Specifying and Generating Abstract Models for Formal Security Analysis

Adwait Godbole
Kevin Cheang
Yatin Manerkar
Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-230
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-230.html

September 8, 2023

Acknowledgement

This work was supported by DARPA contract FA8750-20-C0156, NSF grant 1837132 and a gift from Intel Corporation.

Specifying and Generating Abstract Models for Formal Security Analysis

Adwait Godbole*, Kevin Cheang*, Yatin A. Manerkar†, Sanjit A. Seshia*
*University of California, Berkeley
†University of Michigan, Ann Arbor

Abstract— **Hardware execution attacks exploit subtle microarchitectural interactions to leak secret data. While checking programs for the existence of such attacks is essential, verification of software against the full hardware implementation does not scale. Verification using abstract formal models of the hardware can help provide strong security guarantees while leveraging abstraction to achieve scalability. However, hand-writing accurate abstract models is tedious and error-prone. Hence, we need techniques to automatically generate models which enable sound yet scalable security analysis.**

In this work, we propose micro-update models as a modelling framework that enables sound and abstract modelling of microarchitectural features. We also develop algorithms to semi-automatically generate micro-update models from RTL. We implement our modelling and generation framework in a prototype tool. We evaluate our approach and tool by synthesizing micro-update models for the `Sodor5Stage` processor and components from the `cva6` (Ariane) processor. We demonstrate how these models can be generated hierarchically, thus increasing scalability to larger designs. We observe up to 8× improvement in run time when performing analysis with the generated models as compared to the source RTL.

I. INTRODUCTION

The landscape of hardware execution attacks has evolved since Spectre [38] and Meltdown [40]. Recent attacks exploit deeper microarchitectural state such as store buffers and line-fill-buffers [16], [55], [63], [64]. Checking software for the presence of these attacks requires analyses that take this microarchitectural state into account. However, owing to microarchitectural complexity, verification of software programs directly against the source RTL does not scale with the program being verified. Abstract formal models that only represent parts of the microarchitecture relevant to the attack while eliding the rest are simpler and more interpretable than the RTL. Verifying SW against such models instead of the source HW can improve scalability. However, accurate modelling is necessary for preserving strong guarantees; imprecise models can lead to software being deemed secure when in reality it is not.

Several recent works apply formal methods to verify software against hardware attacks [9], [17], [27], [48], and to identify new ones [22], [61]. These works perform verification against *manually crafted* abstract models that preserve only the vulnerability-relevant components of the HW design. However, manually writing models while ensuring accuracy of preserved components is tedious and error-prone, and requires a deep understanding of the microarchitecture. While a security analyst can indicate *which* source design components they want to perform software-side security analysis against, it is hard to manually specify *how* these components should behave so that the model is sufficiently accurate. Thus, it is desirable to have a technique that, given some *signals-of-interest*, generates a formal model that accurately captures the behaviours of these signals w.r.t. the source RTL.

In this work, we propose ***micro-update models*** as a modelling formalism that can be used in security analysis of software running on the source RTL design. To address the challenges involved in hand-writing models, we develop a framework to mostly automatically ***generate micro-update models*** from RTL designs. The generated model achieves abstraction by capturing signals-of-interest specified by the user, eliding the rest, and achieves accuracy by ensuring functional equivalence of these signals w.r.t. the source RTL. This guarantees soundness, i.e. security properties over the signals-of-interest (e.g. information-flow/non-interference (NI) [20], [25]) that hold on the lifted model also hold on the RTL.

By imperatively modelling functional behaviours of the signals-of-interest, the micro-update model: (a) enables correct-by-construction generation as the model can be checked against the source RTL (e.g. with a functional equivalence proof) and (b) enables sound verification of SW w.r.t. security properties (e.g. NI). In contrast, ISA-based models (e.g. [18], [26]), are equivalent only w.r.t. the *architectural state*, and lack microarchitectural detail necessary for security analysis. On the other hand, approaches such as μspec [42], [43] formulate *axiomatic* ordering models that abstract away functional behaviours. Due to the lack of imperatively modelled state, μspec requires global invariants for checking accuracy with the source RTL. Defining these invariants is challenging [29], [44]. Secondly, properties such as NI [20], [25], which requires a model that preserves functional behaviours cannot be expressed with these models. In conclusion, imperatively modelling functional behaviours of the signals-of-interest makes checking alignment with source RTL, and security properties on SW easier with the micro-update model.

Our micro-update model generation framework is based on *lifting*, which aims to extract high-level code from low-level source implementations while maintaining important properties with respect to the source. While lifting has been demonstrated in the PL and systems domains [3], [4], [35], [56], we extend it to hardware. We demonstrate how lifting can

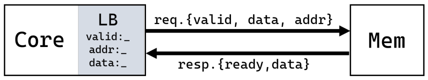


Fig. 1: Schematic for the `Sodor5Stage` processor. Our modification, the load buffer (LB), is shown in grey.

be performed *hierarchically*, thereby increasing its scalability. Thus our techniques aid in generating micro-update models from RTL, which then can be used for software security analysis.

Contributions. Our contributions are as follows:

- **Micro-update models as a new formalism:** We propose micro-update models as a formalism for abstract modelling of hardware designs. Micro-update models provide us with finer temporal resolution than ISA-level models and preserve functional behaviour. Thus, they provide accurate abstractions of the hardware against which software can be verified for microarchitectural attacks.
- **Synthesis techniques for micro-update models:** We develop a methodology to generate micro-update models from RTL using a novel instantiation of *oracle-guided synthesis*[1] [33], [34], [53]. We show how one can generate models *hierarchically*, improving the scalability of synthesis.
- **Empirical Demonstration:** We evaluate our approach on the `Sodor5Stage` processor [62] and components from the `cva6` processor [1] by generating micro-update models for them. We demonstrate how applying the generated models to perform security analysis of software can lead to performance improvements over the source RTL.

Outline. In §II we motivate and contrast the features of our modelling approach with existing ones through an example. In §III we describe the structure of our models and their semantics. In §IV we describe the techniques that we use to generate our models. We discuss the experimental evaluation in §V. Finally, §VI discusses limitations, §VII is related work and §VIII concludes.

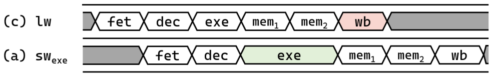
II. Motivating Example

A. Example: modified `Sodor5Stage` core

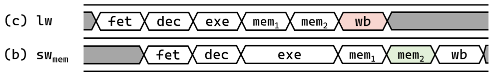
`Sodor5Stage` [62] is a 5-stage in-order processor implementing the RV32UI instruction set [66]. We augment this processor design with a new component, called a load buffer (LB) as shown in Fig. 1. The LB caches the most recently loaded value in its `data` field. It also maintains an address field and a `valid` bit marking whether the entry in the buffer is valid (not outdated). This cached value can be consumed by a subsequent load if its address matches and the entry is valid:

$$\mathit{canLoad} \equiv (\texttt{loadAddr} == \texttt{addr}) \wedge \texttt{valid}$$

However, the entry is flushed by any subsequent store instruction to prevent reads from outdated entries. This feature does not change the ISA-level behaviour of the processor (program counter, register file and memory); its effects are purely microarchitectural. Now we discuss how a hypothetical attack can leak information by using the LB as a side channel.



Behaviour with $\texttt{sw}_{\texttt{exe}}$.



Behaviour with $\texttt{sw}_{\texttt{mem}}$.

Fig. 2: Different microarchitectural behaviours (with a 1-cycle memory latency) for different implementation choices of `sw`.

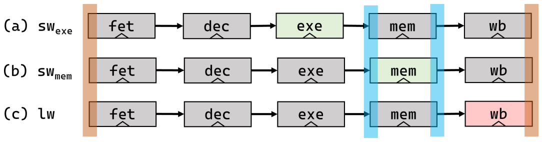


Fig. 3: Pipeline stages for two possible implementations for the `sw` instruction: in (a) the LB flush happens during the `exe` stage, while in (b) it happens during the `mem` stage; (c) illustrates that the refill of the LB due to the `lw` instruction takes place in its `wb` stage.

B. The `lb` optimization vulnerability

Features such as the LB, while not architecturally visible, can have several possible microarchitectural implementations. In our example, *exactly when* the store instruction (`sw`) invalidates the LB entry depends on the implementation. In Fig. 3(a,b) we consider two such implementations for `sw`. In Fig. 3(a) the LB invalidation (flush) takes place during the `exe` stage of the `sw` instruction, while in (b) it does so during the `mem` stage. We call these implementations $\texttt{sw}_{\texttt{exe}}$ and $\texttt{sw}_{\texttt{mem}}$ respectively. The implementation of a load (`lw`) is identical across these cases (ref. Fig. 3(c)). It loads from memory in the `mem` stage and refills the LB in the `wb` stage.

While architecturally invisible, such implementation differences crucially affect security analysis. To see this let us consider the effect of executing a load consecutively followed by a store, and compare them across the $\texttt{sw}_{\texttt{exe}}$ and $\texttt{sw}_{\texttt{mem}}$ implementations. In the case of $\texttt{lw}; \texttt{sw}_{\texttt{exe}}$ (Fig. 2 top), the LB has a valid entry at the end of execution (refilled due to the `lw` instruction). On the other hand, in the case of $\texttt{lw}; \texttt{sw}_{\texttt{mem}}$ (Fig. 2 bottom), the LB entry is flushed by the `sw` *after* the refill caused by `lw`. Consequentially, $\texttt{lw}; \texttt{sw}_{\texttt{exe}}$ leads to an LB which is tainted (due to the `lw`), while $\texttt{lw}; \texttt{sw}_{\texttt{mem}}$ does not.

A timing-based side-channel (e.g. Prime+Probe [24], [41]), can allow an attacker to infer the contents of the LB through a timing measurement. In such a case, the $\texttt{lw}; \texttt{sw}_{\texttt{exe}}$ can lead to secret data (e.g. private keys) getting revealed to the attacker through the LB side channel. On the other hand, since the $\texttt{lw}; \texttt{sw}_{\texttt{mem}}$ sequence leads to a *sanitized* LB, (i.e. the LB is untainted), a timing measurement *cannot* reveal victim secrets.

---

[1] Oracles are humans/tools that provide guidance to the synthesis engine in a specified format through a query-response interface.

Such microarchitectural details can render software insecure under certain implementations, while secure under others. Hence analyses checking software security must operate on models that expose microarchitectural detail.

C. Instruction-level approaches

Models that capture ISA-level behaviours [14], [18], [26], [57], [65] are precise only with respect to software-visible architectural state (e.g. program counter, register file, memory, and CSRs). However, as §II-B shows, hardware attacks (e.g., [16], [38], [40], [55], [63], [64]) manifest at the microarchitecural level and exploit software-invisible features (e.g. caches, buffers, predictors). Consequentially, ISA-level models cannot express these attacks. This is also true of instruction-level-abstraction (ILA) based approaches [30], [32], [71] which aim to model accelerators in addition to general-purpose cores.

The issue with ISA-level approaches is not only with the state elements modelled, but also with their *temporal semantics*. To illustrate this, let $[\![i_1; i_2]\!]$ denote the effect (semantics) of the sequence of two (software) instructions on the HW design. If one is only concerned with the architectural state, σ_{arch}, the effect of $[\![i_1; i_2]\!]$ is equivalent to the effect of these instructions executed in sequentially and *in isolation*, $[\![i_1]\!]$ followed by $[\![i_2]\!]$ (i.e., i_1 finishes execution before i_2 begins):

$$\sigma_{\mathsf{arch}}\ [\![i_1; i_2]\!]\ \sigma'_{\mathsf{arch}} \iff \sigma_{\mathsf{arch}}\ [\![i_1]\!]\ \sigma''_{\mathsf{arch}}\ [\![i_2]\!]\ \sigma'_{\mathsf{arch}}$$

However, the effect of a stream of instructions with respect to the *microarchitectural* state $\sigma_{\mu\mathsf{arch}}$ can be different than that of individual instructions operating in isolation:

$$\sigma_{\mu\mathsf{arch}}\ [\![i_1; i_2]\!]\ \sigma'_{\mu\mathsf{arch}} \not\Longleftrightarrow \sigma_{\mu\mathsf{arch}}\ [\![i_1]\!]\ \sigma''_{\mu\mathsf{arch}}\ [\![i_2]\!]\ \sigma'_{\mu\mathsf{arch}}$$

In the LB example, the semantics of both $\texttt{sw}_{\texttt{exe}}$ and $\texttt{sw}_{\texttt{mem}}$ are identical when executed in isolation. This is true *even with respect to the LB state*, as both flush the LB before finishing execution. ISA-based models (including ILA) define semantics for isolated execution between the fetch and commit points of that instruction (orange lines in Fig. 3). Consequentially, an ISA-based model, even if it were enriched with the LB state, would model $\texttt{sw}_{\texttt{mem}}$ and $\texttt{sw}_{\texttt{exe}}$ identically.

While isolated instruction-level modelling is appropriate for architectural state, it falls short when representing the microarchitecture. In the LB example, the difference between $\texttt{sw}_{\texttt{exe}}$ and $\texttt{sw}_{\texttt{mem}}$ instructions emerges only when they interact with a `lw` instruction. In summary, we need a model that captures the microarchitectural interactions between instructions.

D. Axiomatic ordering-based approaches

Approaches such as μspec [42] model the microarchitecture as a set of *axiomatic ordering constraints* over the entire execution of a program. Executions satisfying these constraints are valid (can be observed) while others cannot. Since these constraints are over full executions, μspec can express behaviours of instructions executing simultaneously (as in a pipeline), and does not suffer from the isolation problem discussed earlier.

μspec largely captures only ordering constraints, abstracting away the functional behaviour. However, functional behaviour is often necessary to model low-level security-relevant microarchitecure. Consider a victim program running on a processor with an LRU-cache. Such processors can leak victim secrets through the LRU state [69]. Detecting such vulnerabilities in the victim program requires security properties that precisely identify memory accesses which lead to attacker-distinguishable LRU states. A non-interference (NI) [25], [60] property identifies such accesses by requiring that subsequent attacker-observable cache hit/miss outcomes should not depend on victim secrets. However, since expressing security properties such as NI requires a model that exposes the functional behaviour of signals, it is infeasible with μspec models. A micro-update model however, can represent the low-level LRU policy functionally, enabling NI-based analysis.

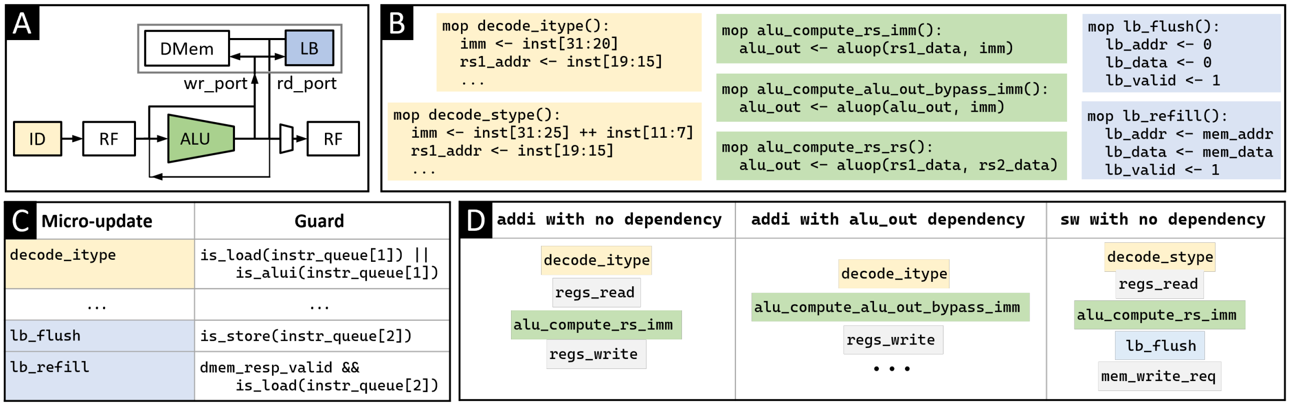
μspec-based security analysis approaches (e.g. [48], [61]) work around this by explicitly identifying execution fragments which are indicative of vulnerabilities, e.g., exploit patterns (§3A-II in [61]) or *transmitters* (§3.2.1 in [48]). While NI uniformly captures all possible vulnerable cases as a single property, these execution fragments (patterns/transmitters) must be manually enumerated. This requires a deep understanding of the microarchitecture and can result in false negatives if some vulnerable execution fragments are missed.

The second advantage of the micro-update model is that it is *operational*. That is, it represents executions as imperative transitions performed on some state at every step. This allows checking *per-step* equivalence (§IV-B) with the RTL design with a model checker (e.g. SymbiYosys [19], JasperGold). Such checks are essential to ensure accuracy when lifting a model. Ordering-based models, on the other hand, are *axiomatic* and define behaviours in terms of declarative constraints. Checking these constraints against the RTL requires *global invariants* (i.e. invariants over full executions as opposed to individual steps). These invariants can be difficult to generate and check automatically (e.g. [29], [44]), which in turn makes checking accuracy with respect to the source RTL more difficult.

In conclusion, the micro-update model (a) represents executions at a finer granularity than ISA models, (b) preserves functional behaviour enabling NI-based security analysis and (c) represents executions operationally, thus allows easier equivalence checking with the source RTL.

III. MICRO-UPDATE MODELS

In this section, we illustrate the micro-update model through Figure 4. Fig. 4(A) and (B) respectively illustrate a simplified processor pipeline and some *micro-updates* for that pipeline. Each micro-update is an imperative code snippet that describes the functional operation performed on signals in the model. Signals in the model are mapped to equivalent signals from the source RTL. For example, the flush micro-update (`lb_flush`) in Fig. 4(B) operates on the LB: it clears the data and address fields and sets the valid bit to zero. While micro-updates (§III-A2) define the functional operation performed



```
mop decode_itype():
  imm <- inst[31:20]
  rs1_addr <- inst[19:15]
  ...

mop decode_stype():
  imm <- inst[31:25] ++ inst[11:7]
  rs1_addr <- inst[19:15]
  ...
```

```
mop alu_compute_rs_imm():
  alu_out <- aluop(rs1_data, imm)

mop alu_compute_alu_out_bypass_imm():
  alu_out <- aluop(alu_out, imm)

mop alu_compute_rs_rs():
  alu_out <- aluop(rs1_data, rs2_data)
```

```
mop lb_flush():
  lb_addr <- 0
  lb_data <- 0
  lb_valid <- 1

mop lb_refill():
  lb_addr <- mem_addr
  lb_data <- mem_data
  lb_valid <- 1
```

Micro-update	Guard
decode_itype	is_load(instr_queue[1]) || is_alui(instr_queue[1])
...	...
lb_flush	is_store(instr_queue[2])
lb_refill	dmem_resp_valid && is_load(instr_queue[2])

addi with no dependency	addi with alu_out dependency	sw with no dependency
decode_itype	decode_itype	decode_stype
regs_read	alu_compute_alu_out_bypass_imm	regs_read
alu_compute_rs_imm	regs_write	alu_compute_rs_imm
regs_write	• • •	lb_flush
		mem_write_req

Fig. 4: **(A)** A simplified processor with a LB and bypassing paths (greyed), **(B)** non-exhaustive set of micro-updates for this processor, **(C)** guards corresponding to these micro-updates, **(D)** sample instructions with some of the induced micro-updates.

on signals, *guards* (ref. Fig. 4(C)) are Boolean predicates that control *when* a micro-update is triggered. Guards (§III-A3) are defined over a set of *trigger signals*. For example, the guard for `lb_flush` is true (and `lb_flush` is executed) at cycles where the instruction in the memory stage (`instr_queue[2]`) is a store. Trigger signals typically are top-level inputs to the source design. The model as a whole captures a slice of the design, as identified by the signals-of-interest (§III-A1).

Example 1: Consider an `addi` instruction running on the processor in Fig. 4(A). During execution `addi` interacts with several components (e.g. decode unit), by invoking multiple micro-updates. Each micro-update is executed when the corresponding guard (Fig. 4(C)) evaluates to true (e.g. `decode_itype` is executed when `instr_queue[1]`, the decoded instruction, is an I-type instruction). While `instr_queue[1]` depends on a single instruction input, in general, guards can depend on multiple inputs. For example, if `rs1_addr(instr_queue[1]) == rd_addr(instr_queue[2])`, then there is a data dependency between the current and previous instructions. In this case, the `alu_compute_alu_out_bypass_imm` micro-update is invoked (ref. Fig. 4(D)) instead of `alu_compute_rs_imm` (which uses data from the RF). While we take the example of `addi`, the model can be extended with other instructions and thus captures a thicker slice of the RTL (with more signals, micro-updates and guards). We illustrate some micro-updates for `sw` in Fig. 4(D) (last column).

We now discuss how this model addresses concerns from §II.

a) ***Finer temporal resolution****:* ISA-based approaches define instruction semantics as if instructions execute atomically in isolation. The micro-update model decomposes instruction execution into several micro-updates, and evaluates guards and micro-updates on a per-cycle basis. Consequently, a micro-update model has finer temporal resolution than ISA-based models by identifying the exact cycle at which a microarchitectural state update happens (blue markings in Fig. 3). Hence, the micro-update model can differentiate between cases that have identical ISA-level but different microarchitectural behaviour. For example, it can differenciate between the $\mathtt{sw}_{\mathtt{exe}}$ and $\mathtt{sw}_{\mathtt{mem}}$ implementations (as introduced in §II-B), and specifically identify $(\mathtt{lw}; \mathtt{sw}_{\mathtt{exe}})$ as being vulnerable.

Imperatively modelled functional detail*:* While an axiomatic modelling approach such as μspec also provides finer temporal resolution (through microarchitectural events), it lacks functional detail. A micro-update model, however, captures functional behaviour through micro-update bodies (Fig. 4(B)). As discussed in §II-D, this allows specification of security properties like NI. Secondly, micro-updates describe imperative operations on the model state (§III-A1). This enables per-cycle equivalence proofs with the RTL, which is essential for accurate model lifting. This contrasts with μspec-like axiomatic approaches which require global invariants ([29], [44]).

A. Components of the micro-update model

1) Signals: A micro-update model captures a design slice which is identified by a set of *signals-of-interest*, which we denote as S_{data}. When generating the model, the user can specify these based on the RTL signals over which they wish to perform security analysis. At each cycle, each data signal from S_{data} is mapped to a value by an *assignment* $\sigma_{data} : \mathsf{S}_{data} \to \mathbb{V}$.

2) Micro-updates: A micro-update is an imperative codeblock that typically operate on small subsets of signals (Fig. 4(B)). The model as a whole comprises a set of micro-updates, L. At each cycle, signals from S_{data} are updated based on the micro-updates from L that are invoked during that cycle. The body of each micro-update $\mathsf{t} \in \mathsf{L}$ provides the semantics $[\![\mathsf{t}]\!]$, which defines how t transforms the S_{data} signals when invoked. An instruction typically performs several of these micro-updates during its lifetime, as Fig. 4(D) illustrates.

3) Trigger signals and guards: The micro-updates are invoked based on a Boolean condition called a *guard*. A guard for a micro-update t is a boolean condition G_{t} evaluated over the *trigger signals* S_{trig}: $G_{\mathsf{t}}(\mathsf{S}_{trig}) \in \{\mathtt{true}, \mathtt{false}\}$. The micro-update t is executed at a cycle iff G_{t} evaluates to true. Trigger signals typically correspond to top-level inputs of the RTL design (e.g. instruction the data memory ports). Thus the micro-update model can directly relate micro-update invocation to the instructions inputs provided to the design. This is beneficial since it allows micro-update models to be easily connected to software-side analyses (which operate over ISA instructions).

B. Properties of micro-updates

Uses and modifies: Knowledge of the signals used and modified by each micro-update allows us to optimize synthesis of the micro-update model (§IV-D2). This information can be extracted from the micro-update code blocks (e.g., by an AST traversal). We denote the set of signals used and modified by micro-update t as $\mathsf{use}(\mathsf{t})$ and $\mathsf{mod}(\mathsf{t})$. For example, `lb_refill` in Fig. 4(B) has $\mathsf{use}(\mathsf{t}) = \{\texttt{mem\_addr}, \texttt{mem\_data}\}$ and $\mathsf{mod}(\mathsf{t}) = \{\texttt{lb\_addr}, \texttt{lb\_data}, \texttt{lb\_valid}\}$.

Sequential and combinational micro-updates: We allow micro-updates of two types: combinational and sequential. These resemble sequential and combinational logic seen in Verilog-like HDLs and have similar semantics.

M-set: We use the term M-set to identify a set of micro-updates that can be triggered at the same cycle of execution. Not all sets of micro-updates can be a valid M-set. For an M-set to be valid, there should not be any combinational cycles, and the constituent micro-updates must modify disjoint signals. Both these conditions can be checked from the uses and modifies information. The effect of triggering an M-set follows the usual sequential and combinational semantics.

IV. MICRO-UPDATE MODEL SYNTHESIS

In this section, we discuss our framework (Fig. 5) to generate micro-update models using formal synthesis techniques. We begin by discussing the key challenges that we face.

A. The challenge posed by formal synthesis

While (semi)-automated approaches to synthesize formal models from RTL are useful, formal synthesis is challenging (more so than formal verification), since it requires a search over model candidates, while also verifying them for correctness. Synthesizing models involving ordering/timing constraints (e.g. for hardware) is especially challenging (compared to program synthesis [46], [58]) since the generated model must capture temporal constraints in addition to functional correctness. We navigate this complexity by decomposing the synthesis objective of a monolithic model into smaller functional (micro-updates) and temporal (guards) components.

Since individual micro-updates operate on small, local subsets of signals, identifying them is easier compared to identifying the right coordination between micro-update invocations. The latter requires a deep understanding of inter-dependencies between micro-updates. This is hard to do manually; e.g. ∼10 out of 42 RTL-bug issues in the `cva6` processor [2] were due to imprecisely coordinated inter-dependencies. Our synthesis procedure does the heavy lifting of determining this choreography of micro-update invocations.

B. Problem formulation and guarantee

Our synthesis framework allows the user to specify the signals S_{data}, which they wish to be captured by the generated model. The choice of S_{data} exposes the tradeoff inherent to the development of formal models: models for a larger S_{data}, while more detailed, are harder to synthesize and analyze, and vice-versa. Given an RTL design, the signals S_{data}, and library L, our goal is to generate guards G_t for each $\mathsf{t} \in \mathsf{L}$, so that Property S_{data}-equivalence holds.

Property 1 (S_{data}-Equivalence): Our synthesis technique generates models in which the behaviours of identified signals-of-interest (S_{data}) are equivalent to behaviours of corresponding signals in the source design.
This property is guaranteed to hold because of the equivalence check performed as the final signoff on the model (§IV-F). With Property 1, a non-interference-based security proof on the model is guaranteed to imply security of the source design. We demonstrate an application of this in §V-B1 and §V-C3.

C. Synthesis overview: Figure 5

Our synthesis procedure operates in two phases. In the first phase, we generate simulation traces (§IV-D1) and use these traces to generate *candidate* M-sets (§IV-D2). These M-sets, while consistent with the simulation traces, may contain spurious candidates that do not apply generally. We use two techniques to weed out spurious candidates: distinguishing oracles and cover properties (§IV-D3). Once we eliminate these, we move to the second phase: guard synthesis (§IV-E). Guard synthesis takes in M-sets from the previous phase as well as the set of trigger signals and synthesizes guards for each micro-update. Guard synthesis results in a complete micro-update model which we check for S_{data}-equivalence (§IV-F).

1) Signals-of-interest and RTL-mapping: The synthesis procedure takes as input the signals S_{data} (§III-A1) to be included in the micro-update model, and a mapping from these signals to corresponding signals in the RTL design. The generated model then satisfies the S_{data}-equivalence property (Property 1).

2) Micro-update library: We also require the user to provide a library of micro-updates (L in Fig. 5). Candidate M-set generation (§IV-D) searches over L to construct valid M-sets for the simulated traces. As mentioned before, since micro-updates operate on small subsets of signals, specifying this library only requires local understanding of the design. In our experiments, we observe that library specification time was negligible compared to design harnessing/instrumentation.

Micro-update library sensitivity: We now comment on the sensitivity of synthesis to the user-provided micro-update library. Firstly, libraries which have incorrect or *irrelevant* micro-updates (e.g. LB micro-updates for a model with only ALU operations) does not affect the soundness of the generated model. Such micro-updates are filtered out by candidate M-set generation, i.e., their guard is assigned `false`. Since this phase does not invoke (costly) solvers, synthesis performance is also not heavily affected. Libraries which are

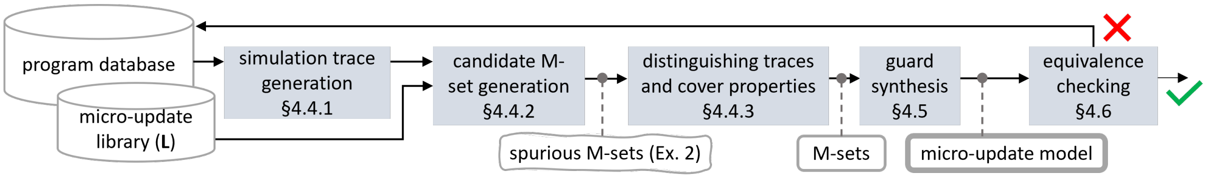


Fig. 5: Synthesis pipeline for generating micro-update models.

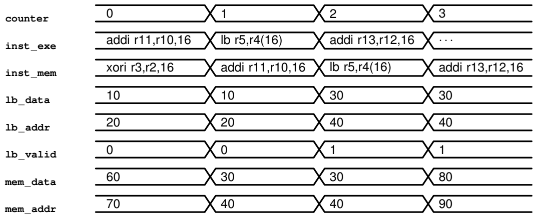


Fig. 6: A sample execution trace from simulation.

incomplete, i.e. do not contain all micro-updates necessary to justify the simulation traces, fail during M-set generation. In such cases, our approach identifies the failing simulation step and signals. This helps to add missing micro-updates.

D. Generating M-sets

1) Simulator-based trace generation: We generate a set of random test traces of the RTL execution from a simulator (e.g. iverilog [67]). These traces must include all the signals from S_{data}. Figure 6 shows a fragment of such a trace which includes signals for the memory port, and the LB component described in §II.

2) Generating candidate M-sets: A candidate M-set is a set of micro-updates that transforms the assignment to S_{data} some cycle i of a trace into the assignment at the next cycle $i+1$. As the first part of the synthesis procedure, we extract candidate M-sets that match the simulation traces. For each step of the simulation traces, we obtain a pre-post assignment pair, $(\sigma_{data}, \sigma'_{data})$. For example, from the second step in Fig. 6 we extract: $\sigma_{data} = [\texttt{lb\_data} \mapsto 10, \cdots]$ and $\sigma'_{data} = [\texttt{lb\_data} \mapsto 30, \cdots]$. The generation of candidate M-sets for $(\sigma_{data}, \sigma'_{data})$ is performed via a depth-first-search (DFS) over micro-updates from the library L. The DFS explores a sequence of micro-updates, which incrementally transforms σ_{data} into σ'_{data}. A naive DFS will consider all possible sequences of micro-updates. However, typically, several micro-updates do not depend on each other. We exploit this notion of independence to eliminate redundant search.

In general, we say that micro-update t' depends on micro-update t if t' is combinational and $\mathsf{use}(\mathsf{t}') \cap \mathsf{mod}(\mathsf{t})$ is non-empty. Since sequential micro-updates only consume values from the previous step, they are not dependent on anything. Subsets of independent micro-updates can be searched independently in the DFS, which avoids redundant orderings.

Search pruning is also performed based on the fact that only one micro-update modifies a given signal at each step. Hence, after choosing a micro-update modifying a signal s, the DFS ignores all other micro-updates modifying s. While the number of micro-update subsets is exponential, these strategies avoid search-space explosion, making candidate M-set generation robust to the size of the micro-update library L.

Even though each candidate M-set generated by the DFS transforms the assignment σ_{data} into σ'_{data}, some of these candidates may be spurious, as we now illustrate.

Example 2 (Spurious M-sets): Consider the trace in Fig. 6. For the third transition (`counter`=2 to 3), the `lb_refill` micro-update (Fig. 4B) correctly updates the LB signals. However, the `lb_hold` micro-update (not shown) which maintains the same values would would also work for this cycle since the signals do not change. The DFS procedure will generate both of these as candidates. However, we note that in the design (Fig. 3) the LB is only refilled when the **`mem`** stage instruction, `inst_mem`, is a load. This is not the case for this transition since `inst_mem` is an `addi` instruction. Hence, `lb_refill` is a spurious candidate which we need to filter out.

3) Eliminating spurious M-sets: We eliminate such spurious M-sets using (a) distinguishing oracles (§IV-D3) and (b) cover properties (§IV-D3).

***Using a distinguishing oracle**:* A distinguishing oracle [33] identifies a test program (and its corresponding trace) on which two given M-sets, $M_{1,2}$, differ in their behaviour. Running M-set generation on distinguishing traces only generates one of the two M-set candidates, eliminating ambiguities such as the one in Example 2.

Example 2 (continued): If applied to the M-sets M_1 and M_2 which include micro-updates `lb_hold` and `lb_refill` respectively, a distinguishing oracle generates a trace which includes the first transition (`counter`=0 to 1) of Example 2. We note that only the `lb_hold` micro-update applies for this transition (since the LB entry remains invalid). The distinguishing oracle can also generate a trace with the second transition (`counter`=1 to 2), where only `lb_refill` applies. The distinguishing oracle can be used to specifically identify traces where M_1 (or M_2) is applicable and the other is not.

Adapting the technique from [33], a distinguishing oracle can be implemented by invoking a hardware model checker (e.g. SymbiYosys [19]) as follows. Given candidate M-sets M_1, M_2, we create two copies of the micro-update model. The first copy invokes micro-updates from M_1 while the second from M_2. Then we invoke the hardware model checker to generate a trace such that at some cycle i, the two copies have equal values *of all used signals*, while at the next cycle

$(i+1)$, M_1 and M_2 generate different values for *at least one modified signal*. Since M_1 and M_2 generate different values for some signal, only one of these can be a valid candidate M-set for the transition from cycle i to $i+1$. Thus the trace generated by the model checker distinguishes M_1 and M_2.

A typical case (§V-B) where the distinguisher helps is when a functional unit operates on one of several possible inputs (e.g. in the case of bypassing). In such cases, one can identify traces in which a specific bypass path is invoked.

Using cover properties: For a Boolean formula ϕ, cover properties of the form $\texttt{cover}(\phi)$ can be supplied to a hardware verification tool (e.g. JasperGold, SymbiYosys [19]). The tool, for a given cover property ϕ, aims to generate an execution in which ϕ holds. While the distinguishing oracle is useful when identifying traces on which specific M-sets do/do not apply, cover properties can be used to identify traces where certain guard predicates do/do not evaluate to true. Traces satisfying the cover property can then be added to the trace corpus for future M-set generation. This allows the generation of rare executions that were not seen during the initial simulation.

Example 3 (Using cover properties): Consider a simple 3-stage pipeline where the trigger signals correspond to the three instructions in the pipeline: $\mathsf{S}_{trig} = \{i_{fet}, i_{exe}, i_{wb}\}$. Consider a bypassing path micro-update that is triggered when there is a data dependency between i_{exe} and i_{fet}, and i_{exe} does not write to zero. The guard corresponding to this micro-update is:

$$\phi_{bypass} \equiv (\texttt{rd}(i_{exe}) = \texttt{rs1}(i_{fet})) \; \&\& \; (\texttt{rd}(i_{exe}) \neq \texttt{5'b00000})$$

Since this condition requires matching register source/destination fields, the probability of it holding is somewhat low ($\sim 1/32$), and it may be missed during random simulation (§IV-D1). In such cases, the lack of example transitions for the bypassing path would result in the micro-update not being represented in the generated model. The cover property $\texttt{cover}(\phi_{bypass})$ generates an execution in which this path is triggered, mitigating this. Thus, the user can generate rare executions not observed in simulation.

E. Guard synthesis

The M-sets generated in the previous stage indicate micro-update invocations that specifically produce the simulation traces. However, we now want to generalize this to all traces by synthesizing the guards (§III-A3). This generalization results in the final micro-update model satisfying Property 1.

We formulate guard synthesis using Syntax Guided Synthesis (SyGuS) which is a type of formal synthesis. The formal synthesis problem aims to generate an implementation of an object (typically a function) such that the generated implementation satisfies some given specification. SyGuS [6] builds on this core idea by restricting the search space of synthesis function implementations to terms from a context-free grammar and where the specification for the synthesis function is provided in the SMT-LIB format [11], [12]. SyGuS is supported by multiple synthesis solvers (we use CVC5 [10]).

We formulate the guard G_{t} for micro-update t as a synthesis function over the trigger signals. For guard G_{t} we extract pairs (σ^i_{trig}, b^i) from the simulation traces and corresponding M-sets. Here, σ^i_{trig} is the trigger signal assignment at step i, and b^i is a Boolean representing whether t belongs to a valid M-set at step i. We ensure that the synthesized guard is consistent with these pairs through the following synthesis constraints.

The first constraint is that for step i, if b_i is false (meaning that micro-update t wasn't a part of any valid M-set), then the guard G_{t} must evaluate to false on σ^i_{trig}: $\bigwedge_i (b^i \vee \neg G_{\mathsf{t}}(\sigma^i_{trig}))$. If not, the generated model would be incorrect at step i.

Additionally, for each signal $\mathsf{s} \in \mathsf{S}_{data}$, we require that exactly one micro-update modify s at each step. We define the set of micro-updates that modify s as $\mathsf{m(s)}$. The following formula enforces the "exactly one" constraint for signal s: $(\bigwedge_{\mathsf{t}\neq\mathsf{t}'\in\mathsf{m(s)}}(\neg G_{\mathsf{t}}(\sigma_{trig}) \wedge \neg G_{\mathsf{t}'}(\sigma_{trig}))) \wedge (\bigvee_{\mathsf{t}\in\mathsf{m(s)}} G_{\mathsf{t}}(\sigma_{trig}))$.

On posing this problem to a SyGuS solver, the solver returns expressions for each guard G_{t} in terms of the trigger signals. This guard expression, together with the micro-update bodies, gives us a complete model. The inability of the solver to synthesize some guard implies that the set of predicates was insufficient, i.e. there are factors outside this set that the guard depends on. The user can then try strategies such as adding more trigger signals, or increasing the SyGuS grammar depth.

F. Equivalence checks

Finally the generated model is checked against the source design for S_{data}-equivalence (Property 1). If the equivalence proof fails, we get back a counterexample trace. This trace is fed back into the trace database and the synthesis loop is repeated. Future iterations of synthesis use this new trace, and hence avoid synthesizing the same (incorrect) model.

G. Hierarchical synthesis

The approach discussed so far monolithically generates micro-update models from the source RTL. However, our approach can benefit from the hierarchy inherent to RTL designs, which is what we now discuss. Consider a micro-update model with signals S_1 generated from a design component C_1. Now suppose we want to generate a model with signals S for a (larger) component C, of which C_1 is a sub-component. By reusing the model for C_1, we only need to generate the slice of the model corresponding to signals $\mathsf{S} \setminus \mathsf{S}_1$ (i.e. signals in S but not S_1). In particular, this requires generating guard predicates only for micro-updates relevant to $\mathsf{S} \setminus \mathsf{S}_1$. This decomposition leads to smaller synthesis queries, thus improving performance.

Hierarchical synthesis, however, requires that (a) the signals S_1 from the sub-component (C_1) not be directly driven by logic outside the component, and (b) that the trigger signals for C_1 map to trigger signals for the parent component C in a straightforward way. Here (a) ensures that S_1 signals preserve their behaviour in the larger model, and (b) ensures that the guards from C_1 in terms of trigger signals from C. We demonstrate an application of this in §V-C2, where we first generate a model for the `store_unit` of `cva6`

(C_1). Then we reuse it when generating a model for the `load_store_unit` (C), of which the the `store_unit` is a sub-component.

V. EXPERIMENTAL EVALUATION

A. Methodology

We implement our framework in a Python-based tool called PAUL (Python-based Atomic-Update Language toolkit). The user specifies signals-of-interest (§IV-C1), a library of micro-updates over these signals (§IV-C2), and a set of predicates for the guards. The tool allows the user to simulate the design over input programs, generate M-sets (§IV-D), and synthesize guards (§IV-E).

Hyperparameters: While Fig. 5 indicates the general synthesis pipeline, in practice, there are knobs that can be used to guide the synthesis. Both the distinguishing (§IV-D3) and cover (§IV-D3) trace generation require calls to a model checker, which can be time-consuming. The user can selectively apply these checks. For the distinguishing oracle, this amounts to choosing two M-sets to distinguish between, while for the cover property oracle, this amounts to providing a (possibly partial) predicate valuation.

Backend setup: PAUL interfaces with iverilog [67] for simulation, and the Yosys [68] based SymbiYosys [19] for model checking. SymbiYosys, uses Boolector [49] and ABC [5] as backend solvers. We perform experimentation on an Intel Core-i7-10610U processor at 2.3GHz with 16GB RAM.

Evaluation overview: We conduct two case studies by synthesizing micro-update models for: (a) `Sodor5Stage` (§V-B) and (b) components from `cva6` (§V-C1). We demonstrate how the hierarchical lifting of micro-update models (§V-C2) can improve scalability. We showcase applications of the lifted models for security verification of software. We demonstrate better performance compared to verification against the source RTL, and strong soundness guarantees (which are often lacking in hand-written models).

B. Case Study: The `Sodor5Stage` processor

The `Sodor5Stage` is a 5-stage in-order processor [62] with a simple scratchpad memory. We augment this design with the load buffer (LB) feature (as discussed in §II). We now discuss the results of model lifting, referring to Table I.

We generate three models from the processor corresponding to different ISA subsets (rows in Tab. I): ALUI (ALU immediate), ALUR (ALU register), and ALULS (ALU immediate + memory instructions). In each case, we identify a set of signals (S_{data}) and the library of micro-operations (L). In all models, we have a queue of the previous five instructions as the control state. Since this is a 5-stage design, this suffices to represent all inter-instruction interactions.

M-set generation: In each case, we perform simulation over the respective instruction subsets by constraining the opcode. We extract the transitions and generate the candidate micro-update M-sets for these simulation traces. We observe that for the comparison instructions (e.g. `slt`), comparisons of several bypassing-path signals have the same (0/1) result leading to spurious M-sets. We provide the counts of distinguishing and cover traces used to eliminate these spurious cases in Table I. As M-sequence generation does not involve (costly) solvers, it requires much less time (¡5s) as compared to guard synthesis.

Guard synthesis and equivalence: In the second phase of the synthesis, we use the M-sets to synthesize guards by formulating a SyGuS [6] query. We use cvc5 [10] as the SyGuS solver. Finally, we check the candidate micro-update model for S_{data}-equivalence with respect to the source RTL. We perform this check using a bounded model checking (BMC) query with SymbiYosys ([19]) model checker. We use a depth of 15 steps for BMC which ensures that all possible inter-instruction interactions across the pipeline are explored.

1) Do lifted models aid software security analysis?: In this section, we explore whether/how lifting micro-update models improves the trustworthiness and performance of security analysis. We begin by recalling the Bounds-Check-Bypass (BCB) gadget from the Spectre vulnerability [37], [38].

Spectre BCB background: Figure 7(A) shows the Spectre BCB gadget and Fig. 7(B) shows assembly fragments for the `array2` access in the gadget. Fig. 7(C) shows the assembly fragment after a minor modification to the gadget (changing `&=` to `=`). In B (but not C), two ALU instructions separate the `lw` instruction (`101ba`) from the `sw` instruction (`101c4`). Inlays in Fig. 7 depict timing diagrams for executions of B,C on `Sodor5Stage` with the LB modification, under the $\mathtt{sw}_{\mathtt{exe}}$ implementation. In B, $\mathtt{sw}_{\mathtt{exe}}$ flushes the LB *after* the `lw` refills it. However, in C, $\mathtt{sw}_{\mathtt{exe}}$ flushes LB *before* the refill. If the `lb_addr` were *attacker observable* (as discussed in §II-B), B would be safe, while C would not.

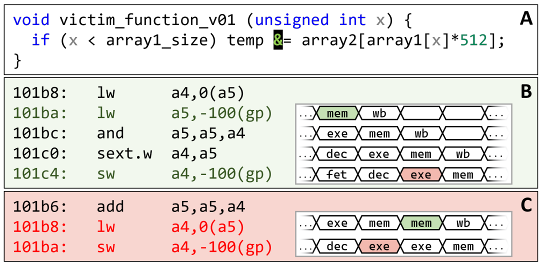
This example shows how small changes to the microarchitecture such as the LB can render the hand-written abstract models adopted by existing approaches [9], [17], [22], [27] imprecise. As an example, the abstract model from [17] which is designed to check Spectre variants would flag both Fig. 7(B), (C) as vulnerabilities, leading to false positives. However, as we now demonstrate, the lifted micro-update model can distinguish between these outcomes and specifically flag the offending (C) case, thus resulting in more trustworthy analysis.

Semantic information-flow experiment: We now demonstrate how the lifted (ALULS) model can be used to perform semantic information-flow analysis on software. We check whether there exists an information-flow from certain source (*src*) signals to the `lb_addr` signal when executing small litmus test programs. Note that `lb_addr` is assumed to be adversary-observable (§II-B). If this check passes, it guarantees that victim secrets (e.g. private keys) are not leaked through the LB side-channel when the test program is executed. We formulate information-flow as a variant of the non-interference [20] hyperproperty. We check this by invoking the SymbiYosys model checker.

Table II draws a comparison of the run times of these information-flow checks when performed against the lifted model and the source design. Each check (row in Tab. II) is

Model slice	Signals-of-interest (S_{data})	$\lvert\mathsf{S}_{data}\rvert$	$\lvert\mathsf{L}\rvert$	Simulation examples	Distinguish/ cover examples	Guard synthesis	Equivalence proof ($d = 15$)
I-type ALU	$S_1 =$ decode, register file I/O, ALU, bypassing	15	25	190	1	1m1s	10m9s
R-type ALU	$S_1 =$ decode, register file I/O, ALU, bypassing	15	29	190	3	2m14s	11m57s
ALU + `lw` + `sw`	$S_2 = S_1 \cup$ memory ports, LB	20	37	190	8	57m	34m24s

TABLE I: Model parameters, example counts, synthesis and equivalence check times for the generated models for `Sodor5Stage`.



```
void victim_function_v01 (unsigned int x) {
  if (x < array1_size) temp &= array2[array1[x]*512];
}
```

```
101b8:   lw       a4,0(a5)
101ba:   lw       a5,-100(gp)
101bc:   and      a5,a5,a4
101c0:   sext.w   a4,a5
101c4:   sw       a4,-100(gp)
```

```
101b6:   add      a5,a5,a4
101b8:   lw       a4,0(a5)
101ba:   sw       a4,-100(gp)
```

Fig. 7: Subtle program changes can expose vulnerabilities: (A) Ex. 1 from Kocher's [37] examples that is vulnerable to a Spectre [38] attack. (B, C) Compiled assembly from the original example, and the example after changing the `&=` to `=`. The accompanying waveforms show how (C) remains tainted under $\texttt{sw}_{\texttt{exe}}$ while (B) does not.

Symbolic instruction sequence (testcase)	Source signals (*src*)	Outcome Safe/Unsafe	Design runtime	Model runtime
`nop lw`	`mem`	Safe	2m43s	8s
`nop lw`	`mem, regs`	Unsafe	3m50s	11s
`nop alui lw`	`mem`	Safe	2m37s	12s
`nop sw lw`	`mem`	Safe	2m30s	9s
`nop lw lw`	`regs`	Unsafe	4m15s	8m51s
$\texttt{nop(lw}+\texttt{ alui)}^2\texttt{ sw}$	`mem, regs`	Safe	4m49s	38s
$\texttt{nop(lw}+\texttt{ alui)}^3\texttt{ sw}$	`mem, regs`	Safe	5m11s	1m31s

TABLE II: Comparison of information-flow proof run times over some representative testcases. Tests are over *symbolic* sequences of instructions represented as a regular expression in the first column. The source run time is over the `Sodor5Stage` RTL (with $\texttt{sw}_{\texttt{mem}}$) while model run time is over our lifted ALULS model.

performed on a 1-4 instruction symbolic litmus tests (where the opcode is fixed and other fields are unconstrained). We observe that verification against the lifted model results in improved performance over the source RTL in most cases. We can guarantee the soundness of this analysis since the lifted model preserves S_{data}-equivalence with the source model (Property 1). *This demonstrates that model lifting can improve the trustworthiness of security analysis over hand-written models while providing performance improvements over the source design.*

C. Case Study: The `cva6` *(Ariane) processor*

In this section, we apply our lifting methodology to components from the `cva6` (Ariane) processor [1]. Ariane is a 6-stage application-class processor implementing the RISC-V 64-bit instruction set. We focus on lifting four memory-facing modules in Ariane: TLB, write-buffer, store-unit, and load-store-unit. Micro-architectural units involved in memory operations (e.g. load/store buffers) are especially prone to side-channel exploits. For instance, MDS attacks (e.g. [16], [55], [63], [64]) swipe in-flight data from such buffers. Hence accurately modelling such units when performing security analysis is essential.

This case study explores two key questions: (1) in §V-C2 we investigate whether hierarchical synthesis (§IV-G) can improve the scalability of lifting and (2) in §V-C3 we apply the lifted models to perform security analysis, and demonstrate significant improvements to verification runtime.

We now briefly describe the components we lift (for details see [1], [47]). The TLB [8], [52] is housed inside the memory management unit (MMU) of the processor and uses a PLRU (pseudo-least-recently-used) eviction scheme [51]. The write-buffer (`wbuffer`) is a coalescing buffer for the write-through data-cache, and sits between the core, data-cache and data-memory. The store unit (`store_unit`) maintains a queue of pending store requests from the core. Requests are initially queued into a speculative queue and are later moved to a commit queue (upon receiving a commit signal from the core). The load store unit (`load_store_unit`) consists of the load and store unit sub-modules. It only handles one request at a time and stalls if there is a load-after-store conflict with an unfulfilled store operation (at the same address).

1) Generating micro-update models for `cva6`*:* We now discuss the generation of micro-update models for the aforementioned components, summarizing results in Tab. III.

TLB and **`wbuffer`***:* For the TLB and `wbuffer`, we generate micro-update models in which the signals-of-interest (S_{data}) include all entries in these buffers. For example, each `wbuffer` entry has three 1-bit signals representing its state: s = (`valid`, `dirty`, `txnblk`). The micro-update library L consists of five micro-updates that: (1, 2) receive a fresh write request or a replayed request to the same address, (3, 4) initiate or conclude a memory transaction, and (5) a nop (no operation). As reported in Tab. III while guard synthesis takes the maximum time of all the steps, the overall generation time of either model is less than ∼4m.

`store_unit` ***and*** **`load_store_unit`***:* The load-store unit is much more complex than the TLB and `wbuffer`, must handle inter-instruction dependencies while interfacing between the core (for requests/commits) and the memory (for requests/responses). This is reflected in the micro-update model synthesis times (Tab. III). While the `store_unit` can be synthesized monolithically, monolithic guard synthesis for the `load_store_unit` hits a time out († in Tab. III) of 1 hr.

2) Can hierarchical synthesis help improve scalability?: Since monolithic synthesis does not scale, we attempt hierar-

Module	Extracted signals-of-interest (S_{data})	M-set generation	Guard synthesis	Equivalence proof ($d = 15$)
TLB	states of buffer entries	¡1s	4s	6s
`wbuffer`	states of buffer entries (`valid`,`dirty`,`txnblk`)	1s	2m47s	1m10s
`store_unit`	store queue, store req. states	1s	5m17s	2m58s
`load_store_unit`	store queue, store req. states, load req. state (e.g. `valid`, `spec`, `commit`, `memresp`)	2s	TO (†)	1m49s
		2s	11m38s ($\star$)	

TABLE III: Summary of the micro-update generation from the components of `cva6`. The row marked with (†) denotes monolithic synthesis, while for ($\star$) we use hierarchical synthesis (§IV-G), by reusing the `store_unit` (sub-module) generated previously.

Symbolic instruction sequence (testcase)	Constraint on the testcase program	Safe/ UnSafe	Design runtime	Model runtime
`alui sw lw alui`	NONE	US	34s	17s
`alui sw lw alui`	`regs` equal (RE)	S	3m57s	48s
`alui sw lw alui`	addr(`sw`) == addr(`lw`)	S	1m16s	34s
$\texttt{lw}_1$ `sw` $\texttt{lw}_2$ `alui`	NONE	US	1m29s	25s
$\texttt{lw}_1$ `sw` $\texttt{lw}_2$ `alui`	RE	US	1m10s	27s
$\texttt{lw}_1$ `sw` $\texttt{lw}_2$ `alui`	addr(`sw`) == addr($\texttt{lw}_2$)	S	54s	24s
$\texttt{lw}_1$ `sw` $\texttt{lw}_2$ `alui`	RE, $\mathrm{rd}(\texttt{lw}_1) \neq \mathrm{rs1}(\texttt{sw})$	US	1m	28s
$\texttt{lw}_1$ `sw` $\texttt{lw}_2$ `alui`	RE, $\mathrm{rd}(\texttt{lw}_1) \neq \mathrm{rs1}(\texttt{sw}) \wedge$ $\mathrm{rd}(\texttt{lw}_1) \neq \mathrm{rs1}(\texttt{lw}_2)$	S	1m48s	44s
$\texttt{lw}_1$ `sw` $\texttt{lw}_2$ $\texttt{lw}_3$	RE, $\mathrm{rd}(\texttt{lw}_1) \neq \mathrm{rs1}(\texttt{sw}) \wedge$ $\mathrm{rd}(\texttt{lw}_1) \neq \mathrm{rs1}(\texttt{lw}_2)$	US	9m4s	1m2s
$\texttt{lw}_1$ `sw` $\texttt{lw}_2$ $\texttt{lw}_3$	RE, $\mathrm{rd}(\texttt{lw}_1) \neq \mathrm{rs1}(\texttt{sw}) \wedge$ $\mathrm{rd}(\texttt{lw}_1) \neq \mathrm{rs1}(\texttt{lw}_2) \wedge$ $\mathrm{rd}(\texttt{lw}_1) \neq \mathrm{rs1}(\texttt{lw}_3)$	S	12m36s	1m29s

TABLE IV: Security analysis checking whether the test cases result in different timing behaviours. "Design" and "Model" runtimes are for the `load_store_unit` RTL and lifted model, respectively.

chical synthesis (§IV-G) to generate a micro-update model for the `load_store_unit`. Hierarchical synthesis is feasible since the two requirements outlined in §IV-G are satisfied: (a) `store_unit` signals are not directly written to by outside logic and (b) almost all inputs to the `load_store_unit` are directly passed to the `store_unit`. The exception to (b) is an input signalling whether the incoming request into the `load_store_unit` is a load/store. We needed to condition it being a store before passing it to the `store_unit`. By adopting the previously (monolithically) synthesized `store_unit`, we only needed to synthesize the non-`store_unit` guards. While monolithic synthesis timed out, we could generate the model using the hierarchical approach in ~12m ($\star$ in Tab. III).

Hierarchical synthesis is generally suitable when the composition is performed at module boundaries of the RTL. Such cases tend to satisfy condition (a) from §IV-G. However, as seen in this case, condition (b) from §IV-G may be harder to satisfy if there is intermediate logic between the trigger signals of the parent module and those of submodules. While the logic was manually identifiable in this case, in the future one could use a version of guard synthesis to extract it. *Thus hierarchical lifting leads to better scalability when synthesizing complex models. However, this might require the user to provide additional instrumentation/hints.*

3) Does performance of security analysis improve with the micro-update model compared to source RTL?:

a) Experimental setup: We investigate whether using the lifted model can improve the performance of security verification. However, since we only lifted a model corresponding to the `load_store_unit`, we first wrap the generated model in a simple processor shim. This shim executes `alui`, `lw`, and `sw` instructions. While `alui` instructions are executed locally, the shim interfaces with the `load_store_unit` for `sw` and `lw` instructions. Execution is stalled if the `load_store_unit` is busy. We also wrap the source `load_store_unit` RTL in an identical shim for a valid comparison. We now conduct an experiment to compare security analysis run times when using the (shim-wrapped) lifted model/source `load_store_unit` RTL as the underlying HW platform (similar to §V-B1).

b) The `load_store_unit` timing channel and results: In our experiment, we analyze whether a given a software instruction sequence is vulnerable to a `load_store_unit`-based timing channel. We formulate *invulnerbility* as a non-interference [20] property stating that the timing behaviour of the instruction sequence is independent of victim data (we assume that the victim's secret resides in data memory). We check this property by invoking the SymbiYosys model checker. If the check passes, the instruction sequence is secure while a counterexample would indicate a vulnerability.

We present our results in Tab. IV, where rows denote the symbolic test cases that are checked. The test cases are based on read gadgets seen in hardware attacks. *We observe that verification run times with the lifted model are up to 8× lower than the source RTL.* This tends to increase for larger tests.

We also note that certain test programs are unsafe (US). This is the case since the `load_store_unit` blocks loads when there are previous pending stores at the same address. Thus, the timing behaviour of $\texttt{sw} \cdot \texttt{lw}$ when the `lw` and `sw` are on the same address is different than when they are on different addresses. If the address constitutes a victim secret, an attacker could infer the secret through a timing-based attack [24].

This demonstrates that models lifted from security-relevant design components can be used to check the existence of vulnerabilities in SW with significant performance improvement.

Evaluation highlights: Our experiments demonstrate: (a) the feasibility of micro-update model lifting, and the ability to improve scalability through hierarchical synthesis, and (b) the application of the lifted models to perform security analysis of software with greater reliability than handwritten models and better performance than with source RTL.

VI. DISCUSSION AND LIMITATIONS

Manual effort: The main manual effort required in our approach is for identifying signals-of-interest (S_{data}), the micro-update library (L) and design instrumentation. The first is fundamental to our framework as S_{data} captures the design slice the user is interested in analyzing. While the current approach requires a user-specified micro-update library, going forward we foresee automating this by utilizing techniques such as specification/invariant-mining [21], [39]. A large chunk of effort in our case studies was in understanding and instrumenting the designs. This can be reduced if lifting is performed in lockstep with the design phase/with the aid of designers.

Signals-of-interest and analysis coverage: While we require the user to identify the signals-of-interest, automatically identifying the complete attack surface is a major open challenge. Prior work on analyzing hardware exploits also rely on manually identified signals (e.g. [9], [17], [27], [48], [61]). Through the signal-of-interest (S_{data}), our approach exposes a tradeoff between model coverage and scalability. While a model capturing with smaller S_{data} is less detailed, it can still be used to prove security against multiple attack scenarios targeting those signals (e.g. in §V-C3 we study security implications of a `load_store_unit` against several instruction sequences). For generating a model with high coverage, hierarchical synthesis (§IV-G) can improve scalability.

VII. RELATED WORK

Several approaches propose formal specifications/models at the ISA-level, such as the RISC-V specification [66], as well as formal models of ISA-level semantics (e.g. in SAIL [26] and Kami [35]). Approaches such as instruction-level-abstraction (ILA) [30], [31], [71], [72] extend these models to include additional architectural state (e.g. accelerator state). ILA-MCM [72] also considers a model which is composed of code blocks, however, these are composed axiomatically. These models have been extensively used for specification [7] and verification of processor designs (e.g. [14], [54], [57], [65]). There is also work on synthesizing such models from hardware [59]. However, as discussed in §II-C, these approaches are not precise enough to model microarchitecture-level interactions, which is essential for accurate security analysis.

There is work on developing microarchitectural models of hardware, such as μspec [43]. μspec has been used to model memory consistency [42], coherence [45], and security properties [61]. More recently, there has been work on generating μspec models from RTL [29]. [48] uses a μspec based model to detect vulnerabilities. As discussed in §II-D, these models have difficulty capturing functional details necessary for checking semantic security properties.

Approaches that define [28] and verify [9], [17], [22], [27] security properties from a software standpoint manually develop platform models for program execution. This is error-prone due to subtle microarchitectural interactions. In §V-B1 we demonstrated how micro-update models generated with strong guarantees can increase the level of assurance of these approaches. [70] develop a speculative platform model with a guarantee of invulnerability to some attacks, while [13] develops a methodology to validate platform models.

UPEC [23] performs verification of RTL by checking whether executing read gadgets (e.g. [36], [38]) leads to a security vulnerability. This is orthogonal since we lift models from RTL for scalable software verification. In particular, a UPEC-like approach could be soundly performed on the lifted model owing to our equivalence guarantee (§IV-B). Moreover, as our results indicate, verification against the lifted model is more performant as compared to the RTL.

Micro-updates bear resemblance to rules from BlueSpec [50] or transactions from TLM [15]. However, our focus is modelling and lifting as opposed to hardware design.

VIII. CONCLUSION

In this work, we proposed micro-update models as a formalism for developing abstract formal models of the microarchitecture. By accurately preserving security-relevant microarchitectural detail, micro-update models provide an abstract yet sound substrate for performing scalable security verification of software. To address the challenge in hand-writing formal models, we developed a semi-automated technique to hierarchically synthesize micro-update models from RTL. We evaluated our approach by synthesizing models from the `Sodor5Stage` and `cva6` processors. We demonstrated how the generated models can be used to soundly perform semantic security analysis, with improved performance over the source RTL. Our modelling and lifting framework can allow HW designers to increase the level of assurance of security analysis while reducing the efforts involved in formal model development.

ACKNOWLEDGEMENTS

This work was supported by DARPA contract FA8750-20-C0156, NSF grant 1837132 and a gift from Intel Corporation.

REFERENCES

[1] CVA6 (Ariane) User Manual. https://docs.openhwgroup.org/projects/cva6-user-manual/index.html.

[2] CVA6 RTL Bug Issues. https://github.com/openhwgroup/cva6/issues?q=is%3Aissue+label%3AType%3ABug+label%3AComponent%3ARTL+is%3Aopen. Accessed: 2023-02-20.

[3] Maaz Bin Safeer Ahmad and Alvin Cheung. Optimizing data-intensive applications automatically by leveraging parallel data processing frameworks. *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.

[4] Maaz Bin Safeer Ahmad and Alvin Cheung. Automatically leveraging MapReduce frameworks for data-intensive applications. *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[5] Alan Mischenko et al. Berkeley ABC tool. https://github.com/berkeley-abc/abc, 2022.

[6] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.

[7] Alasdair Armstrong, Thomas Bauereiß, Brian Campbell, Alastair David Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian David Bede Stark, Neelakantan R. Krishnaswami, and Peter Sewell. Isa semantics for armv8-a, risc-v, and cheri-mips. *Proceedings of the ACM on Programming Languages*, 3:1 – 31, 2019.

[8] Remzi H. Arpaci-Dusseau. Operating systems: Three easy pieces. *login Usenix Mag.*, 42, 2017.

[9] Musard Balliu, Mads Dam, and Roberto Guanciale. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[10] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength smt solver. In *TACAS*, 2022.

[11] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[12] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, 2009.

[13] Pablo Buiras, Hamed Nemati, Andreas Lindner, and Roberto Guanciale. Validation of side-channel models via observation refinement. *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[14] Jerry R. Burch and David L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, 1994.

[15] Lukai Cai and Daniel Gajski. Transaction level modeling: an overview. *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721)*, pages 19–24, 2003.

[16] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[17] Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. A formal approach to secure speculation. *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 288–28815, 2019.

[18] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: a platform for high-level parametric hardware specification and its modular verification. *Proceedings of the ACM on Programming Languages*, 1:1 – 30, 2017.

[19] Claire Wolf, et. al. Symbiyosys. https://github.com/YosysHQ/sby, 2022.

[20] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *2008 21st IEEE Computer Security Foundations Symposium*, pages 51–65, 2008.

[21] Michael D. Ernst and David Notkin. Dynamically discovering likely program invariants. 2000.

[22] Xaver Fabian, Marco Guarnieri, and Marco Patrignani. Automatic detection of speculative execution combinations. *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[23] Mohammad Rahmani Fadiheh, Johannes Müller, Raik Brinkmann, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[24] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8:1–27, 2016.

[25] Joseph A. Goguen and José Meseguer. Unwinding and inference control. *1984 IEEE Symposium on Security and Privacy*, pages 75–75, 1984.

[26] Kathryn E. Gray, Gabriel Kerneis, Dominic P. Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-isa architectural envelope definition, and test oracle, for ibm power multiprocessors. *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 635–646, 2015.

[27] Marco Guarnieri, Boris Köpf, José Francisco Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2020.

[28] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1868–1883, 2021.

[29] Yao Hsiao, Dominic P. Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. Synthesizing formal models of hardware from rtl for efficient verification of memory model implementations. *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[30] Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, and Sharad Malik. ILAng: A modeling and verification platform for SoCs using instruction-level abstractions. In *TACAS*, 2019.

[31] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. Instruction-level abstraction (ila): A uniform specification for system-on-chip (soc) verification. *ArXiv*, abs/1801.01114, 2018.

[32] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. Instruction-level abstraction (ila). *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 24:1 – 24, 2019.

[33] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 1:215–224, 2010.

[34] Susmit Jha and Sanjit A. Seshia. A Theory of Formal Synthesis via Inductive Learning. *Acta Informatica*, 54(7):693–726, 2017.

[35] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016.

[36] Vladimir Kiriansky and Carl A. Waldspurger. Speculative buffer overflows: Attacks and defenses. *ArXiv*, abs/1807.03757, 2018.

[37] Paul Kocher. Spectre mitigations in microsoft's c/c++ compiler, 2018.

[38] Paul C. Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Michael Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[39] Wenchao Li, Alessandro Forin, and Sanjit A. Seshia. Scalable specification mining for verification and diagnosis. *Design Automation Conference*, pages 755–760, 2010.

[40] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul C. Kocher, Daniel Genkin, Yuval Yarom, and Michael Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018.

[41] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 605–622. IEEE Computer Society, 2015.

[42] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Pipecheck: Specifying and verifying microarchitectural enforcement of memory consistency models. *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 635–646, 2014.

[43] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. Coatcheck: Verifying memory ordering at the hardware-os interface. *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[44] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. Rtlcheck: Verifying the memory consistency of rtl designs. *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 463–476, 2017.

[45] Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Ccicheck: Using μhb graphs to verify the coherence-consistency interface. *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 26–37, 2015.

[46] Zohar Manna and Richard J. Waldinger. A Deductive Approach to Program Synthesis. In *TOPL*, 1979.

[47] Valentin Martinoli, Yannick Teglia, Abdellah Bouagoun, and Régis Leveugle. CVA6's Data cache: Structure and Behavior, 2022.

[48] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. Axiomatic hardware-software contracts for security. *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022.

[49] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *J. Satisf. Boolean Model. Comput.*, 9(1):53–58, 2014.

[50] Rishiyur S. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.*, pages 69–70, 2004.

[51] David A. Patterson and John L. Hennessy. Computer architecture: A quantitative approach. 1969.

[52] David A. Patterson and John L. Hennessy. Computer Organization and Design, Fifth Edition: The Hardware/Software Interface. 2013.

[53] Elizabeth Polgreen, Andrew Reynolds, and Sanjit A. Seshia. Satisfiability and synthesis modulo oracles. In *VMCAI*, 2022.

[54] Alastair David Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Mustafa Zaidi. End-to-end verification of arm ® processors with isa-formal. 2016.

[55] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[56] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, H. Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[57] Jens Ulrik Skakkebæk, Robert B. Jones, and David L. Dill. Formal verification of out-of-order execution using incremental flushing. In *CAV*, 1998.

[58] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS XII*, 2006.

[59] Pramod Subramanyan, Bo-Yuan Huang, Yakir Vizel, Aarti Gupta, and Sharad Malik. Template-based parameterized synthesis of uniform instruction-level abstractions for soc verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37:1692–1705, 2018.

[60] Tachio Terauchi and Alexander Aiken. A capability calculus for concurrency and determinism. In *TOPL*, 2008.

[61] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Security Verification via Automatic Hardware-Aware Exploit Synthesis: The CheckMate Approach. *IEEE Micro*, 39:84–93, 2019.

[62] UCB-BAR. Sodor processor collection. https://github.com/ucb-bar/riscv-sodor, 2013.

[63] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-Flight Data Load. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105, 2019.

[64] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. *2021 IEEE Symposium on Security and Privacy (SP)*, pages 339–354, 2021.

[65] Miroslav N. Velev and Randal E. Bryant. Superscalar processor verification using efficient reductions of the logic of equality with uninterpreted functions to propositional logic. In *CHARME*, 1999.

[66] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. The risc-v instruction set manual. 2014.

[67] Steven Williams. iverilog. https://github.com/steveicarus/iverilog, 2022.

[68] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. 2013.

[69] Wenjie Xiong and Jakub Szefer. Leaking information through cache lru states. *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 139–152, 2020.

[70] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data. *IEEE Micro*, 40:81–90, 2020.

[71] Yu Zeng, Aarti Gupta, and Sharad Malik. Automatic generation of architecture-level models from rtl designs for processors and accelerators. *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 460–465, 2022.

[72] Hongce Zhang, Caroline Trippel, Yatin A. Manerkar, Aarti Gupta, Margaret Martonosi, and Sharad Malik. Ila-mcm: Integrating memory consistency models with instruction-level abstractions for heterogeneous system-on-chip verification. *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, 2018.