was convenient for abstractly defining MIL and developing its metatheory. However, this encoding prevents many definitions from being computable, which is a prerequisite for translation to CakeML. To obtain computable definitions, we introduced a refined runtime state (i, s, c, f) that replaces all sets with polymorphic lists. We then developed list-based analogues of the semantic definitions in Section III-B, such as *addr* and *str-act*, and proved that they preserve set-based behavior, assuming that names of instructions in i are unique.

Using our list-based semantic definitions, we developed a HOL4 function for running MIL refined runtime states and returning executions, dubbed *io-bounded-execution*. Besides the initial state, the function takes an *instruction offset* argument and a *fuel* argument. We found using fuel convenient since MIL program execution is not guaranteed to terminate. In *io-bounded-execution*, we proceed by looking up the instruction at the indicated offset, completing that instruction, and moving on to the next instruction in the list until fuel runs out. We proved the correctness of *io-bounded-execution* both in terms of IO and OoO transitions, but outline only the former and defer details to the supplementary material.

Soundness: If instructions in the initial state (i, s, c, f) are sorted by name and completed up to position p, and *io-bounded-execution*$((i, s, c, f), p, n) = \pi$, then π represents an IO execution starting in the initial state and ending in a state where all instructions up to position $p + n$ are completed.

Completeness: If the initial state is well-formed, resource initialized, and has instructions sorted by name and completed up to p, *io-bounded-execution* will indeed output an execution.

We used the same approach as for *io-bounded-execution* to develop a verified function dubbed *io-bounded-trace* that only outputs the corresponding trace of an execution from a given state, with some basic optimizations to handle large states and perform many transitions. These functions are useful not only for running concrete MIL programs—they also allow us to partially automate proofs [7].

B. Refinement of Computable Functions to CakeML

While feasible for states of small to moderate size, evaluating the functions *io-bounded-execution* and *io-bounded-trace* inside HOL4 can be slow and does not scale to large and long-running MIL programs. We therefore refined our datatypes and functions for MIL to be compatible with CakeML's HOL4 *translation frontend* [23]. We then proved the refined functions equivalent to our previous list-based definitions. Once the CakeML translator accepted all our refined functions, we obtained a verified MIL evaluator as a native program.

C. Translation from BIR to MIL

To allow generating MIL programs from ISA level code, we developed an *unverified* translation in Standard ML (SML) from BIR to MIL, using the SML interfaces of each HOL4 theory. The main SML translation function takes a BIR program term and a function name g, and as a side effect defines a function in HOL4 with that name, mapping BIR block addresses (and other necessary parameters) to collections of MIL microinstructions. The function g then takes the place of *translate* in our MIL semantics; in particular, we can pass g to *io-bounded-trace* together with a (refined) MIL state.

Since MIL does not have a canonical expression semantics, we manually adapted the BIR expression semantics to MIL by introducing the corresponding expression abstract syntax and using the same HOL4 word theory operations as BIR in an executable MIL expression evaluation function.

VI. Verification of Conditional Noninterference

We develop a general verification strategy for conditional noninterference that follows the hypotheses of a lemma:

Lemma 1: If there exist (1) a relation $\mathbf{L}$ such that $\sim_\ell \cap \simeq_{\mathsf{IO}} \subseteq \mathbf{L}$ (i.e., $\mathbf{L}$ underapproximates program information leakage during IO execution), (2) and a bisimulation $\mathbf{R}$ for OoO semantics (i.e., $\mathbf{R}$ overapproximates program information leakage during OoO execution), and (3) $\sim_\ell \cap \mathbf{L} \subseteq \mathbf{R}$ (i.e., the initial knowledge of the attacker and the IO information leakage "are not less than" the OoO leakage), then $CNI(\sim_\ell)$.

Below, we demonstrate our strategy using the MIL program $I_{\mathsf{ceq}}(b_1, b_2)$ from Example 2. The supplementary material [10] contains applications of our strategy in HOL4 to verify CNI for the Example 1 program, the Example 2 program, and a program that moves values between two registers.

A. Computing the Relation $\mathbf{L}$

Our strategy uses the IO executor function *io-bounded-execution* described in Section V-A to analyze the information leakage relation symbolically, together with self composition [24]. Since the IO semantics is deterministic, we can compute the post-relation by limiting the analysis to maximal executions when programs terminate. In fact, a system is noninterferent for the IO semantics iff the traces of maximal executions of any two states in $\sim_\ell$ are indistinguishable.

For a state with I_{ceq}, the IO executor generates the trace $[dl\ b_1, dl\ b_2, il\ (pc_0 + 4)]$. By using self composition, we generate the relation $\mathbf{L} \triangleq pc_0 = pc_0' \wedge b_1 = b_1' \wedge b_2 = b_2'$, where primed variables are the parameters for the second state.

B. Identifying and Proving a Bisimulation Relation $\mathbf{R}$

Let $(I_1, s_1, C_1, F_1) = \sigma_1\ \mathbf{R}\ \sigma_2 = (I_2, s_2, C_2, F_2)$. To guarantee that the two states can produce the same observations, i.e., lists of fetches and commits, we work under the assumption of control flow preservation, reflecting the no-branch-on-secrets condition common in cryptographic practice.

This condition leads to a number of constraints on $\mathbf{R}$ that can be used in a proof search procedure:

- Preservation of executed, committed and fetched instructions, i.e., $dom(s_1) = dom(s_2)$, $C_1 = C_2$, and $F_1 = F_2$.
- Preservation of labels (addresses of PC stores and memory loads/stores), e.g., $s_1(t_{\mathsf{c43}}) = s_2(t_{\mathsf{c43}})$ for Example 2.
- Preservation of dependencies (including active stores for loads) and guards. For instance, for t_{c22} and t_{c23} in I_{ceq}, this leads to $s_1(t_{\mathsf{c21}}) = s_2(t_{\mathsf{c21}})$, since $t_{\mathsf{c21}} == 1$ is used as the guard condition of t_{c22} and t_{c23}.

These constraints are then backpropagated to previous microinstructions, which for the example results in requiring that the initial value of the flag z (needed for t_{c22}) and pc are the same (needed for t_{c43}) in s_1 and s_2.

The bisimulation proof is greatly simplified by control flow preservation. The main challenge is to prove preservation of the active stores. This is done by showing that an assignment to a name t will either have no effect on the active stores, or else the same instruction will be eliminated.

C. Proving the Entailment of the Bisimulation

The last verification step, $\sim_\ell \cap \mathbf{L} \subseteq \mathbf{R}$, is largely automated. For the initial states, each bisimulation constraint must be guaranteed by either $\mathbf{L}$ (e.g., for Example 2 the equality of pc is implied by $pc_0 = pc_0'$ in $\mathbf{L}$), when the same information is leaked by both the OoO and IO semantics, or by $\sim_\ell$ (e.g., for Example 2, the equality of the flag z can be guaranteed only if we consider the initial value of z to be public, since it is not leaked by the IO execution), when the OoO execution introduces additional leakage.

VII. Related Work

A. Theorem proving for hardware and its interfaces

Specifications of popular ISAs, e.g., ARMv8-A and RISC-V, are available for many theorem provers [14], [25], [26]. However, compilers and program analysis tools that only consider these specifications are unable to rule out illicit information flows due to microarchitectural vulnerabilities such as Spectre, Meltdown, and Foreshadow [1]–[3]. On the hardware side, theorem prover formalizations are available for HDLs and corresponding circuit synthesizers and compilers [27]–[32], but program analysis tools using such specifications have to target specific low-level microarchitectures and hardware, which may be unrelated to high-level languages or ISAs.

An alternative is to perform *end-to-end* specification and verification across high-level languages, ISAs, microarchitectures, and hardware. For instance, Lööw et al. [17] connect the compiler for the CakeML language to the Silver ISA and single-core processor in HOL4, and Erbsen et al. [33] specify and verify in Coq the functional correctness (including instruction reordering) of a system based on a pipelined processor implementing the RISC-V ISA. However, these efforts focus only on functional correctness, and are tied to a particular stack of ISA and hardware. This makes proof reuse in other settings difficult. In particular, the instruction pipeline reordering proof by Erbsen et al. is specific to a processor defined in the Kami HDL. We believe that MIL, in contrast, can enable proof reuse across end-to-end verification efforts.

B. Formal models of low-level information flow

Several works have addressed the formalization of microarchitectural optimizations, such as different forms of speculation, to capture Spectre-like vulnerabilities [21], [34]–[37]. Similarly to MIL, these proposals model an attacker that can observe the program counter, memory load addresses, and memory store addresses. Their security conditions are defined as noninterference or a conditional hyperproperty, similarly to conditional noninterference, that compares information flows

of the same program in a speculative and a sequential semantics. The semantics by Barthe et al. [35] describes out-of-order execution, but memory commits have to be done in-order and consequently memory consistency is straightforward. Fadideh et al. [37] consider a SAT-based register transfer level analysis of transient execution for concrete OoO processor designs, but they do not provide a general model. Other works only consider speculative in-order instruction processing.

Many of these works have inspired the implementation of tools, e.g., Spectector [38], to analyze program side channels using some form of (relational) symbolic execution. However, to our knowledge, the only mentioned work whose semantics and verification approach has reached an interactive theorem prover is that of Cheang et al. [36], which was formalized by Griffin and Dongol in Isabelle/HOL [39]. Using a translation from C-like programs to Isabelle theories similar to our BIR translation, Griffin and Dongol reason about information flow during speculative execution using Hoare-style triples, but they do not account for out-of-order execution. While our MIL semantics introduces nondeterminism relationally, i.e., by some states simply having several possible transitions according to the OoO step relation, the semantics used by Griffin and Dongol consults an abstract oracle to resolve nondeterminism [21].

C. Validation of hardware information flow models

Buiras et al. [15], [40] and Oleksenko et al. [41] developed tools (called Scam-V and Revizor, respectively) to validate hardware information flow models. The approaches are based on testing leakage models (e.g., the attacker observations of Section IV-E) using black box testing on actual CPUs. Both Scam-V and Revizor use a variation of conditional noninterference, where the goal is to establish that states that produce indistinguishable traces in a model produce indistinguishable cache footprints on the real hardware. We believe such tools can facilitate trustworthy connections between MIL-based information flow analyses and hardware behavior.

VIII. Conclusion

We presented a formalization in HOL4 of MIL, a language which captures key features of microarchitectures to allow reasoning about low-level program information flow. The formalization includes the in-order and out-of-order dynamic semantics of MIL, a proof of memory consistency between the two semantics, and a notion of conditional noninterference that rules out trace-driven cache based side channels. The formalization is around 34,000 lines of code with examples, and took around 24 person months to develop. The code [10] was tested on HOL4 kananaskis-14 and PolyML 5.9.

We envision that our MIL formalization and tools will be integrated into a trustworthy program information flow analysis workflow based on CakeML, where binaries for ISAs supported by HolBA are first represented in BIR and then translated to MIL to establish conditional noninterference or to demonstrate side channels. Our unverified BIR-to-MIL translation and verified example programs indicate that the workflow is feasible, but the manual effort of conditional noninterference proofs is currently the main obstacle. In particular, bisimulation based reasoning can easily lead to unproductive exploration of the many possible transitions available due to nondeterminism. However, even without full automation of conditional noninterference proofs, we believe MIL and its metatheory and tools can improve productivity in formal verification of confidentiality properties of practical systems.

References

[1] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Symposium on Security and Privacy*, 2019, pp. 1–19.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium*, 2018, pp. 973–990.

[3] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security Symposium*, 2018, pp. 991–1008.

[4] R. Guanciale, M. Balliu, and M. Dam, "InSpectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis," in *Conference on Computer and Communications Security*, 2020, pp. 1853–1869.

[5] HOL development team, "HOL interactive theorem prover," 2022. [Online]. Available: https://hol-theorem-prover.org

[6] O. Acıiçmez and Ç. K. Koç, "Trace-driven cache attacks on AES," in *Information and Communications Security*, 2006, pp. 112–121.

[7] B. Barras, "Programming and computing in HOL," in *Theorem Proving in Higher Order Logics*, 2000, pp. 17–37.

[8] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish, "The verified CakeML compiler backend," *Journal of Functional Programming*, vol. 29, p. e2, 2019.

[9] A. Lindner, R. Guanciale, and R. Metere, "TrABin: Trustworthy analyses of binaries," *Sci. Comput. Program.*, vol. 174, pp. 72–89, 2019.

[10] K. Palmskog, X. Yao, N. Dong, R. Guanciale, and M. Dam, "MIL formalization source code and documentation," 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6997534

[11] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," *Proc. Computer Architecture*, pp. 34–44, 1985.

[12] W.-M. Hwu and Y. N. Patt, "HPSm, a high performance restricted data flow architecture having minimal functionality," in *International Symposium on Computer Architecture*, 1986, pp. 297–306.

[13] K.-A. Tran, A. Jimborean, T. E. Carlson, K. Koukos, M. Själander, and S. Kaxiras, "SWOOP: Software-hardware co-design for non-speculative, execute-ahead, in-order cores," in *Conference on Programming Language Design and Implementation*, 2018, pp. 328–343.

[14] A. Fox, "Improved tool support for machine-code decompilation in HOL4," in *Interactive Theorem Proving*, 2015, pp. 187–202.

[15] H. Nemati, P. Buiras, A. Lindner, R. Guanciale, and S. Jacobs, "Validation of abstract side-channel models for computer architectures," in *Computer Aided Verification*, 2020, pp. 225–248.

[16] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the Pentium 4 processor," *Intel Technology Journal*, vol. 5, 2001. [Online]. Available: http://www.ecs.umass.edu/ece/koren/ece568/papers/Pentium4.pdf

[17] A. Lööw, R. Kumar, Y. K. Tan, M. O. Myreen, M. Norrish, O. Abrahamsson, and A. Fox, "Verified compilation on a verified processor," in *Conference on Programming Language Design and Implementation*, 2019, pp. 1041–1053.

[18] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Symposium on Principles of Programming Languages*, New York, NY, USA, 1988, pp. 12–27.

[19] J. Hurd, "Predicate subtyping with predicate sets," in *Theorem Proving in Higher Order Logics*, 2001, pp. 265–280.

[20] J. Harrison, "The HOL Light theory of Euclidean space," *J. Autom. Reasoning*, vol. 50, pp. 173–190, 2012.

[21] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "SoK: Practical foundations for software Spectre defenses," in *Symposium on Security and Privacy*, 2022, pp. 1517–1517.

[22] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *USENIX Security Symposium*, 2016, pp. 53–70.

[23] M. O. Myreen and S. Owens, "Proof-producing translation of higher-order logic into pure and stateful ML," *Journal of Functional Programming*, vol. 24, no. 2-3, pp. 284–315, 2014.

[24] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," in *Computer Security Foundations Workshop*, 2004, pp. 100–114.

[25] A. Fox and M. O. Myreen, "A trustworthy monadic formalization of the ARMv7 instruction set architecture," in *Interactive Theorem Proving*, 2010, pp. 243–258.

[26] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 2019.

[27] M. Vijayaraghavan, A. Chlipala, Arvind, and N. Dave, "Modular deductive verification of multiprocessor hardware designs," in *Computer Aided Verification*, 2015, pp. 109–127.

[28] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: A platform for high-level parametric hardware specification and its modular verification," *Proc. ACM Program. Lang.*, vol. 1, no. ICFP, 2017.

[29] A. Lööw and M. O. Myreen, "A proof-producing translator for Verilog development in HOL," in *International Conference on Formal Methods in Software Engineering*, 2019, pp. 99–108.

[30] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, "The essence of Bluespec: A core language for rule-based hardware design," in *Conference on Programming Language Design and Implementation*, 2020, pp. 243–257.

[31] A. Lööw, "Lutsig: A verified Verilog compiler for verified circuit development," in *Conference on Certified Programs and Proofs*, 2021, pp. 46–60.

[32] Y. Herklotz, J. D. Pollard, N. Ramanathan, and J. Wickerson, "Formal verification of high-level synthesis," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, 2021.

[33] A. Erbsen, S. Gruetter, J. Choi, C. Wood, and A. Chlipala, "Integration Verification Across Software and Hardware for a Simple Embedded System," in *Conference on Programming Language Design and Implementation*, 2021.

[34] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *Symposium on Security and Privacy*, 2021, pp. 1868–1883.

[35] G. Barthe, S. Cauligi, B. Grégoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, and P. Schwabe, "High-assurance cryptography in the Spectre era," in *Symposium on Security and Privacy*, 2021, pp. 788–805.

[36] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in *Computer Security Foundations Symposium*, 2019, pp. 288–303.

[37] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz, "A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors," in *Design Automation Conference*, 2020, pp. 1–6.

[38] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "SPECTECTOR: Principled detection of speculative information flows," in *Symposium on Security and Privacy*, 2020, pp. 1–19.

[39] M. Griffin and B. Dongol, "Verifying secure speculation in Isabelle/HOL," in *Formal Methods*, 2021, pp. 43–60.

[40] P. Buiras, H. Nemati, A. Lindner, and R. Guanciale, "Validation of side-channel models via observation refinement," in *International Symposium on Microarchitecture*, 2021, pp. 578–591.

[41] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, "Revizor: Testing black-box CPUs against speculation contracts," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 226–239.