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
Copyright © 2023, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

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
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:

\[
\text{canLoad} \equiv (\text{loadAddr} == \text{addr}) \land \text{valid}
\]

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.

B. **The \textit{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 `sw\_{exe}` and `sw\_{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 `sw\_{exe}` and `sw\_{mem}` implementations. In the case of `lw;sw\_{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 `lw;sw\_{mem}` (Fig. 2 bottom), the LB entry is flushed by the `sw` after the refill caused by the `lw`. Consequently, `lw;sw\_{exe}` leads to an LB which is tainted (due to the `lw`), while `lw;sw\_{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 `lw;sw\_{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 `lw;sw\_{mem}` sequence leads to a sanitized LB, (i.e. the LB is untainted), a timing measurement cannot reveal victim secrets.
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 microarchitectural level and exploit software-invisible features (e.g. caches, buffers, predictors). Consequently, 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, $\sigma_{arch}$, 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):

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

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

$$\sigma_{micro} [i_1; i_2] \sigma'_{micro} \iff \sigma_{micro} [i_1] \sigma''_{micro} [i_2] \sigma''_{micro}$$

In the LB example, the semantics of both $sw_{exe}$ and $sw_{mem}$ 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 $sw_{exe}$ and $sw_{mem}$ 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 $sw_{exe}$ and $sw_{mem}$ 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 $\mu$spec [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, $\mu$spec can express behaviours of instructions executing simultaneously (as in a pipeline), and does not suffer from the isolation problem discussed earlier.
A micro-update model can differentiate between cases for a micro-update (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 (§II-B).

### 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 \( \sigma_{data} : S_{data} \rightarrow \mathcal{V} \).

2) **Micro-updates**: A micro-update is an imperative code-block 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 \( t \in L \) provides the semantics \( t \), which defines how \( t \) transforms the \( S_{data} \) signals when invoked. An invocation 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** \( S_{strig} \); \( G_t(S_{strig}) \in \{true, false\} \). The micro-update \( t \) is executed at a cycle iff \( G_t \) evaluates to true (and \( G_{\text{false}} \)).

### 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_iotype is executed when instr_queue[1], the decoded instruction, is an I-type instruction). While instr_queue[1] depends on a single instruction input parameter, guards can depend on multiple inputs. For example, if \( rsl_{addr}(\text{instr}_{queue}[1]) == \text{rd}_{addr}(\text{instr}_{queue}[2]) \), then there is a data dependency between the current and previous instructions. In this case, the alu_computealu_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.

- **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 differentiate between the \text{sw}_{exec} and \text{sw}_{mem} implementations (as introduced in §II-B), and specifically identify \( (lw; sw)_{exec} \) 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]).
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 $use(t)$ and $mod(t)$. For example, $lb\_refill$ in Fig. 4(B) has $use(t) = \{mem\_addr, mem\_data\}$ and $mod(t) = \{lb\_addr, lb\_data, lb\_valid\}$.

**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 $t \in L$, 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
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: \( \sigma_{data} = [lb\_data \rightarrow 10, \cdots ] \) and \( \sigma'_{data} = [lb\_data \rightarrow 30, \cdots ] \). 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 \( \sigma_{data} \) into \( \sigma'_{data} \). 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 \( use(t') \cap \mod(t) \) 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 \( \sigma_{data} \) into \( \sigma'_{data} \), 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 (\( \text{counter}=2 \) to 3), the \( \text{lb\_refill} \) micro-update (Fig. 4B) correctly updates the LB signals. However, the \( \text{lb\_hold} \) micro-update (not shown) which maintains the same values 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 \( \text{mem} \) stage instruction, \( \text{inst\_mem} \), is a load. This is not the case for this transition since \( \text{inst\_mem} \) is an \( \text{addi} \) instruction. Hence, \( \text{lb\_refill} \) 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 \( \text{lb\_hold} \) and \( \text{lb\_refill} \) respectively, a distinguishing oracle generates a trace which includes the first transition (\( \text{counter}=0 \) to 1) of Example 2. We note that only the \( \text{lb\_hold} \) micro-update applies for this transition (since the LB entry remains invalid). The distinguishing oracle can also generate a trace with the second transition (\( \text{counter}=1 \) to 2), where only \( \text{lb\_refill} \) 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 \( \phi \), cover properties of the form \( \text{cover}(\phi) \) can be supplied to a hardware verification tool (e.g. JasperGold, SymbiYosys [19]). The tool, for a given cover property \( \phi \), aims to generate an execution in which \( \phi \) 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: \( S_{\text{trig}} = \{i_{\text{fet}}, i_{\text{exe}}, i_{\text{sh}}\} \). Consider a bypassing path micro-update that is triggered when there is a data dependency between \( i_{\text{exe}} \) and \( i_{\text{fet}} \), and \( i_{\text{exe}} \) does not write to zero. The guard corresponding to this micro-update is:

\[
\phi_{\text{bypass}} \equiv (\text{rd}(i_{\text{exe}}) = r\text{sl}(i_{\text{fet}})) \& (\text{rd}(i_{\text{exe}}) \neq 5'\text{b}00000)
\]

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 \( \text{cover}(\phi_{\text{bypass}}) \) 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 \((\sigma_{\text{trig}}^i, b^i)\) from the simulation traces and corresponding \( M \)-sets. Here, \( \sigma_{\text{trig}}^i \) 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 \( \sigma_{\text{trig}}^i \): \( \bigwedge_{i=1}^t (b^i \lor \neg G_t(\sigma_{\text{trig}}^i)) \). If not, the generated model would be incorrect at step \( i \).

Additionally, for each signal \( s \in S_{\text{data}} \), we require that exactly one micro-update modify \( s \) at each step. We define the set of micro-updates that modify \( s \) as \( m(s) \). The following formula enforces the “exactly one” constraint for signal \( s \): \( \bigwedge_{i=1}^t (\neg G_t(\sigma_{\text{trig}}^i) \land \neg G_{t'}(\sigma_{\text{trig}}^i)) \land (\bigvee_{i \in m(s)} G_t(\sigma_{\text{trig}}^i)) \).

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 \( S_{\text{data}} \)-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 \( S \setminus S_1 \) (i.e. signals in \( S \) but not \( S_1 \)). In particular, this requires generating guard predicates only for micro-updates relevant to \( S \setminus S_1 \). 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...
Then we reuse it when generating a model for the load_store_unit (C), of which 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.


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/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 $\leq$ 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 sw_{exe} implementation. In B, sw_{exe} flushes the LB after the lw refills it. However, in C, sw_{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 handwritten 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
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 $s = to =$. The accompanying waveforms show how (C) remains tainted under $\text{S}_{\text{wex}}$ while (B) does not.

**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 $\text{S}_{\text{wex}}$) while model run time is over our lifted ALULS model.

<table>
<thead>
<tr>
<th>Symbolic instruction sequence (testcase)</th>
<th>Source signals (src)</th>
<th>Outcome</th>
<th>Safe/Unsafe</th>
<th>Design runtime</th>
<th>Model runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop lw</td>
<td>mem</td>
<td>Safe</td>
<td>2m43s</td>
<td>8s</td>
<td></td>
</tr>
<tr>
<td>nop lw</td>
<td>mem,regs</td>
<td>Unsafe</td>
<td>3m50s</td>
<td>11s</td>
<td></td>
</tr>
<tr>
<td>nop alui lw</td>
<td>mem</td>
<td>Safe</td>
<td>2m37s</td>
<td>12s</td>
<td></td>
</tr>
<tr>
<td>nop alui lw</td>
<td>mem,regs</td>
<td>Unsafe</td>
<td>2m30s</td>
<td>9s</td>
<td></td>
</tr>
<tr>
<td>nop sw lw</td>
<td>mem</td>
<td>Safe</td>
<td>4m15s</td>
<td>8m51s</td>
<td></td>
</tr>
<tr>
<td>nop sw lw</td>
<td>mem,regs</td>
<td>Safe</td>
<td>4m59s</td>
<td>38s</td>
<td></td>
</tr>
<tr>
<td>nop lw + alui                  * sw</td>
<td>mem</td>
<td>Safe</td>
<td>5m11s</td>
<td>1m31s</td>
<td></td>
</tr>
<tr>
<td>nop lw + alui                  * sw</td>
<td>mem,regs</td>
<td>Safe</td>
<td>5m11s</td>
<td>1m31s</td>
<td></td>
</tr>
</tbody>
</table>

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 $S_{\text{data}}$-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 ($\text{wbuffer}$) is a coalescing buffer for the write-through data-cache, and sits between the core, data-cache and data-memory. The store unit ($\text{store\_unit}$) 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 ($\text{load\_store\_unit}$) 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 ($S_{\text{data}}$) include all entries in these buffers. For example, each wbuffer entry has three 1-bit signals representing its state: $s = (\text{valid}, \text{dirty}, \text{txnbk})$. The micro-update library 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 $\sim 4m$.

**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 ($\dagger$ in Tab. III) of 1 hr.

2) Can hierarchical synthesis help improve scalability?: Since monolithic synthesis does not scale, we attempt hierar-
TABLE III: Summary of the micro-update generation from the components of cva6. The row marked with (†) denotes monolithic synthesis, while for (⋆) we use hierarchical synthesis (§IV-G), by reusing the store_unit (sub-module) generated previously.

<table>
<thead>
<tr>
<th>Module</th>
<th>Extracted signals-of-interest ($S_{data}$)</th>
<th>$M$-set generation</th>
<th>Guard synthesis</th>
<th>Equivalence proof (d = 15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>states of buffer entries</td>
<td>1s</td>
<td>4s</td>
<td>6s</td>
</tr>
<tr>
<td>store_buffer</td>
<td>states of buffer entries (valid,dirty,txnblk)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>store_unit</td>
<td>store queue, store req. states</td>
<td>1s</td>
<td>2m47s</td>
<td>1m10s</td>
</tr>
<tr>
<td>load_store_unit</td>
<td>store queue, store req. states, load req. state (e.g. valid, spec, commit, memresp)</td>
<td>2s</td>
<td>TO (†)</td>
<td>1m49s</td>
</tr>
</tbody>
</table>

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 (⋆ 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 invulnerability 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 8x 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 sw·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 microarchitectural-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


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.


