2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)

RTL2MμPATH: Multi-μPATH Synthesis with Applications to Hardware Security Verification

Yao Hsiao
Stanford University
yaohsiao@stanford.edu

Nikos Nikoleris
Arm
Nikos.Nikoleris@arm.com

Artem Khyzha
Arm
Artem.Khyzha@arm.com

Dominic P. Mulligan*
Amazon Web Services
dominic.p.mulligan@gmail.com

Gustavo Petri*
Amazon Web Services
gfpetri@amazon.co.uk

Christopher W. Fletcher
University of California, Berkeley
cwfletcher@berkeley.edu

Caroline Trippel
Stanford University
trippel@stanford.edu

***Abstract*—The *Check* tools automate formal memory consistency model and security verification of processors by analyzing *abstract models* of microarchitectures, called μSPEC models. Despite the efficacy of this approach, a verification gap between μSPEC models, which must be manually written, and RTL limits the *Check* tools' broad adoption. Our prior work, called RTL2μSPEC, narrows this gap by automatically synthesizing formally verified μSPEC models from SystemVerilog implementations of simple processors. But, RTL2μSPEC assumes input designs where an instruction (e.g., a load) cannot exhibit more than one *microarchitectural execution path* (μPATH, e.g., a cache hit or miss path)—its *single-execution-path assumption*.**

In this paper, we first propose an automated approach and tool, called RTL2MμPATH, that resolves RTL2μSPEC's single-execution-path assumption. Given a SystemVerilog processor design, instruction encodings, and modest design metadata, RTL2MμPATH finds a complete set of formally verified μPATHs for each instruction. Next, we make an important observation: an instruction that can exhibit more than one μPATH strongly indicates the presence of a microarchitectural side channel in the input design. Based on this observation, we then propose an automated approach and tool, called SYNTHLC, that extends RTL2MμPATH with a symbolic information flow analysis to support synthesizing a variety of formally verified *leakage contracts* from SystemVerilog processor designs. Leakage contracts are foundational to state-of-the-art defenses against hardware side-channel attacks. SYNTHLC is the *first automated methodology* for formally verifying hardware adherence to them.

I. INTRODUCTION

A common strategy to improve the efficacy of a formal verification procedure is to analyze an *abstract model* of the target system, which omits irrelevant design details [27], [52], [64]. This approach is exemplified by the *Check* tools [63], which automate formal memory consistency model [55]–[59], [82] and security [80] verification of processors.

At their core, the Check tools conduct *microarchitectural happens-before* (μHB) analysis [55], which models hardware-specific program executions as directed μHB graphs (Fig. 1). A node in a μHB graph represents a microarchitectural event, such as a dynamic program instruction (column label) updating a particular set of hardware state elements (row label) [46]. Directed edges denote happens-before relationships [51].

*Work done while at Arm.

To facilitate μHB analysis, the Check tools analyze a microarchitecture in the guise of an *axiomatic* μSPEC *model*—an abstract model of a microarchitecture, which omits irrelevant RTL details [56]. In particular, a μSPEC model is a set of first-order logic axioms (rules) that describe how to construct μHB graphs to model hardware-specific program executions. Axioms encode (i) all *microarchitectural execution paths* (μPATHs—our term) for each implemented instruction, to instantiate column-wise nodes/edges (as in Fig. 1) and (ii) all possible *microarchitectural dependencies* between pairs of executed instructions, to instantiate edges between columns [46].

Despite finding bugs in real hardware [55], [60], [80]–[82], the *Check* tools have not achieved broad adoption due to a verification gap between μSPEC models, which must be manually written, and RTL.

Our prior work narrows this gap via an automated approach and tool, called RTL2μSPEC, for synthesizing formally verified μSPEC models from simple SystemVerilog processor designs [46]. However, RTL2μSPEC possesses a critical limitation: it cannot discover more than one μPATH per implemented instruction. Hence, designs where instructions may exhibit more than one μPATH are not supported—the *single-execution-path assumption* [46]. This restriction is incompatible with pervasive hardware features like: caches, which create hit and miss paths for memory instructions; variable-time functional units (e.g., serial dividers), which create a few to many path possibilities for certain instructions; speculation, which creates commit and squash paths for virtually all instructions; and multiple copies of the same functional unit, where certain instructions may take a path that uses any one of the units. We further show that it precludes designs with microarchitectural side channels, making RTL2μSPEC-synthesized μSPEC models useless for *Check*-based hardware security verification [80].

A. This Paper

This paper makes three key contributions.

Foundation: Synthesizing Microarchitectural Execution Paths: **Our first contribution** is an automated approach and tool, called RTL2MμPATH, that resolves RTL2μSPEC's single-execution-path assumption. Given a SystemVerilog processor design, instruction encodings, and modest (mostly standard)

2379-315/24/$31.00 ©2024 IEEE
DOI 10.1109/MICRO61859.2024.00045

design metadata, RTL2MμPATH uses static netlist analysis, linear temporal logic (LTL) [62], [72] property generation (from property templates), and model checking [15], [28] to find a complete set of formally verified μPATHs (μHB graph columns) for each instruction. Automated μPATH synthesis with RTL2MμPATH is enabled by two key aspects of its design.

First, we extend the μHB graph formalism from prior work with *cycle-accurate* timing information. In this paper, a μHB node represents an instruction updating a particular set of hardware state elements *in a specific cycle*; edges encode *one-cycle* happens-before relationships. Cycle-accurate μPATHs enable RTL2MμPATH to distinguish executions where the same set of state elements is updated a different number of times.

Second, RTL2MμPATH recognizes a μHB node during an instruction's execution on a microarchitecture when the instruction *visits* (i.e., *occupies*) a particular *performing location* (PL) in a given cycle. Similar to a pipeline stage, but more granular, a PL represents a *step* of an instruction's execution during which it has exclusive write access to a particular subset of design states. That is, PLs encapsulate instructions' state updates per cycle. We show that PLs are precisely captured by certain finite state machines within a processor's control-path (§III-C). By conceptualizing μHB nodes as instructions' visits to PLs, RTL2MμPATH supports speculative, superscalar, and out-of-order pipelines, plus caches.

Observation: Security Implications of μPATH Variability: In designing RTL2MμPATH, we make an important observation about how programs leak their private data in hardware side-channel attacks. Briefly, these attacks are often defined using a telecommunications analogy [48]: a *transmit instruction* (or *transmitter*, in the victim program) modulates a *channel* (hardware resource) in an operand-dependent manner, and a *receiver* (attacker) observes the channel modulation to infer the operand value [48]. We observe that:

> **Observation I:** When a transmitter modulates a channel in an operand-dependent manner, it creates operand-dependent μPATH variability (>1 μPATH) for one or more *transponder instructions* (or *transponders*—our term). A receiver observes a transmitter's distinct channel modulations as distinct μPATHs for transponder(s).

As an example, consider Fig. 1, which shows two μPATHs for a 32-bit multiply (`MUL`) instruction executing on CVA6-MUL, a variant of the RISC-V CVA6 CPU [100] that implements the *zero-skip multiply* optimization [10], [12], [40]. On this design, a `MUL` will spend one cycle in the multiplication unit if it has at least one zero operand; else, it will spend four cycles [12]. Such a `MUL` is a transmitter [88]: it occupies a hardware resource for an operand-dependent number of cycles.

Clearly, a `MUL` creates μPATH variability for itself: it may visit `mulU`, the *multiplication unit* PL, for one (μPATH 0) or four (μPATH 1) consecutive cycles. More subtly, a `MUL` may create multiple μPATHs for subsequent (in program order), concurrently in-flight instructions, which may stall behind the `MUL` for one to five cycles before committing (after completing). So, `MUL` transmitters implicate themselves *and* younger, concurrent instructions as transponders. A receiver observes a `MUL`'s distinct channel modulations as distinct μPATHs (e.g., with distinct latencies) for any of its transponders.

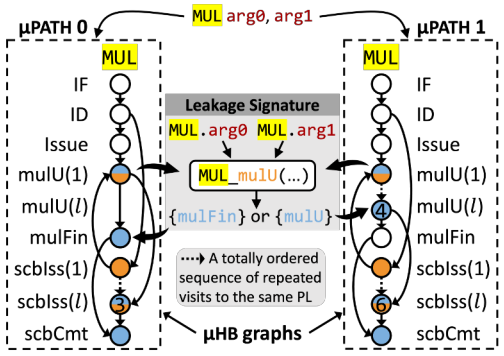


Fig. 1: Two μPATHs for `MUL` on CVA6-MUL (§I-A) and a leakage signature, which defines transponder `MUL`'s μPATH variability as a function of its own operands following its visit to the `mulU` PL. Row(1/l): 1st/l-th visit to Row. Node label: value of l for the μPATH.

Observation I captures all instances of operand-dependent hardware resource usage (§V-C2), including prior notions such as *implicit branches* [99], e.g., conditional cache accesses for loads (μPATH variability) that arise due to store and load address-dependent store-to-load forwarding. Thus, **our second contribution** is augmenting the aforementioned telecommunications analogy with the notion of a transponder.

Application: Synthesizing Leakage Contracts: Observation I also inspires **our third contribution**: an open-source [1] automated approach and tool, called SynthLC, which extends RTL2MμPATH with a symbolic information flow analysis to support synthesizing *a variety of* formally verified microarchitectural *leakage contracts* from SystemVerilog processor designs. Leakage contracts are foundational to hardware side-channel defenses, implemented in software [25], [35], [67], [68], [87], [101] or hardware [16], [20], [26], [53], [76], [97]–[99]. SynthLC is the *first automated methodology* for formally verifying hardware adherence to them.

Specifically, SynthLC synthesizes a complete set of formally verified microarchitectural *leakage signatures* from processor RTL. Leakage signatures are a novel formalism that we introduce to capture all relevant features of state-of-the-art leakage contracts, which are not already captured by μPATHs. They are effectively function signatures, which characterize how one or more transmitters create operand-dependent μPATH variability for a transponder with respect to (i.e., following) some step (PL) of the transponder's execution. Fig. 1 depicts a leakage signature, which defines how a `MUL` transmitter creates μPATH variability for itself (it is also a transponder) as a function of its operands, following its `mulU` execution step.

From μPATHs and leakage signatures, a variety of formally verified leakage contracts can be easily derived. The canonical *constant-time (CT) contract*, which enumerates a microarchitecture's transmitters and their unsafe operands [3], [11], [47], is captured by the inputs to the leakage signature in Fig. 1.

Overall, SYNTHLC supports synthesizing *six different leakage contracts* from SystemVerilog processor designs (Table I): CT contracts plus five bespoke leakage contracts from the literature. Collectively, these contracts support two software defenses—the *CT programming defense* [25], [35] and the *speculative constant-time (SCT) programming defense* [67], [68], [87], [101]—and eight hardware defenses [16], [20], [26], [53], [76], [97]–[99] against hardware side-channel attacks.

Case Study: Deploying SYNTHLC *on a Processor Core and Cache:* We deploy SYNTHLC on the RISC-V CVA6 core [100], surfacing 94 unique leakage signatures, 72 transponders, and 26 transmitters. Compared to prior work that analyzes the same design [31], [32], SYNTHLC finds a novel channel that leaks store and load address operands.

We separately deploy SYNTHLC on the CVA6 L1 data cache and cache controller, making it the first leakage contract verification procedure to analyze a realistic processor cache. Beyond uncovering various hardware side channels, our cache experiment showcases the performance and scalability benefits of converting SYNTHLC into a modular procedure.

II. BACKGROUND

This section gives an informal overview of hardware side-channel attacks (§II-A) and defenses (§II-B), including descriptions of six leakage contracts whose implementation in hardware can be formally verified with SYNTHLC.

A. Hardware Side-Channel Attacks

In this paper, we study hardware side-channel attacks where a *transmit instruction* (or *transmitter*, in the victim program) modulates a *channel* (hardware resource) in an operand-dependent manner, and a *receiver* (attacker) observes the channel modulation to infer the operand value [48]. We highlight standard assumptions for channels and receivers below.

Characterizing Channels: Many hardware resources have been implicated as channels, e.g., caches [70], [95], [96], branch predictors [8], [37], functional units [10], [40], memory ports [9], and more [39], [65], [71], [88]–[90], [93], [94]. Owing to the circumstantial nature of leakage due to particular channels [4], [30], prior work informally classifies them as *stateless* or *stateful* [4]. Stateless channel modulations are observable only in very narrow, specific time windows, usually requiring the receiver to be running at the same time as the victim, e.g., a victim may create memory port contention for a receiver during a victim memory access. Stateful channels may be observed long after they are modulated, e.g., a victim cache access may create a cache miss for a receiver far in the future. We refer to stateless/stateful channels as *dynamic/static* to denote formal definitions that we introduce (§IV-C).

Receiver Assumptions: Hardware side-channel attacks and defenses are studied under specific *receiver assumptions*, consisting of an *observer model* and *attacker strategy*.

An observer model defines how channel modulations manifest as observations for a specific receiver. Informally, they may be perceived via their (measurable) effects on certain non-deterministic aspects of program execution, e.g., execution time [18], [41], [70], resource contention [8]–[10], [39], [65], [71], [93], and so on [14], [44], [50], [61], [74], [84].

An attacker strategy specifies whether the receiver may *passively* or *actively* monitor victim channel modulations [19], [30], [79], [88]. In a passive attack, the receiver monitors victim channel modulations without explicitly interfering with the victim's execution, e.g., by measuring victim execution time. In an active attack, the receiver explicitly interacts with the channel modulated by the victim, e.g., by priming and probing shared cache state. The notion of a transponder enables defining both strategies formally (§IV-C).

B. Hardware Side-Channel Defenses & Leakage Contracts

The goal of a hardware side-channel defense is to ensure that a specific victim program running on a particular microarchitecture will not leak its private data to some receiver via hardware side channels. Towards this goal, state-of-the-art defenses assume the availability of microarchitectural *leakage contracts*, which characterize implementations' transmitters.

The Canonical Leakage Contract: The most widely-adopted leakage contract, the *constant-time (CT) contract*, enumerates a microarchitecture's transmitters and their unsafe ("leaky") operands. Nascent ISA leakage contracts fall into this category [3], [11], [47]. Given a CT contract, a hardware side-channel defense can ensure that secrets *never* reach the unsafe operands of transmitters when programs execute. This strategy is embodied in the *CT programming defense* [25], [35], the gold standard software defense for protecting the *architectural* executions of cryptopgraphic code [29], [69], [83], [102] from hardware side-channel attacks. Some software [67], [68], [87], [101] and hardware [16], [76] defenses against *transient execution attacks*—which exploit hardware faults and mis-predictions to *steer* secrets towards the unsafe operands of *transient* (bound-to-squash) transmitters [23], [49]—extend this strategy to target programs' *speculative* executions as well. We refer to software variants of such defenses as *speculative constant-time (SCT) programming defenses* [87].

Five Bespoke Leakage Contracts: Today, the most performant defenses against transient execution attacks are implemented in hardware [20], [26], [53], [97]–[99]. Instead of CT contracts [20], [98], [99] (or in addition to them [26], [53], [97]), these defenses adopt bespoke leakage contracts that are much finer grained. We consider five such fine-grained leakage contracts in this paper. Table I in §IV shows how features of these five contracts, described below for each of the six hardware defenses they enable, and the CT contract map onto the various components of our proposed leakage signatures.

MI6 [20] defends enclaves from hardware side-channel attacks via two mechanisms. First, it requires identifying *dynamic (stateless—§II-A) channels* that arise due to transmitter operand-dependent hardware *resource contention*. Data-independent scheduling logic is implemented for impacted resources. Second, it requires identifying *static (stateful) channels* in order to implement a purge instruction, which flushes relevant microarchitectural states, and partitioning schemes.

OISA [97] attaches secrecy labels to architectural state and detects if a transmitter's unsafe operand, as defined in a CT contract, is ever passed secret data at runtime. It enables transmitters that exhibit execution variability due to *input-dependent arithmetic units* (like `MUL` in §I-A) to safely process secrets as follows. First, the designer identifies arithmetic units that may be occupied by a transmitter for an operand-dependent number of cycles. Second, control logic is added to enforce operand-independent execution for the unit whenever a transmitter arrives with a secret-labeled unsafe operand.

STT [99] requires identifying all channels and classifying them as *explicit channels* and *implicit channels*. Explicit and implicit channels, respectively, arise when transmitters' operand-dependent hardware resource usage affects their own execution behavior and the execution behavior of other instructions. Transmitters that modulate explicit channels are not permitted to execute with (potentially) secret operands. To block implicit channels, designers must also identify *explicit branches* or *implicit branches*. Explicit branches are architectural control-flow instructions, like conditional branches. Implicit branches are instructions whose execution behavior depends on the operands of other transmitters. Finally, implicit channels are categorized as *prediction-based* or *resolution-based*. To block prediction-based and resolution-based channels, respectively, the designer must identify predictor structures that are updated by explicit or implicit branches and the points at which these branches resolve their predictions.

SDO [98] extends STT by optimizing its explicit channel defense. First, the designer identifies transmitters that modulate explicit channels. To enable safe speculative execution of these transmitters with (potentially) secret operands, the designer creates a number of data-independent execution path variants for each. These so-called *data-oblivious variants* are derived from the set of realizable microarchitectural execution paths for each transmitter on the baseline design. Then, the designer adds control logic to select which path a transmitter should take based on public predictor state.

Dolma [53] requires a CT contract to delay the execution of transmitters until they become non-speculative. To improve performance, Dolma has several additional requirements. First, the designer must identify *variable-time micro-ops* and any *contention-based dynamic channels* they create. Second, *inducive micro-ops* that exhibit execution variation as a function of *resolvent micro-ops'* (transmitters') operands must be flagged, along with the *prediction resolution point* during the inducive micro-op's execution at which this variation arises. Lastly, *persistent state modifying micro-ops* (transmitters that modulate static channels) must be identified and their persistent state updates delayed until they become non-speculative.

SPT [26] and STT have the same fine-grained leakage contract and differ only in their policy for when it safe to declassify (mark public) data. STT declassifies data once it is not a function of speculatively accessed data. SPT declassifies data once it is inevitable that it will be, or has been, transmitted architecturally; so, SPT additionally requires a CT contract.

III. RTL2MμPATH Approach: Formalizing Microarchitectural Execution Paths

Our first contribution is an automated approach and tool, called RTL2MμPATH,[1] for uncovering a complete set of formally verified *microarchitectural execution paths* (μPATHs) for each instruction implemented on a SystemVerilog processor design. In this section, we explain the two key technical advances that enable such an analysis: an extension to the μHB graph formalism from prior work (§III-B) and a novel mapping of μHB nodes onto RTL signals (§III-C). We establish the need for both with a motivating example (§III-A). A discussion of RTL2MμPATH's implementation details is deferred until §V.

A. Motivating Example: Operand Packing

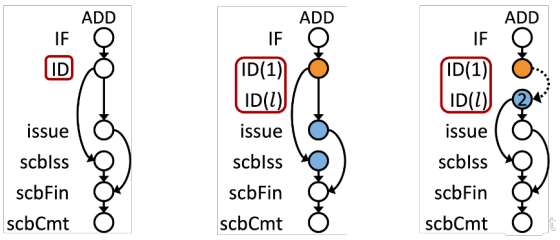
Consider an extension to the RISC-V CVA6 CPU [100] that we verify in §VI, called CVA6-OP. The baseline CVA6 design may fetch up to two compressed instructions or one uncompressed instruction per cycle, but only one instruction can be sent to decode per cycle. CVA6-OP is identical to CVA6, except that the ALU has been modified to support *operand packing* [21], and up to *two* instructions can be sent to decode per cycle. In particular, if a pair of (concurrently) decoded instructions perform identical ALU operations *and* feature narrow width operands (32 bits or less, i.e., the upper halves of their 64-bit operands contain all zero bits), they will be packed into a single operation and executed together.

Suppose we want to formally evaluate the vulnerability of CVA6-OP to hardware side-channel attacks with the *Check* tools (i.e., with *CheckMate* [80]). As discussed in §I, we first need to acquire an abstract axiomatic model of the microarchitecture, called a μSPEC model.

The enabler for *Check*-based hardware security verification is the fact that μSPEC models define how instructions may exhibit variable hardware resource usage when they run on a specific hardware implementation (e.g., a load may experience a cache hit or miss [80]). That is, they specify the various *microarchitectural execution paths* (μPATHs—our term) that may be exhibited by each instruction type (e.g., a `LD`, `MUL`, `ADD`, etc.) on the design. A μSPEC model encodes distinct μPATHs as distinct *microarchitectural happens-before* (μHB) graphs (Fig. 2). A node in a μHB graph represents a microarchitectural event, such as a dynamic program instruction (column label) updating a particular set of hardware state elements (row label) during its execution [46]. Directed edges denote happens-before relationships [51].

A μSPEC model for CVA6-OP *should* capture the fact that `ADD`s may exhibit two distinct μPATHs, depending on whether they are packed or not. However, prior work [55]–[59], [80], [82] would represent both of these scenarios with the same μPATH—the one in Fig. 2a. This is because the packed versus unpacked μPATHs differ according to how long an `ADD` spends in the decode stage (one or two cycles), i.e., how many times the `ID` node appears. The μHB formalism has not been used to express repeated instances of the same μHB node.

[1] "RTL2MμPATH" indicates that it finds *multiple* (M) μPATHs per instruction.



(a) *(Non-)packed* μPATH (b) *Packed* μPATH (c) *Non-packed* μPATH

Fig. 2: ADD μPATHs on CVA6-OP (§III), using standard μHB graphs from prior work (a) and our new cycle-accurate μHB graphs (b, c). Row(1/l): 1st/l-th visit to Row. Node label: value of l for μPATH.

Even if we resolve this minor modeling issue, the only approach and tool for synthesizing formally verified μSPEC models from processor RTL, called RTL2μSPEC, cannot uncover more than one μPATH per implemented instruction (§I). Beyond missing repeated instances of the same node, RTL2μSPEC's mapping of μHB nodes to RTL signals precludes identifying nodes that appear in some μPATHs but not others. We address these limitations in §III-B and §III-C, respectively.

B. Extending μHB Graphs with Cycle-Accurate Timing

We extend the μHB graph formalism from prior work [46], [55]–[59], [80], [82] with *cycle-accurate* timing information. In particular, *for all μPATHs in this paper* (except in Fig. 2a), a μHB node represents an instruction updating a particular set of state elements *in a specific cycle*, while an edge represents a *one-cycle* happens-before relationship. This extension enables μHB graphs to express that an instruction may update the same set of state elements in multiple (consecutive or non-consecutive) cycles, which is needed to represent real designs, e.g., ones where an execution unit has variable latency (like CVA6-MUL in Fig. 1) or throughput (like CVA6-OP in Fig. 2).

Using our cycle-accurate μHB graph notation, the μPATHs in Figs. 2b and 2c depict distinct (concrete) executions of an ADD on CVA6-OP that distinguish when the ADD is packed or not, respectively. We use row label Row(n) to denote the n-th update to the set of state elements referred to by Row.

We use a special notation to summarize l *consecutive* updates to the same set of state elements. Specifically, a pair of row labels Row(1) and Row(l) denote the first and last (l-th) consecutive updates to the state elements referred to by Row. Node labels specify the value of l (which may be execution dependent) for a particular concrete μPATH. For example, in Fig. 2c, the node at ID(l) denotes an ADD's second consecutive update to the state elements referred to by ID. Dashed edges that relate a Row(1) node to its corresponding Row(l) node represent a totally ordered (by μHB edges) sequence of $l-2$ nodes, ordered after Row(1) and before Row(l), with no other outgoing edges. When $l = 2$ (as in Fig. 2c for ID(l)), the dashed edge represents a normal (solid) μHB edge.

In every μPATH in this paper—each of which was synthesized from CVA6 [100] using RTL2MμPATH (or, in the case of Figs. 1 and 2, adapted from synthesized μPATHs)—IF represents the first set of state elements that an instruction updates upon being fetched, and scbCmt represents those updated upon commit. Thus, one can deduce that a set of state elements S (row label) is updated in cycle t of an instruction's execution, along some μPATH, if t is longest sequence of μHB edges from the node at IF to the node at S, accounting for implicit nodes and edges due to consecutive state updates. An instruction's overall latency is given by the longest sequence of μHB edges from the node at IF to the node at scbCmt, e.g., four (Fig. 2b) or five (Fig. 2c) cycles for a packed or non-packed ADD on CVA6-OP, respectively.

In summary, our cycle-accurate μPATH abstraction provides a precise accounting of an instruction's hardware resource usage (i.e., state updates) in time and space.

C. Recognizing μHB Nodes with Performing Locations

RTL2MμPATH uses static netlist analysis, linear temporal logic (LTL) [62], [72] property generation (from property templates), and model checking [15], [28] to uncover all μPATHs for each implemented instruction on a given SystemVerilog processor design (§V). Importantly, it requires that a model checker be able to recognize the various components of a μPATH—μHB nodes and edges—when exploring instructions' executions. As one-cycle happens-before relationships, edges are readily expressible in LTL syntax. However, nodes must be explicitly conceptualized in terms of RTL signals.

Recall that a μHB node represents an instruction updating a particular set of design states in a given cycle (§III-B). Thus, a model checker recognizing a μHB node equates to it detecting what state elements are updated in some execution cycle and attributing these updates to a particular in-flight instruction. At first glance, this task appears highly challenging, given that modern processors execute numerous instructions concurrently and out-of-order. However, we observe that the same mechanisms that enable a processor to orchestrate instructions' write access to design states can also be leveraged by a model checker to recognize μHB nodes.

In particular, we find that a subset of *finite state machines* (FSMs) within a processor's control path, which we call *micro-op FSMs* (μFSMs), govern instructions' state updates throughout their execution, from the time they are fetched until possibly after they commit (e.g., when stores update cache state). A μFSM is a tuple $\langle$iir, vars$\rangle$, where iir is an *instruction identifying register* (IIR), which holds a unique *instruction identifier* (IID) for an in-flight instruction, and vars is a collection of state elements, which encode the μFSM's *state variables*. Example IIDs include program counters (PCs) [46], [73], reorder buffer (ROB) or scoreboard (SCB) identifiers, and memory transaction identifiers. An in-flight instruction acquires a μFSM by placing one of its IIDs in the μFSM's iir, progresses through various μFSM states (i.e., concrete valuations of its vars), and then releases the μFSM (i.e., by setting its vars to an *idle* state). The valuation of a μFSM's vars in a given cycle grants the instruction whose IID is contained in its iir exclusive write access to a particular subset of design states.

```
// store_unit.sv
  enum logic {...} state_d, state_q;                  // vars0
  logic [TRANS_ID_BITS-1:0] trans_id_n, trans_id_q;// iir0
+ logic [riscv::VLEN-1:0] st_pc_n, st_pc_q;          // pcr0
// load_store_unit.sv
  assign lsu_req_i = {
  lsu_valid_i, fu_data_i.trans_id, ...    // (vars1, iir1)
+ , pc_i };                               // pcr1
```

Fig. 3: Close proximity of μFSMs' `iir` and `vars` components (§III-C). PCRs are added (+) near IIRs that do not hold PCs (§V).

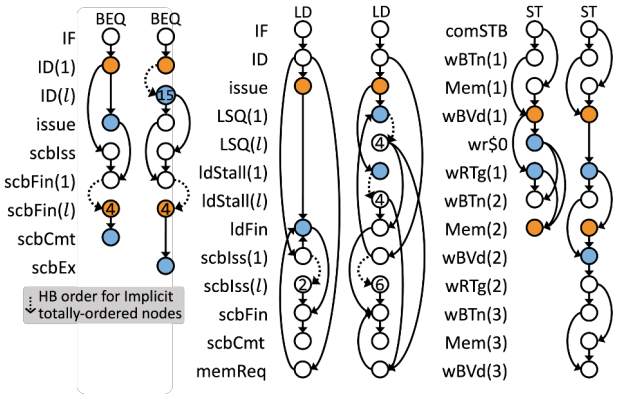
Leveraging μFSMs, we conceptualize μHB nodes in terms of RTL signals as follows. First, we define the notion of a *performing location* (PL)—similar to a pipeline stage, but more granular—as a $\langle \mu\texttt{fsm}, \texttt{state} \rangle$ tuple, where μ`fsm` is a μFSM (i.e., an $\langle \texttt{iir}, \texttt{vars} \rangle$ tuple), and `state` denotes a valid, non-idle valuation of μ`fsm`'s `vars`. Hence, the set of all PLs for a processor implementation denotes the set of all valid, non-idle states across all of the design's μFSMs. Next, we say that "an instruction i *visits* (i.e., *occupies*) PL $\langle \mu\texttt{fsm}, \texttt{state} \rangle$ some cycle," if at the start of that cycle, μ`fsm`'s `iir` contains an IID of i and μ`fsm`'s `vars` is set to `state`. Like a pipeline stage, a PL can be occupied by a single instruction at a time, whose state updates (which take effect at the start of the next cycle) it encapsulates. Unlike a pipeline stage, an instruction may visit multiple PLs in the same cycle. Finally, we direct a model checker to recognize a μHB node during an instruction's execution on a microarchitecture when it detects the instruction visiting a particular PL in a given cycle. Hence, in all μPATH figures in this paper, row labels (ignoring parentheticals) denote PLs.

We require the designer to identify all signals that comprise μFSMs (i.e., their IIR and state variable components) in the input design (§V-A). RTL2MμPATH uses these signals to uncover all PLs for the design and then all ways in which they can be assembled into valid μPATHs for instructions. Specifying μFSMs turns out to be straightforward, since they are already in use to support functional verification efforts [42], [43]. Fig. 3 illustrates the close proximity of μFSMs' IIR and state variable signal definitions in CVA6 [100].

IV. SynthLC Approach: Applying Multi-μpath Synthesis to Hardware Security Verification

In designing RTL2MμPATH, we make an important observation: an instruction that exhibits μPATH variability (>1 μPATH) on some processor implementation strongly indicates the presence of a microarchitectural side channel in the design. Based on this observation, we develop SynthLC, the first automated approach and tool for formally verifying hardware adherence to microarchitectural leakage contracts (§II-B).

This section presents the SynthLC approach; a presentation of its implementation as a tool is deferred until §V. We first formalize instances of instruction μPATH variability using the notion of a *decision* (§IV-B). Second, we show how an instruction's decisions can be attributed to the outputs of *path selector functions* in the microarchitecture (§IV-C). If a path selector function output depends on some instruction's



(a) BEQ μPATHs (b) LD μPATHs (c) ST μPATHs

Fig. 4: A sampling of μPATHs for the CVA6 core (for `BEQ`, `LD`) and data cache (for `ST`), synthesized by RTL2MμPATH (§VII). Row(1/l): 1st/l-th visit to Row. Node label: value of l for μPATH.

operands, it is a *leakage function*, and said instruction is a transmitter. Instructions whose decisions can be attributed to leakage functions are *transponders*. From leakage functions, we derive *leakage signatures*—a unifying formalism that captures all relevant features of six leakage contracts from prior work (summarized in §II-B and Table I), which are not already captured by μPATHs (§IV-D). SynthLC conducts leakage contract verification by synthesizing a comprehensive set of leakage signatures from processor RTL (§V).

We present the concepts above using a more complex illustrative example involving store-to-load stalling (§IV-A).

A. Illustrative Example: Store-to-Load Stalling

Consider two μPATHs for a load (`LD`) executing on the RISC-V CVA6 CPU [100], shown in Fig. 4b. Following its visit to the `issue` PL (orange node), a `LD` exhibits one of two *decisions*. It stalls (right μPATH), progressing to `LSQ` and `ldStall` (blue nodes), if the page offset of its address operand matches that of any older pending store (`ST`) in the speculative or committed store buffers (STBs)—its *path selector function*. Else, it completes (left), progressing to `ldFin` (blue node).

Hardware side-channel defenses (§II-B) must ensure that stores (*transmitters*) do not leak their (private) address operands to a receiver who observes the μPATHs of executing loads (*transponders*), e.g., by timing their execution latency—five or nine cycles for the left or right μPATH, respectively, in Fig. 4b. CT contracts would declare store address operands as unsafe for processing secrets. STT, SDO, and SPT would flag such a load as an implicit branch, while Dolma would categorize the load/store as inducive/resolvent micro-ops. OISA would specify that store address operands are unsafe or require hardware mechanisms avoid this channel for stores with secret operands. MI6 declares this sort of leakage as out-of-scope.

B. Formalizing Instances of μPATH Variability with Decisions

We propose the notion of a *decision* to characterize specific variations across the different μPATHs of a particular instruc-

tion on a microarchitecture (e.g., a LD on CVA6).

Suppose $\boldsymbol{\mu}\mathbf{PATH}^I_M$ is the set of all possible μPATHs that a dynamic instance of instruction I (e.g., I = LD) can exhibit when it runs on microarchitecture M. Informally, "a decision made by I on M" is a tuple $(src, \mathbf{dst})$—where src is a single *decision source PL* (or *decision source*) and $\mathbf{dst}$ is a set of *decision destination PLs* (or *decision destinations*)—that pinpoints a divergence between a pair of I's μPATHs on M. Orange/blue nodes in μHB graph figures throughout the paper denote some, *but not all* (for clarity of presentation), decision sources/destinations. Suppose $\mathbf{d}^I_M$ is the set of all decisions that I can make on M. Then, $(src, \mathbf{dst}) \in \mathbf{d}^I_M$ if and only if: For some $p, p' \in \boldsymbol{\mu}\mathbf{PATH}^I_M$, I visits src in p one cycle before it visits exactly the PLs in $\mathbf{dst}$, and I visits src in p' one cycle before it visits a *different* set of PLs than exactly those in $\mathbf{dst}$. By $\mathbf{src}^I_M$ we denote the set of all decision sources across all of I's decisions on M.

For example, given the μPATHs in Figs. 2b and 2c, and considering the orange and blue colored nodes exclusively (for brevity), ADD has decision sources and decisions:

$$\mathbf{src}^{\texttt{ADD}}_{\text{CVA6-OP}} = \Big\{\texttt{ID}\Big\}$$

$$\mathbf{d}^{\texttt{ADD}}_{\text{CVA6-OP}} = \Big\{(\texttt{ID}, \{\texttt{issue}, \texttt{scbIss}\}), (\texttt{ID}, \{\texttt{ID}\})\Big\}.$$

Similarly, given Fig. 4b and considering only colored nodes, LD has decision sources and decisions:

$$\mathbf{src}^{\texttt{LD}}_{\text{CVA6}} = \Big\{\texttt{issue}\Big\}$$

$$\mathbf{d}^{\texttt{LD}}_{\text{CVA6}} = \Big\{(\texttt{issue}, \{\texttt{ldFin}\}), (\texttt{issue}, \{\texttt{LSQ}, \texttt{ldStall}\})\Big\}.$$

Note that in practice, decisions are defined with respect to PLs, irrespective of how many times they have been visited. For example, in Fig. 1, scbIss is a decision source for MUL on CVA6-MUL; it may be followed by decision destinations $\{\texttt{scbIss}, \texttt{mulU}\}$, $\{\texttt{scbIss}\}$, or $\{\texttt{scbCmt}\}$.

C. Selecting a Decision with a Path Selector Function

Suppose i_I is a dynamic instance of instruction I that visits decision source $src \in \mathbf{src}^I_M$ during its execution on microarchitecture M. Which decision i_I exhibits with respect to src—i.e., which decision destination(s) i_I visits one cycle after visiting src—is determined by a *path selector function* in hardware. In particular, during the cycle in which i_I visits src, a path selector function is queried to determine where i_I will progress to next. We use I_src to denote a path selector function that is queried when an instruction of type I visits decision source src; it returns a set of decision destinations.

Fig. 5 shows example path selector functions for CVA6-OP and CVA6 [100]. Path selector functions may have *explicit inputs* that are provided in the function argument list and *implicit inputs* that are not. Explicit inputs are instructions whose operands appear in the function body, capturing how architectural state influences μPATH variability. Implicit inputs are any other microarchitectural structures whose contents are accessed in the function body, capturing how microarchitectural state influences μPATH variability.

```
// CVA6-OP Core ADD (§III-A): ADD (i.e., ADD^N) in ID is issued if it is
ready (the oldest in ID) or eligible for operand packing; else, it is stalled.
dst ADD_ID(ADD^N i0, ADD^D_O i1) :
  return ite(((visit(i1, ID) ∧ (∀arg ∈ {i0.arg0, i0.arg1, i1.arg0, i1.arg1
    }, msb(arg) < 32)) ∨ rdy(i0)), {scbIss, issue}, {ID})

// CVA6 Core LD (§IV-A): LD in issue finishes or is stalled depending
on whether its address page offset overlaps with that of a pending ST.
dst LD_issue(LD^N i0, ST^D_O i1) :
  return ite((i1.addr ∈ (comSTB ∪ specSTB) ∧
    offset(i0.addr) == offset(i1.addr)),{ldStall, LSQ}, {ldFin})

// CVA6 Cache ST (§VII-A2): A ST accesses one of two data banks on
a hit in the 4-way set-assoc. no-write-alloc. cache.
dst ST_wBVld(ST^N i0, LD^S i1) :
  hit = (cacheTag[set(i1.addr)][way] == tag(i1.addr) ∧ set(
    i0.addr) == set(i1.addr) ∧ tag(i0.addr) == tag(i1.addr))
  return ite(hit,{wRTag, wr$[way/2]}, {wRTag})

// CVA6 Core ST (new channel, §VII-A1): ST at comSTB is stalled from
draining to memory if its address offset does not match a younger LD.
dst ST_comSTB(SW^N i0, LD^D_Y i1) :
  return ite((visit(i1, issue) ∧ offset(i0.addr) == offset(i1.addr)),
    {memRq, comSTB}, {comSTB}})
```

Fig. 5: Leakage function examples. Implicit inputs and leakage signature components are highlighted. $T^{\texttt{N}}/T^{\texttt{D}}_{\texttt{0|Y}}/T^{\texttt{S}}$: intrinsic / older or younger dynamic / static transmitters. PO: program order. msb: most significant bit. ite(c, t, f): t if c is true; else, f.

Moreover, each explicit input to a leakage signature has a type that captures both its *instruction type* (opcode/function) and particular *runtime conditions* (encoded with N, D, or S superscripts and 0 or Y subscripts—detailed below) that must be satisfied for the leakage signature to be applicable. In particular, with respect to a dynamic instruction i_I of type (opcode/function) I that visits $src \in \mathbf{src}^I_M$ on microarchitecture M and queries path selector function I_src, each explicit input i_T (an instruction with opcode/function T) is typed as:

- *Intrinsic* ($T^{\texttt{N}}$ i_T), if $i_I = i_T$, i.e., i_T is the instruction currently visiting src.
- *Dynamic* ($T^{\texttt{D}}_{\texttt{0|Y}}$ i_T), if i_T is older (0) / younger (Y) than i_I (in program order), and i_T must be in-flight (visiting some PL) when i_I visits src for it to influence I_src's output. Notably, when i_T is younger than i_I, M can be susceptible to speculative interference attacks [17].
- *Static* ($T^{\texttt{S}}$ i_T), otherwise.

Illustrative Example: LD_issue *Path Selector Function:* For our store-to-load stalling example (§IV-A), the path selector function LD_issue in Fig. 5 is queried to select a decision for a dynamic LD instruction $i_{\texttt{LD}}$ during any cycle in which it visits the issue PL. Its explicit inputs $\texttt{LD}^{\texttt{N}}$ $i0$ and $\texttt{ST}^{\texttt{D}}_{\texttt{0}}$ $i1$ indicate that $i_{\texttt{LD}}$'s decision at decision source issue is a function of its own operands and those of another older in-flight ST, respectively. Its outputs are one of two sets of destination PLs: $\{\texttt{ldStall}, \texttt{LSQ}\}$ (stall μPATH) or $\{\texttt{ldFin}\}$. The function body shows that the output of LD_issue specifically depends on the *address* operands of $i_{\texttt{LD}} = i0$ and $i1$.

A receiver that can determine which μPATH $i_{\texttt{LD}}$ exhibits relative to issue, learns LD_issues's return value. Since the return value depends on the address operands of explicit inputs

Defense	Leakage Contract Components	μ	Leakage Sig.					
			P	src	T^{N}	T^{D}	T^{S}	a
CT, SCT (§II-B) SpecShield [16] ConTExt [76]	Constant-time contract (§II-B)	-	-	-	✓	✓	✓	✓
MI6 [20]	Contention-based dynamic channels	-	✓	✓	✓	✓	-	-
	Static channels	-	✓	✓	-	-	✓	-
OISA [97]	Input-dependent arithmetic units	-	-	✓	✓	-	-	✓
STT [99] SDO [98] SPT [26]	Explicit channels	-	✓	✓	✓	-	-	✓
	Implicit channels	-	✓	✓	-	✓	✓	✓
	Implicit branches	-	✓	-	-	✓	✓	✓
	Prediction-based channels	-	✓	✓	-	-	✓	✓
	Resolution-based channels	-	✓	✓	-	✓	-	✓
SDO [98]	Data-oblivious variants	✓	-	-	✓	-	-	✓
Dolma [53]	Variable-time micro-ops	-	-	-	✓	-	-	✓
	Contention-based dynamic channels	-	✓	✓	✓	✓	-	✓
	Inducive micro-ops		✓	-	-	✓	-	✓
	Resolvent micro-ops	-	-	-	-	✓	-	✓
	Prediction resolution points	-	✓	✓	-	✓	-	✓
	Persistent state modifying micro-ops	-	-	-	-	-	✓	✓

TABLE I: Six leakage contracts (§II-B) mapped onto μPATHs (μ) and leakage signatures. a: Arguments. ✓/-: Relevant/irrelevant to the leakage contract component.

$i_{\texttt{LD}} = i0$ and $i1$, such a receiver may infer their values.

Leakage Functions and Transponders: At least one operand of an explicit input to a path selector function may leak its value to a receiver that observes the function output. So, we call a path selector function with at least one explicit input a *leakage function*; explicit inputs are *transmitters*. A receiver observes a path selector function output as a μPATH decision of the instruction that queried it. We call an instruction whose perceived μPATH variability leaks transmitter operands a *transponder*, extending the telecommunications analogy for characterizing hardware side-channel attacks (§II-A).

To summarize, a leakage function `dst P_src`$(i_T, i_{T'}, ...)$ characterizes a microarchitectural side channel, mapping the operand space(s) of transmitters $T, T', ...$ that modulate it onto the observation space of a receiver that observes it. Assuming a receiver that observes μPATHs of executing instructions, the observation space consists of the set of distinct decisions that *transponder* P may exhibit relative to decision source src.

Formally Characterizing Channels: Notably, leakage functions enable formally defining static versus dynamic channels and passive versus active attacks (§II-A).

We call a channel (leakage function) static iff it is modulated by a static transmitter. We call a channel dynamic iff it is modulated by an intrinsic or dynamic transmitter. A channel may be static *and* dynamic, e.g., consider a leakage function whose output decides whether or not a transponder LD stalls on a cache access. The decision may depend on another dynamic $\texttt{LD}^{D}$ that contends for the same read port *or* another static $\texttt{LD}^{S}$ that evicted LD's cache line, causing a cache miss. SYNTHLC discovers this scenario when analyzing the CVA6 cache (§VI).

By definition, a transmitter is an instruction in a victim program (§II-A). In a passive attack, the transponder is a victim instruction, which the attacker passively monitors. In an active attack, the transponder is a receiver instruction.

D. Unifying Leakage Contracts with Leakage Signatures

We propose a unifying formalism, called a *leakage signature*, to capture all relevant features of six state-of-the-art leakage contracts, summarized in §II-B, which are not already captured by μPATHs. A leakage signature is a leakage function that is restricted to the yellow-highlighted components of Fig. 5: transponder and decision source (function name), typed transmitters (explicit inputs), unsafe transmitter arguments (in the function body), and decision destinations (return values).

Table I shows how various components of the six leakage contracts in §II-B can be derived from μPATHs (μ) and leakage signatures. These contracts do not consider notions of decision destinations nor do they explicitly distinguish older versus younger dynamic transmitters, so the table omits these details.

Consider the five leakage contract components shared by STT [99], SDO [98], and SPT [26]. Each can be derived from the checked (✓) leakage signature components in Table I as follows. *Explicit channels* denote sources of μPATH variability (src) for intrinsic transmitters (T^{N}), which are transponders (P) by definition, as a function of their arguments (a). *Implicit channels* are sources of μPATH variability (src) for transponders (P) as a function of dynamic (T^{D}) or static (T^{S}) transmitters' arguments (a). *Implicit branches* are transponders (P) that exhibit μPATH variability due to dynamic (T^{D}) or static (T^{S}) transmitters' arguments (a). *Prediction-based channels* and *resolution-based channels* manifest as sources of μPATH variability (src) for transponders (P) due to dynamic (T^{D}) and static (T^{S}) transmitters' arguments (a), respectively.

V. SYNTHLC TOOL: USING RTL2MμPATH TO SYNTHESIZE LEAKAGE SIGNATURES FROM PROCESSOR RTL

We present SYNTHLC, an automated approach and tool for synthesizing a comprehensive set of formally verified leakage signatures from a SystemVerilog processor design, from which one can derive the leakage contracts in Table I. First, SYNTHLC uses RTL2MμPATH to uncover all μPATHs for each instruction implemented on the design (§V-B). Instructions with more than one μPATH are *candidate transponders*. Second, SYNTHLC uses a symbolic information flow analysis to classify each candidate transponder's decisions as dependent on the unsafe operand(s) of (typed) transmitter(s) or not (§V-C). The result is a set of true transponders with leakage signatures that characterize all of their transmitter operand-dependent μPATH variability.

A. Inputs & Metadata Requirement

Both SYNTHLC and RTL2MμPATH require three inputs: the SystemVerilog *design under verification* (DUV), a list of *encodings* for each implemented instruction, and design *metadata*. We detail our metadata requirement below, which follows from the fact that these tools make extensive use of model checkers to evaluate auto-generated LTL properties, formulated as SystemVerilog Assertions (SVAs) [2]. Table II quantifies this metadata for the CVA6 Core and Cache (§VI).

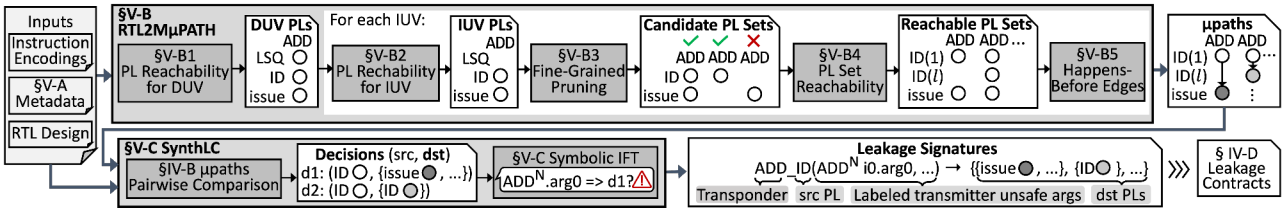


Fig. 6: RTL2MμPATH (top) and SYNTHLC (bottom) synthesize a complete set of formally-verified leakage signatures.

First, the designer must identify the *instruction fetch register* (IFR) [46], which holds fetched instruction encodings before they are supplied by the processor frontend to the backend. RTL2MμPATH uses the IFR to constrain the execution traces considered by a model checker, e.g., to those that feature some specific *instruction under verification* (IUV) [32], [46], [73].

RTL2MμPATH uses a single IID to track an IUV as it progresses through various PLs during its execution on the DUV (§III-C)—its PC. Hence, RTL2MμPATH requires each μFSM's IIR to be a *program counter register* (PCR) [32], [46], [73], which contains the PC of the instruction occupying it. PCRs may be present in the original DUV or added in parallel to the IIRs in the DUV solely for verification [32], [46], [73].[2] Fig. 3 shows an excerpt of two CVA6 design files where we add PCRs in parallel to existing IIRs.

Once the DUV is augmented (if necessary) with PCRs, RTL2MμPATH requires the user to supply all μFSMs as tuples of signal names $\langle$`pcr`, `vars`$\rangle$ that denote their PCR and state variable components (§III-C). Plus, since RTL2MμPATH considers invalid any PL $\langle \mu$`fsm`, `state`$\rangle$ where `state` $=$ *idle*, the user is also required to supply each μ`fsm`'s *idle* state(s). For CVA6, there are 21 PCRs, and thus μFSMs, in total; we add 14 PCRs to the baseline design.

To detect when instructions commit, RTL2MμPATH requires the user to supply the DUV's `commit` signal.

Two final inputs support SYNTHLC's symbolic information flow analysis. First, the user must identify the *architectural register file* (ARF) and *architectural main memory* (AMEM) to block taint propagation between instruction outputs/inputs. Second, they must identify *operand registers*, located at the issue or register read stage, to enable taint introduction for transmitter operands.

B. RTL2MμPATH Tool: Synthesizing μPATHs from RTL

For each IUV, taken from the input list of instruction encodings, RTL2MμPATH finds all of its μPATHs by using model checkers to explore its execution behavior in *all reachable contexts*, starting from a *valid reset state*, i.e., a hard processor reset, where *only* architectural state is symbolically initialized. All reachable contexts indicates that the IUV may be preceded/followed by an arbitrary number of valid instructions.

We explain RTL2MμPATH's synthesis procedure below, as depicted in Fig. 6. Each step involves instantiating numerous

[2]Such auxiliary state elements exist exclusively within the verification environment, and are removed prior to synthesis and fabrication.

Identified in CVA6 Core							
IFR	IIRs (PCRs)	μFSMs	PCRs	commit	operand	ARF	AMEM
1 reg	21 (7) regs	38 regs	21 regs*	1 wire	2 regs	1 array	1 array

Added to Core	
PCRs	SV
14 regs	39 LoC

Added to Cache	
PCRs	SV
9 regs	74 LoC

Identified in CVA6 Cache		
IIRs (PCRs)	PCRs	μFSMs
9 (0) regs	9 regs*	13 regs

TABLE II: User annotations required by SYNTHLC (§V-A) and all/some [31], [32], [36] of prior works (§VIII). *: With added ones. μFSMs signals correspond to their state variables (§III-C).

SVA properties from templates. Important SVA syntax for understanding our templates includes the notions of **`cover`** and **`assume`** statements. A **`cover`** property directs a model checker to search for *any* execution trace that satisfies a given condition. A *reachable* outcome is returned when a trace is found. An *unreachable* outcome is a proof that no such trace exists. An *undetermined* outcome indicates that a satisfying trace cannot be found due to a timeout or resource constraints, but it *may* exist. We discuss implications of undetermined model checker outcomes in §VII-B4. SVA **`assume`** statements constrain the execution traces considered by a model checker to those that satisfy their specified condition when evaluating SVA properties (e.g., **`cover`** properties).

1) PL Reachability for DUV: For each μ`fsm` $= \langle$`pcr`, `vars`$\rangle$ (§III-C), RTL2MμPATH enumerates its feasible PLs by considering all constant valuations of `vars` and excluding user-identified *idle* states. Then RTL2MμPATH instantiates SVA properties to prune those PLs that are proven *unreachable* on the DUV *by any instruction*. The remaining **DUV PLs** are those PLs reachable by *some* IUV (i.e., *some* IUV can visit them). Our CVA6 case study has a total of 41 DUV PLs.

Next, RTL2MμPATH conducts several *IUV-specific* analyses.

2) PL Reachability for IUV: Like the first step, but conditioned on a specific IUV, RTL2MμPATH instantiates SVA properties to discard from consideration PLs that are proven *unreachable* by the IUV, producing a set of **IUV PLs**. For example, in Fig. 6, the `LSQ` PL is part of the set of DUV PLs, but not included in the set of IUV PLs for an `ADD`.

3) Fine-Grained Pruning: For an IUV, our first goal is to derive its **Reachable PL Sets**. A Reachable PL Set is a set of PLs that is *exclusively* visited in one of the IUV's executions, i.e., there exists an execution of the IUV that visits all of the PLs in the set and no others.

Deriving all Reachable PL Sets for an IUV, naïvely, requires asking a model checker to consider each element in the *power*

set of the IUV PLs and deduce its reachability. RTL2MμPATH prunes this power set in two ways using SVAs: it removes elements from the power set that contradict *dominates* and *exclusive* relationships between PLs. We say pl_0 *dominates* pl_1 iff all executions of the IUV that visit pl_0 also visit pl_1. We say pl_0 and pl_1 are (mutually) *exclusive* iff there exists no execution of the IUV that visits both pl_0 and pl_1.

RTL2MμPATH deduces dominates (exclusive) relationships by instantiating and evaluating the top (bottom) SVA property template below for each ordered (unordered) pair of IUV PLs. In all such listings in the paper, `blue` (**`brown`**) terms are template arguments (SystemVerilog or SVA keywords).

```
pl_0_dom_pl_1: cover (!pl_0_visited & pl_1_visited);
pl_0_excl_pl_1: cover (pl_0_visited & pl_1_visited);
```

An *unreachable* outcome for the top (bottom) property proves there exists no execution trace where the IUV visited pl_1 but not pl_0 (visited both), thus helping to prune PL Sets.

4) PL Set Reachability: For each **Candidate PL Set** that remains after pruning, RTL2MμPATH instantiates this property:

```
// {pl_0, pl_1, ..., pl_n}: IUV PLs
// cand_pl_set consists of pl_0, pl_1, ..., pl_j
assume (!pl_{j+1} & !pl_{j+2} ... & !pl_n);
cand_pl_set: cover (pl_0_visited & pl_1_visited & ... &
     pl_j_visited & !(pl_0 | pl_1 | ... | pl_j));
```

A *reachable* outcome indicates an execution trace exists where, by the time the IUV has disappeared from the processor (`!(pl_0 | ... )`), it has visited exclusively PLs in the Candidate PL Set (`cand_pl_set`).

Next, RTL2MμPATH iterates over the elements of each **Reachable PL Set** (discovered above) and uses SVAs to determine which constituent PLs may non-consecutively or consecutively be revisited (§III-B). Non-consecutively revisited PLs are simply marked as such. Consecutively revisited PLs are duplicated and tagged as representing the first/last consecutive visit, e.g., `ID(1)`/`ID(`l`)` in Fig. 4. Knowing *which* PLs may be non-consecutively or consecutively revisited is sufficient to place HB edges and uncover all *decisions* across an IUV's μPATHs—what SYNTHLC ultimately derives from RTL2MμPATH's output (§IV-B). So, RTL2MμPATH can be configured to avoid deducing the exact number of revisits per PL (e.g., all possible values of l for `ID(`l`)`) as an optimization. The majority of our experiments in do this (§VI).

5) Happens-Before Edges: RTL2MμPATH extends each Reachable PL Set to a full μPATH by deriving a partial order on visited PLs. RTL2MμPATH considers as candidate HB edges all ordered pairs of PLs that are connected via pure combinational logic in the DUV to capture *causal* happens-before relationships among visited PLs exclusively. Each candidate edge is evaluated for every Reachable PL Set using SVAs to determine if it constitutes a true HB relationship.

6) All Cycle-Accurate μPATHs: While not needed by SYNTHLC, RTL2MμPATH can be configured to uncover: (i) for each IUV, for each PL that it may revisit, the set of *revisit cycle counts* for the PL that the IUV can exhibit across all of its executions, or (ii) all μPATHs that concretize the precise number of visits to each revisited PL (as in Fig. 1, 4, 2b, and 2c). We direct RTL2MμPATH to perform (i) for CVA6 to support SDO (Table I). RTL2MμPATH is currently not optimized for (ii), which generates *many* (still easy-to-check) properties, proportional to the cross product of distinct concrete values per PL derived in (i). We expect we can drastically prune these properties (as in §V-B3), but it is unnecessary for our focus: leakage contract verification.

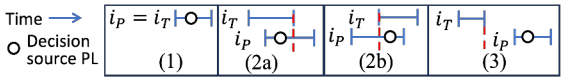


Fig. 7: Constraints on SVA properties to classify transmitters as (1) intrinsic, (2a/b) older/younger dynamic, or (3) static.

C. Attributing μPATH Variability to Transmitters

After RTL2MμPATH uncovers all μPATHs for each implemented instruction on the DUV, SYNTHLC identifies candidate transponders and collects all decisions for each (§IV-B).

1) Symbolic IFT: For each candidate transponder P, SYNTHLC considers each of its decisions $(src, \mathbf{dst}) \in \mathbf{d}_M^P$ (M is the DUV) in turn to determine if P exhibits $(src, \mathbf{dst})$ as a function of some transmitter T's unsafe operand op. All possible (T, op) pairs are considered for each decision.

To do this, SYNTHLC first augments the DUV with cell-level *information-flow tracking* (IFT) circuitry, which supports per-data-bit introduction and propagation of *taint* to track the explicit and implicit flows of certain data dynamically at runtime [78]. Next, it uses a model checker to consider a dynamic instance of P, i_P, and a dynamic instance of T, i_T, executing together in all reachable contexts following a valid reset state (§V-B), subject to three assumptions (Fig. 7). In all cases, each data bit is assigned one taint bit, taint is introduced for op of i_T *exclusively*, and taint is prohibited from propagating architecturally between instruction outputs/inputs.

Assumption 1 determines if T is an intrinsic transmitter with respect to $(src, \mathbf{dst})$ by constraining i_T and i_P to be the same dynamic instruction. *Assumption 2*, which is composed of sub-assumptions 2a and 2b, determines if T is a dynamic transmitter by constraining i_T to be in-flight when i_P visits its decision source src. Sub-assumption 2a/2b considers the case where i_T is fetched before/after i_P (i.e., i_T is older/younger than i_P). *Assumption 3* determines if T is a static transmitter by constraining i_T to have materialized and dematerialized in M *before* i_P reaches src.

Note that the third assumption uses one additional taint bit per data bit to support flushing "sticky" taint that is associated with op's dynamic influence on transponders' μPATHs, thereby considering its static influence exclusively.

The SVA template below has two **`assume`**s. The first one introduces taint (exclusively) at the register corresponding to op (§V-A), when i_T is at the issue stage. The second restricts execution traces to those satisfying one of three aforementioned assumptions. The **`cover`** property searches for an execution trace where i_P visits src (`src_pl`) once cycle before (`##1`) it visits all of the PLs in $\mathbf{dst}$ (`(dst_pl_0 & ...)`) *and* the μFSMs of these decision destinations are tainted—signaling a dependence on i_T's operand op.

```
// Candidate transponder decision (src_pl, {dst_pl_0, ...})
assume ((iT_at_issue) ^ (op_reg_taint == 1));
assume (assumption 1/2/3);
decision_taint: cover (src_pl ##1 ((dst_pl_0 & ...) &
 (dst_pl_0_taint | ... )));
```

A *reachable* outcome results in assigning a *tag* to P's decision $(src, \mathbf{dst})$, denoting that it is dependent on typed (intrinsic / older or younger dynamic / static) transmitter T's unsafe operand op. An *unreachable* outcome assign no tag.

After using the property above to evaluate every one of P's decisions (under all three assumptions, for every possible (I, op) pair), we can construct a leakage signature for each of P's decision sources as follows (§IV-D). If P exhibits at least two transmitter operand-dependent decisions with respect to $src \in \mathbf{src}_M^P$, we construct a leakage signature corresponding to leakage function `P_src` (function name).[3] Examining tags assigned to all such decisions (those involving src) gives us typed transmitters (explicit inputs) and unsafe transmitter arguments (in the function body). Decision destinations (return values) are all sets of PLs $\mathbf{dst}$ such that $(src, \mathbf{dst}) \in \mathbf{d}_M^P$.

Notably, SYNTHLC's symbolic IFT step also uncovers implicit inputs (Fig. 5) to leakage functions, which may be useful towards implementing the hardware side-channel defenses in §II-B using the leakage contracts in Table I.

2) Security Argument: A proof in our repository [1] shows that SYNTHLC produces a set of leakage signatures that capture all violations of *hardware side-channel safety*, as defined below, subject to a receiver $R_{\mu\text{PATH}}$. $R_{\mu\text{PATH}}$ observes the PLs occupied by in-flight instructions in each cycle, modeling a receiver that perceives channel modulations via their impact on instruction/program execution time or resource contention.

Definition V.1 (Hardware Side-Channel Safe)**.** A microarchitecture M is *hardware side-channel safe* with respect to receiver R, or SC-Safe(M, R), iff:

$$\forall p.\forall \pi.\forall \sigma, \sigma'.\forall \mu. ArchCtrl(p) \implies (\sigma \approx_\pi \sigma' \implies O_R(\{\!|p|\!\}_M^{\langle\sigma,\mu\rangle}) = O_R(\{\!|p|\!\}_M^{\langle\sigma',\mu\rangle})) \tag{V.1}$$

Eq. V.1 quantifies over all programs p and security policies π (which label program inputs as public or secret), all pairs of initial architectural states σ, σ' and all initial microarchitectural states μ. Looking at the second line, the antecedent, $\sigma \approx_\pi \sigma'$, checks that the initial architectural states are *low-equivalent* with respect to π: they agree on the values of low data in π, i.e., p's public data inputs. The consequent, $O_R(\{\!|p|\!\}_M^{\langle\sigma,\mu\rangle}) = O_R(\{\!|p|\!\}_M^{\langle\sigma',\mu\rangle})$, asserts that R obtains identical observation traces when p runs on microarchitecture M from initial states $\langle\sigma, \mu\rangle$ and $\langle\sigma', \mu\rangle$. Given our focus on *microarchitectural* (not architectural) side channels, the first line requires p to feature the same sequence of instructions along all branches of secret-dependent control-flow instructions ($ArchCtrl(p)$).

Eq. V.1 violations indicate that the observation trace obtained by receiver R from running program p with privacy policy π on microarchitecture M is *indisputably* a function of p's high inputs. Note that secret-dependent control-flow instructions can still behave as microarchitectural transmitters and cause Eq. V.1 violations, e.g., if they create operand-dependent squashes. Moreover, virtually all microarchitecture, for any realistic receiver, will trigger Eq. V.1 violations—the goal of a leakage contract is to account for them all.

[3] A single decision may be tagged transmitter operand-dependent due to imprecision of symbolic IFT (§V-C). At least two decisions must be operand-dependent to yield >1 receiver observations as a function of operand values.

VI. SYNTHESIZING LEAKAGE SIGNATURES FROM CVA6

We use SYNTHLC to synthesize leakage signatures from the RISC-V CVA6 CPU (commit `#00236BE`), considering all 72 instructions in the RV64I ISA and M extension (**RV64IM**).

CVA6 [100] is a 64-bit, 6-stage, single-issue RISC-V core featuring speculation and limited out-of-order write-back with diverse functional units (ALU, LSU, Mul/Div, CSR buffer). It has a FIFO scoreboard (SCB) that tracks instructions from issue to commit and retires instructions in-order. Functional units may complete out-of-order in the SCB provided older, non-retired instructions' destination registers do not match.

We configure the design as follows. Both speculative and committed store buffers (STBs) are sized to two entries. The SCB is sized to four entries, but due to a bug in CVA6 that we discovered during our case study, only three entries are ever occupied at a time by active instructions (§VII). Such down-scaling is typical in formal verification [33], [34]. We configure CVA6 without a memory management unit, and our main experiment instantiates the CVA6 Core as the DUV. Another experiment instantiates the CVA6 Cache (L1 data cache and cache controller) as the DUV. CVA6 does not come with a Verilog behavioral model of memory, so we add a single-port RAM behavioral model consisting of 32 64-bit words for the CVA6 Core DUV; the load-store unit (LSU) and committed STB are modified to directly interface with memory.

The elaboration step black-boxes the design's frontend, since RTL2MμPATH drives all issued instructions at the IFR with a model checker. The multiplier is also black-boxed to reduce verification complexity. Since RTL2MμPATH explores control state behavior, purely combinational logic circuits can be safely abstracted—*a key benefit*. The core design features 8,577 lines of SystemVerilog; after elaboration there are 22,138 wires, 19,575 standard cells, 482 registers (11,985 D flip-flop bits), and 3 memory arrays (including ARF and AMEM in Table II). The data-cache is 4-way, 128B (scaled down from 32 KB), featuring 2,279 lines of SystemVerilog.

We supply required metadata from §V-A to RTL2MμPATH and SYNTHLC (Table II). Our implementations of both tools use SVA 2009 [2] and the JasperGold v21.03 property verifier [22]. We use Verific [13] and Yosys [92] to parse SystemVerilog and instrument the DUV with IFT logic using CellIFT [78]. All experiments are run on three compute nodes, each of which features two 32-core 2.9GHz Intel Xeon CPUs with 512GB RAM.

VII. RESULTS & DISCUSSION

SYNTHLC synthesizes a complete set of leakage signatures for the CVA6 Core (considering all instructions), and a partial set for the CVA6 Cache (considering loads and stores).

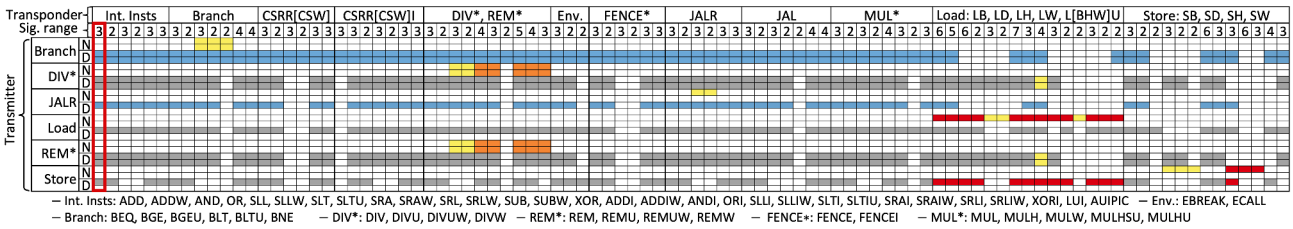


Fig. 8: SYNTHLC CVA6 Core results. Transponders (coarse-grained columns) and their leakage signatures with output range sizes (fine-grained columns), plus explicit inputs from intrinsic/dynamic transmitters in N/D-labeled rows, where the top/bottom sub-row is rs1/rs2. We distinguish secondary leakage (§VII-A1) and false-positive leakage (§VII-B1) from primary leakage (§VII). Primary leakage is categorized as involving explicit channels implicit channels, or explicit branches using STT [99] terminology.

A. Summary of Results

We first discuss the transmitters and transponders surfaced by SYNTHLC in our experiments.

1) Transponders and Transmitters: CVA6 Core: Fig. 8 summarizes SYNTHLC's synthesis results for the CVA6 Core. Coarse-grained row/column types denote transmitters/transponders. Fine-grained row and column labels, respectively, denote transmitter types (intrinsic/dynamic) and ranges for distinct leakage signatures. The top/bottom sub-row for each fine-grained row indicates transmitter operand rs1/rs2.

SYNTHLC finds transponders and leakage signatures per §V. We observe that (i) classes of transponders feature identical leakage signatures, and (ii) classes of transmitters are explicit inputs to the same leakage signatures where they feature identical types. So, Fig. 8 groups transponders and transmitters accordingly. Each fine-grained column represents a leakage signature P_src, where *P* can be any transponder in the class represented by the coarse-grained column label. Colored cells within a column indicate P_src's explicit inputs having intrinsic/dynamic transmitters on N/D crossing rows.

As an example, consider the leftmost fine-grained column, outlined in red. It corresponds to a leakage signature ADD_ID that SYNTHLC synthesizes for ADD transponders on the CVA6 CPU. ADD_ID may output one of three decisions for ADDs with respect to decision source ID. The top-/bottom-most colored cell in the column indicates that operand rs1 of a dynamic branch/store is an explicit input to ADD_ID. Overall, this column indicates that an ADD exhibits three-way μPATH variability at ID as a function of dynamic branch/division/remainder operands (both rs1 and rs2), load/store operands (rs1, the base address), and JALR operands (rs1, the target address).

Colored Fig. 8 cells represent primary (orange, red, blue) versus secondary (gray) leakage. Primary/secondary leakage indicates that the transponder (column) can/cannot leak the transmitter's (row) unsafe operand without the presence of other transponders. Secondary leakage often arises due to shared resources, e.g., an ADD that is stalled from committing at the SCB, stuck behind an intrinsic transmitter (e.g., DIV).

SYNTHLC flags all 72 evaluated instructions as transponders and finds that the CVA6 core features intrinsic and dynamic transmitters exclusively (hence the omission of static transmitter labels in Fig. 8). **Nineteen** intrinsic transmitters are found: eight division (DIV) and remainder (REM) variants, seven load (LD) variants, and four store (ST) variants. **Twenty-six** dynamic transmitters are found: all intrinsic transmitters plus six branch variants and JALR. Notably, all intrinsic transmitters except stores can exhibit execution time variability as a function of their operands. The paragraphs below summarize key findings, organized around classes of transponders.

Load: On CVA6, a LD transponder may exhibit several decisions at issue, including proceeding to destinations {ldFin}/{LSQ, ldStall} as described in §IV-A as a function of rs1 of the LD itself (LD^{N}) and rs1 of a dynamic store (ST^{D}_{0}).

Store: A ST transponder exhibits μPATH variability following a PL in the committed STB (comSTB), where it stalls if a *younger* in-flight load with a different address is ready to access the single-R/W-port memory; CVA6 prioritizes serving the younger load. The leakage signature (ST_comSTB in Fig. 5) output depends on rs1 of the ST itself (ST^{N}) and rs1 of a dynamic load ($\text{LD}^{\text{D}}_{\text{Y}}$). *We are the first to uncover this channel* when conducting CV6 leakage contract verification [31], [32].

Interestingly, this channel renders CVA6 susceptible to a *new* class of speculative interference attacks [17], involving *transient dynamic transmitters* (LDs, in the shadow of older excepting instructions) that create μPATH variability for older, *committed* transponders (STs). Since this μPATH variability takes place after STs commit, it does not impact their execution time. Classic variants [17] involve *transient intrinsic transmitters* whose own μPATH variability creates timing-differentiable μPATH variability for older, *bound-to-commit* transponders. Using transponders, we can more generally define speculative interference attacks as involving transient transmitters that create μPATH variability for older non-transient transponders.

Division/Remainder: SYNTHLC flags all DIV/REM variants as intrinsic transmitters, and thus, transponders. Both use serial division circuitry, taking one to sixty-six cycles to compute their results (based on revisit cycle counts, §V-B6).

All: All transponders (all instructions) can be stalled in ID (from issuing) or scbFin (from committing) as a function of the operand(s) of dynamic LD, ST, DIV, and/or REM transmitters. The stall in ID/scbFin can be 1 to 68/4 consecutive cycles. They may also be flushed at almost any PL as a function of dynamic branch or JALR transmitter operands: all six branches and JALR are flagged as dynamic transmitters. The one exception is LD transponders; once LDs visit certain

PLs in the load unit (ldStall, ldFin), they cannot be flushed until they exit the load unit. Branches, as a function their rs1 and rs2 operands, and JALR, as function of its rs1 operand, flush a transponder upon a mis-prediction. Prior work classifies branch and JALR operands as unsafe on CVA6, but cannot deduce why [32]. Note that no control-flow instruction is flagged as a static transmitter, because we black-box the CVA6 Core front-end (§VI), where predictor structures reside.

2) Transponders and Transmitters: CVA6 Cache: SYNTHLC is *not* a modular verification procedure. However, we use the SYNTHLC approach to conduct a *conservative* and *partial* security evaluation of the CVA6 Cache in order to show that it can: (i) analyze a realistic cache, which no prior leakage contract verification work has attempted [31], [32], [85], [86], [91]; (ii) handle *non-consecutive* re-visit behavior (§III-B), which exists in the Cache DUV only; (iii) benefit from modularity from a scalability perspective (§VII-B3).

First, SYNTHLC collects all LD/ST decisions based on the RTL2MμPATH's analysis results on the Cache DUV. Second, we select three source PLs apiece for LD/ST—the three with the highest number of destination PL sets (four on average). Third, we instantiate a Core+Cache DUV and check that these reachable Cache decisions are reachable on the full design; all pass this check in our experiment. Finally, SYNTHLC conducts its symbolic IFT step on the Cache DUV to produce leakage signatures for all six decision source PLs.

Interestingly, all leakage signatures for LD (ST) transponders have *identical* explicit inputs, which flag every relevant transmitter type—intrinsic/dynamic/static LDs (STs) and dynamic/static STs (LDs)—as leaking its address operand. Plus, SYNTHLC uncovers channels involving hardware structures in nearly all Cache files, specifically: tag banks, fully-associative write buffer, MSHRs, shared ports to the AXI interface.

Fig. 4c shows some μPATHs for stores. A ST visiting wBVd, where it accesses the cache, may exhibit several decisions, including progressing to destinations {wRTg, wr$0}/{wRTg} in the left/right μPATH upon a cache miss/hit. The synthesized leakage signature ST_wBVd, shown in Fig. 5, flags LDs as *static* transmitters (LD^{S}), but not STs (since the cache is no-allocate on write), and the ST itself as an intrinsic transmitter (ST^{N}).

Results from our Cache evaluation are conservative (sound but incomplete, §VII-B4), since symbolic IFT can *possibly* flag transmitter-transponder interactions that are possible on the Cache DUV, but not the Cache+Core. We manually inspect results and confirm that all flagged leakage looks plausible.

B. Discussion

1) False-Positives from IFT: SYNTHLC exhibits some false positives (Fig. 8) due to IFT imprecision. Interestingly, it *does not* identify any false-positive transmitters. It *does*, however, identify a handful of false-positive transmitter-transponder associations, i.e., indicating that some transponder's decision is operand-dependent on some (intrinsic/dynamic) transmitter's operand when it is not. In particular, 14/94 (1/6) unique leakage signatures obtained from the CVA Core (Cache) include extraneous explicit inputs. So, a few extraneous (benign) dynamic/intrinsic transmitters are flagged. We find these cases arise when distinct μFSMs in the same structure (e.g., in different SCB entries) share fan-in signals, which are updated as a function of transmitter operands, causing over-taint.

2) CVA6 Bugs: RTL2MμPATH/SYNTHLC helped us identify three new functional bugs in CVA6 (two have security implications), involving JAL/JALR/branches [45].

RTL2MμPATH finds that following its visit to the scbFin (*SCB finished* PL), JALR never progresses to scbExcp (*SCB exception* PL), while JAL and branches sometimes do. The RISC-V ISA requires that JAL/JALR/branches all trigger exceptions if their target addresses are not 4-byte aligned. From inspection, we find CVA6 does not enforce any alignment restrictions for JALR (as its μPATHs indicate). While investigating JALR, we also notice that JAL only enforces 2-byte alignment checks. Notably, these functional bugs can expand the attack surface for control-flow hijacking attacks [24].

SYNTHLC reports that whether a conditional branch progresses to scbCmt (*SCB commit* PL) or scbExcp following scbFin is *independent* of its operands. But, RISC-V requires that branches raise misaligned target exceptions only when their (operand-dependent) outcome is taken. From inspection, we find that branches incorrectly raise exceptions whenever their target address is misaligned, regardless of their outcome.

From the RTL waveforms produced by RTL2MμPATH's reachable SVA cover properties (§V-B), we observe that the SCB is always underutilized by one entry. We localized this counterintuitive behavior to an incorrect counter width declaration in the CVA6 Core. By the time we noticed this microarchitectural bug, it had been fixed (commit #5c0dc19).

3) Property Evaluation Performance: For μPATHs synthesis for the CVA6 Core, RTL2MμPATH evaluates 124,459 properties in 4.43 minutes per property on average under a 30 minute time-out; 16.39% of properties per instruction are undetermined. RTL2MμPATH/SYNTHLC can be configured to interpret undetermined model checker outcomes (§V-B) as *reachable* or *unreachable*. We do the latter (§VII-B4). For leakage signature synthesis for the Core, SYNTHLC evaluates 30,774 (additional) properties for 2.35 minutes per property on average under the same time-out; 13.74% are undetermined. Interestingly, for the Cache, *all* 4,178 properties evaluated by RTL2MμPATH/SYNTHLC complete within 3 seconds on average, highlighting the benefits of modularization. We use RTL2MμPATH to uncover revisit cycle counts (§V-B6) on the Core only, evaluating 8,043 additional properties in 32 seconds on average with a 10 minute time-out; 0.7% are undetermined.

4) Soundness/Completeness: When discussing *theoretical* guarantees of RTL2MμPATH/SYNTHLC, we assume that there are no undetermined model checker outcomes. Recall that all SVAs are evaluated from a valid reset state of the DUV (§V-B).

The RTL2MμPATH procedure is theoretically sound: if it outputs μPATH p for instruction I, then an execution trace where I exhibits p, was deemed *reachable* on the DUV. And it is theoretically complete: if it does not output μPATH p for instruction I, then such a trace was deemed *unreachable*.

Interpreting undetermined model checker outcomes as unreachable (§VII-B3) impacts RTL2MμPATH's completeness guarantee. If RTL2MμPATH does not output μPATH p for instruction I due to an undetermined outcome, there is a chance that I exhibits p in some reachable trace. Our manual inspection of RTL2MμPATH's output for CVA6 suggests most undetermined μPATHs would eventually resolve as unreachable. Such μPATHs usually feature instructions visiting PLs in unrelated functional units (e.g., an `ADD` visiting a STB PL).

The SYNTHLC procedure is theoretically sound: If it classifies some candidate transponder P's decisions at source PL src as independent of some transmitter T's unsafe operand op, then it is indeed independent. That is, an execution trace (on the IFT-augmented DUV), where op introduces taint, P exhibits decision $(src, \mathbf{dst})$, and $\mathbf{dst}$ is becomes tainted, was deemed *reachable* for *at most one* such decision and *unreachable* for all others (§V-C1). However, SYNTHLC is theoretically incomplete: such an execution trace may be deemed *reachable* for *more than one* such decision even if P's decisions at src are *independent* of op, due to imprecision of symbolic IFT.

We are optimistic that we can leverage established IFT techniques to minimize false positive leakage flagged by SYNTHLC, e.g., custom taint propagation rules to reduce taint spread [75] or specialized taint flushing mechanisms to prevent over-taint in shared buffers (e.g., the SCB) [77].

Interpreting undetermined model checker outcomes as unreachable impacts SYNTHLC's soundness guarantees. That is, if SYNTHLC classifies some transponder's decision at some source PL as independent of some transmitter's unsafe operand due to an undetermined outcome, it may actually be dependent in some reachable trace. Nevertheless, our cache evaluation suggests that this issue can be addressed through DUV decomposition and modular verification.

In practice, SYNTHLC automatically localizes side-channel leakage in RTL with exceptional precision.

VIII. Related Work

Contract Verification: No existing approaches can verify hardware adherence to leakage contracts as detailed as leakage signatures. The closest prior work verifies hardware adherence to CT contracts [32], [36], [38], [91] by checking some version of Eq. V.1 *in one shot* with varying receiver assumptions using a *product circuit* formulation—*two copies* of the DUV are instantiated and analyzed together. These methodologies are all theoretically incomplete due to their use of *symbolic initial states* (*all* registers are symbolic), which enable the product circuit approach to scale at the cost of *false counterexamples*. Notably, the user must manually inspect counterexamples to classify them as false and add constraints to avoid them. All but UPEC-DIT [32] are theoretically sound, i.e., ignoring model-checker limitations.

UPEC-DIT [32] is a CT *contract synthesis* approach. LEAVE [91] and UPEC-DIT-23 [31] are CT *contract satisfaction* approaches. CONJUNCT [36] is both.

UPEC-DIT [32] inputs include the DUV's operand registers, `commit` signal, existing PCRs, ARF, AMEM, and IFR (§V-A). During synthesis, the user manually classifies registers as *control*/*data* upon counterexample inspection, attributes control value divergence to instruction operands, and writes assumptions to avoid spurious counterexamples. For CVA6, this amounts to refining over a hundred lines of SVA properties [6].

LEAVE's inputs include all of UPEC-DIT's, except PCRs, plus design invariants and extra internal design signals that may hold unsafe transmitter operands. It tries to automatically refine the invariants to construct an inductive proof, but requires updating them upon a (manually detected) spurious proof failure. Note, "[LEAVE] does not yet scale to processors of the complexity of, e.g., the CVA6 core" [66]. Over twenty lines of invariants are needed for an Ibex core variant [5], [54].

UPEC-DIT-23's inputs include UPEC-DIT's plus a list of transmitters. During verification, the user writes invariants to exclude false counterexamples and performs manual tasks from UPEC-DIT. Over a hundred of lines of invariants [7] are needed in addition to the property refinement from UPEC-DIT.

CONJUNCT [36] inputs are the same as UPEC-DIT's. Its synthesis step classifies each instruction as a candidate (non-)transmitter via a bounded analysis. Its verification step uses automatically-generated candidate invariants to conduct unbounded verification that the set of non-transmitters indeed contains no transmitters. Upon misclassification (during synthesis) or proof failure (during verification), the user inspects traces and constructs assumptions to avoid false positive transmitters or invariants to derive complete proofs, respectively. The authors evaluate in-order pipelines, where few non-transmitters are misclassified as transmitters. One pipeline required carefully-constructed assumptions to cull false positives due to unreachable states. Out-of-order designs (that use SCBs, ROBs, etc.) will require more of these constraints, and therefore commensurately more manual effort.

IX. Conclusion

We design the first automated approach and tool for uncovering all microarchitectural execution paths for instructions as implemented on a particular SystemVerilog processor design. In doing so, we observe that such path variability can be used to localize hardware side channels—a requirement for designing defenses against hardware side-channel attacks. Subsequently, we design the first automated approach and tool for synthesizing microarchitectural leakage contracts, required by ten defenses, from processor RTL.

Acknowledgements

We thank Mohammad Rahmani Fadiheh, Sally Wang and the anonymous reviewers for their constructive comments and feedback. This work was supported in part by the National Science Foundation (NSF), under awards 2153936, 1954521, 1942888, 2154183, 8191902, 2321489, and 2236855 (CAREER), and the Defense Advanced Research Projects Agency (DARPA) under contract W912CG-23-C-0025 and subcontract from Galois, Inc. We gratefully acknowledge a Verific Design Automation academic license and gifts from Apple and Intel.

References

[1] "SynthLC Github Repository," https://github.com/yaohsiaopid/SynthLC.

[2] "Institute of electrical and electronic engineers (IEEE) standard for SystemVerilog–unified hardware design, specification, and verification language," 2009.

[3] "Risc-v cryptography extensions volume i scalar & entropy source instructions," 2021.

[4] "Configuring workloads for microarchitectural and side channel security," 2023, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/securing-workloads-against-side-channel-methods.html.

[5] "LeaVe - ibex-mul-div," 2024, https://github.com/zilongwang123/LeaVe/blob/main/config/Ibex-mult-div.yaml.

[6] "UPEC DIT – CVA6," 2024, https://github.com/RPTU-EIS/UPEC-DIT/blob/master/examples/processor-cores/CVA6/ariane_dit.sva.

[7] "UPEC DIT arxiv 23 – CVA6," 2024, https://github.com/RPTU-EIS/UPEC-DIT/blob/master/examples/processor-cores/CVA6/ariane_dit_1-cycle.sva.

[8] O. Aciicmez, J.-P. Seifert, and C. K. Koc, "Predicting secret keys via branch prediction," *IACR'06*, 2006.

[9] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," IACR'18, 2018.

[10] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On subnormal floating point and abnormal timing," in *2015 IEEE Symposium on Security and Privacy*, 2015.

[11] Arm, "Arm a64 instruction set architecture." https://static.docs.arm.com.

[12] Arm, "ARM7TDMI (rev 3) core processor, instruction speed summary," 2007, urlhttps://developer.arm.com/documentation/dvi0027/b/arm7tdmi/instruction-speed-summary.

[13] V. D. Automation, "Verific's parser platform," 2019.

[14] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic side-channel attacks on printers," *19th USENIX Security Symposium*, 2010.

[15] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[16] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Specshield: Shielding speculative data from microarchitectural covert channels," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019.

[17] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison, F. Mckeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. Alameldeen, "Speculative interference attacks: Breaking invisible speculation schemes," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[18] D. J. Bernstein, "Cache-timing attacks on aes," Tech. Rep., 2005, https://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[19] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. Emer, and M. Yan, "Casa: End-to-end quantitative security analysis of randomly mapped caches," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 1110–1123.

[20] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "Mi6: Secure enclaves in a speculative out-of-order processor," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[21] D. Brooks and M. Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," in *Proceedings Fifth International Symposium on High-Performance Computer Architecture*, 1999.

[22] Cadence Design Systems, Inc., "Cadence JasperGold formal verification platform," accessed 12$^{\text{th}}$ April 2021. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html

[23] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 249–266.

[24] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "{Control-Flow} bending: On the effectiveness of {Control-Flow} integrity," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 161–176.

[25] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "Fact: A flexible, constant-time programming language," *SecDev'17*.

[26] R. Choudhary, J. Yu, C. Fletcher, and A. Morrison, "Speculative privacy tracking (spt): Leaking information from speculative execution without compromising privacy," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 607–622.

[27] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1512–1542, 1994.

[28] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 2000.

[29] F. Denis, "libsodium," 2019, https://github.com/jedisct1/libsodium.

[30] P. W. Deutsch, W. T. Na, T. Bourgeat, J. S. Emer, and M. Yan, "Metior: A comprehensive model to evaluate obfuscating side-channel defense schemes," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–16.

[31] L. Deutschmann, J. Mueller, M. R. Fadiheh, D. Stoffel, and W. Kunz, "A scalable formal verification methodology for data-oblivious hardware," *arXiv preprint arXiv:2308.07757*, 2023.

[32] L. Deutschmann, J. Müller, M. R. Fadiheh, D. Stoffel, and W. Kunz, "Towards a formally verified hardware root-of-trust for data-oblivious computing," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022.

[33] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol verification as a hardware design aid." in *ICCD*, vol. 92. Citeseer, 1992, pp. 522–525.

[34] D. L. Dill and J. Rushby, "Acceptance of formal methods: Lessons from hardware design," *IEEE Computer*, vol. 29, no. 4, pp. 23–24, 1996.

[35] S. Dinesh, G. Garrett-Grossman, and C. W. Fletcher, "SynthCT: Towards portable constant-time code," in *NDSS*, 2022.

[36] S. Dinesh, M. Parthasarathy, and C. Fletcher, "Conjunct: Learning inductive invariants to prove unbounded instruction safety against microarchitectural timing attacks," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024, pp. 177–177.

[37] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[38] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz, "A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020.

[39] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security'18*, 2018.

[40] J. Großschädl, E. Oswald, D. Page, and M. Tunstall, "Side-channel analysis of cryptographic software via early-terminating multiplications," in *ICISC'09*, 2009.

[41] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games—bringing access-based cache attacks on AES to practice," *2011 IEEE Symposium on Security and Privacy (S&P)*, 2011.

[42] C.-M. R. Ho, "Validation tools for complex digital designs," Ph.D. dissertation, 1997.

[43] R. Ho and M. Horowitz, "Validation coverage analysis for complex digital designs," in *Proceedings of International Conference on Computer Aided Design*, 1996.

[44] N. Homma, T. Aoki, and A. Satoh, "Electromagnetic information leakage for side-channel analysis of cryptographic modules," *2010 IEEE International Symposium on Electromagnetic Compatibility (EMC)*, 2010.

[45] Y. Hsiao, "Cva6 bug fix for pull request #1467," https://github.com/openhwgroup/cva6/commit/283177c24a1884a558f2aab26dea17d785a929db.

[46] Y. Hsiao, D. P. Mulligan, N. Nikoleris, G. Petri, and C. Trippel, "Synthesizing formal models of hardware from RTL for efficient verification of memory model implementations," in *Proceedings of the Fifty-Fourth IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 54, 2021.

[47] Intel, "Data operand independent timing instruction set architecture (isa) guidance," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html.

[48] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[49] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *CoRR*, vol. abs/1801.01203, 2018, https://arxiv.org/abs/1801.01203.

[50] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *CRYPTO'99*, 1999.

[51] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[52] D. E. Long, *Model checking, abstraction, and compositional verification*. Carnegie Mellon University, 1993.

[53] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci, "Dolma: Securing speculation with the principle of transient non-observability." in *USENIX Security Symposium*, 2021.

[54] lowRISC, "Ibex RISC-V core," 2019, https://github.com/lowRISC/ibex.

[55] D. Lustig, M. Pellauer, and M. Martonosi, "PipeCheck: Specifying and verifying microarchitectural enforcement of memory consistency models," *Proceedings of the* 47^{th} *International Symposium on Microarchitecture (MICRO)*, 2014.

[56] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee, "COATCheck: Verifying memory ordering at the hardware-OS interface," *Proceedings of the* 21^{st} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[57] Y. A. Manerkar, D. Lustig, M. Martonosi, and A. Gupta, "PipeProof: Automated memory consistency proofs for microarchitectural specifications," *Proceedings of the* 51^{st} *International Symposium on Microarchitecture (MICRO)*, 2018.

[58] Y. A. Manerkar, D. Lustig, M. Martonosi, and M. Pellauer, "RTLCheck: Verifying the memory consistency of RTL designs," *Proceedings of the* 50^{th} *International Symposium on Microarchitecture (MICRO)*, 2017.

[59] Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "CCICheck: Using μhb graphs to verify the coherence-consistency interface," *Proceedings of the* 48^{th} *International Symposium on Microarchitecture (MICRO)*, 2015.

[60] Y. A. Manerkar, C. Trippel, D. Lustig, M. Pellauer, and M. Martonosi., "Counterexamples and proof loophole for the C/C++ to POWER and Armv7 trailing-sync compiler mappings," *CoRR*, vol. abs/1611.01507, 2016, http://arxiv.org/abs/1611.01507. [Online]. Available: http://arxiv.org/abs/1611.01507

[61] S. Mangard, "A simple power-analysis (SPA) attack on implementations of the aes key expansion," *5th International Conference on Information Security and Cryptology (ICISC)*, 2003.

[62] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems*. Springer, 1995.

[63] Martonosi Research Group, "Check: Research tools and papers," 2017, http://check.cs.princeton.edu.

[64] K. L. McMillan and K. L. McMillan, *Symbolic model checking*. Springer, 1993.

[65] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, "MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations," *International Journal of Parallel Programming*, vol. 47, no. 4, 2019.

[66] G. Mohr, M. Guarnieri, and J. Reineke, "Synthesizing hardware-software leakage contracts for risc-v open-source processors," 2024.

[67] N. Mosier, H. Nemati, J. Mitchell, and C. Trippel, "Serberus: Protecting cryptographic code from spectres at compile-time," in *S&P (To Appear)*, 2024.

[68] S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen *et al.*, "Swivel: Hardening {WebAssembly} against spectre," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1433–1450.

[69] "OpenSSL: Cryptography and SSL/TLS toolkit," 2021, https://www.openssl.org/.

[70] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," *2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2006.

[71] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *USENIX Security'16*, 2016.

[72] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science*, 1977.

[73] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, "End-to-end verification of Arm® processors with ISA-formal," in *Proceedings of the* 28^{th} *International Conference on Computer Aided Verification (CAV)*, 2016.

[74] A. P. Sayakkara, N. Le-Khac, and M. Scanlon, "A survey of electromagnetic side-channel attacks and discussion on their case-progressing potential for digital forensics," *CoRR*, vol. abs/1903.07703, 2019. [Online]. Available: http://arxiv.org/abs/1903.07703

[75] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE symposium on Security and privacy*. IEEE, 2010, pp. 317–331.

[76] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "Context: A generic approach for mitigating spectre," in *Network and Distributed System Security Symposium*, 2020.

[77] A. Slowinska and H. Bos, "Pointless tainting? evaluating the practicality of pointer tainting," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 61–74.

[78] F. Solt, B. Gras, and K. Razavi, "{CellIFT}: Leveraging cells for scalable and precise dynamic information flow tracking in {RTL}," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2549–2566.

[79] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard, "Systematic classification of side-channel attacks: A case study for mobile devices," *IEEE communications surveys & tutorials*, vol. 20, no. 1, pp. 465–488, 2017.

[80] C. Trippel, D. Lustig, and M. Martonosi, "CheckMate: Automated synthesis of hardware exploits and security litmus tests," *Proceedings of the* 51^{st} *International Symposium on Microarchitecture (MICRO)*, 2018.

[81] C. Trippel, D. Lustig, and M. Martonosi, "MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols," *CoRR*, vol. abs/1802.03802, 2018. [Online]. Available: http://arxiv.org/abs/1802.03802

[82] C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, "TriCheck: Memory model verification at the trisection of software, hardware, and ISA," *Proceedings of the* 22^{nd} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[83] "Mbed TLS," 2023, https://www.trustedfirmware.org/projects/mbed-tls/.

[84] L. Uhsadel, A. Georges, and I. Verbauwhede, "Exploiting hardware performance counters," in *2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography*, 2008.

[85] K. v. Gleissenthall, R. G. Kıcı, D. Stefan, and R. Jhala, "IODINE: Verifying constant-time execution of hardware," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[86] K. v. Gleissenthall, R. G. Kıcı, D. Stefan, and R. Jhala, "Solver-aided constant-time hardware verification," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[87] M. Vassena, C. Disselkoen, K. v. Gleissenthall, S. Cauligi, R. G. Kıcı, R. Jhala, D. Tullsen, and D. Stefan, "Automatically eliminating speculative leaks from cryptographic code with blade," *Proc. ACM Program. Lang.*, 2021.

[88] J. Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher, "Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data," in *ISCA'21*.

[89] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, "Augury: Using data memory-dependent prefetchers to leak data at rest," in *IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022.

[90] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX," in *CCS '17*, 2017.

[91] Z. Wang, G. Mohr, K. von Gleissenthall, J. Reineke, and M. Guarnieri, "Specification and verification of side-channel security for open-source processors via leakage contracts," *arXiv preprint arXiv:2305.06979*, 2023.

[92] C. Wolf, J. Glaser, and J. Kepler, "Yosys: a free verilog synthesis suite," in *Proceedings of the* 21^{st} *Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[93] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," *2015 IEEE Symposium on Security and Privacy (S&P)*, 2015.

[94] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World," in *IEEE S&P*, 2019.

[95] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," *23rd USENIX Security Symposium*, 2014.

[96] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA," *IACR'16*, 2016.

[97] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data oblivious ISA extensions for side channel-resistant and high performance computing," in *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[98] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[99] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[100] F. Zaruba and L. Benini, "CVA6 RISC-V CPU," 2019, https://github.com/openhwgroup/cva6.

[101] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom, "Ultimate {SLH}: Taking speculative load hardening to the next level," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7125–7142.

[102] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "Hacl*: A verified modern cryptographic library," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1789–1806.

APPENDIX I
ARTIFACT APPENDIX

A. Abstract

This artifact uses RTL2MμPATH and SYNTHLC to conduct multi-μPATH and leakage signature synthesis, respectively, on the RISC-V CVA6 CPU [100].

B. Artifact check-list (meta-information)

- **Data set:**
  - Original CVA6 SystemVerilog design as the *design under verification* (DUV)
  - IFT-instrumented CVA6 SystemVerilog design as the DUV
  - RTL2MμPATH *and* SYNTHLC code base, implemented using Python3, SVA, and TCL.
- **Run-time environment:**
  - Cadence JasperGold: For evaluation of *SystemVerilog Assertion* (SVA) properties generated by both tools.
- **Output:**
  - Complete runs of RTL2MμPATH and SYNTHLC on CVA6 with respect to a subset of instructions from the RISC-V ISA (`ADD`, `DIV`, `LW`, `SW`, `BEQ`) to derive the following: 1) μPATHs (Fig. 2b, 2c, and 4) for these instructions, and 2) leakage signatures corresponding to this subset of the ISA.
  - Complete run of SYNTHLC seeded with μPATHs for the whole ISA to reproduce the leakage signatures as in Fig. 8.
- **Approximate total time:** The execution time primarily depends on how many JasperGold jobs the machine can run in parallel, which depends on the core number and the memory of the machine. The execution time provided below is tested on a machine with 128 cores and 700GB memory and configured to run $N = 3$ jobs in parallel. For a machine with only 48-64 cores, we recommend configuring the machine to run $N = 2$ jobs in parallel (more details in §I-F). Lastly, the execution time reported below roughly scales with a factor of $3/N$.
  - For the first two experiments (μPATHs and leakage signatures for `ADD`, `DIV`, `LW`, `SW`, `BEQ`), ~47 (~71) hours if the machine is configured to run $N = 3(2)$ JasperGold jobs in parallel.
  - For the third experiment (complete reproduction of all leakage signature in Fig. 8), the total time can take over 16 days. But this step is incremental and can produce results at a rate of about one leakage signature (a column of the Fig. 8) every $10\times3/N \sim 40\times3/N$ hours when machine runs N jobs in parallel, depending on the number of decisions the leakage signatures control. While one can stop early, to produce minimal set of results as discussed in our experiment workflow (§I-F) will take minimally 100 hours or so.

  In summary, **the total runtime is around minimally 150 hours** to see the results discussed in the instructions files (§I-F).
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.13288445

C. Description - How to access

All files including data set, code base, and instructions can be found at https://github.com/yaohsiaopid/synthlc.

D. Software Dependencies

- JasperGold for SVA evaluation
- Python3 and packages including networkx, cvc5, pandas, and matplotlib for the execution of RTL2MμPATH and SYNTHLC
- Graphviz for visualization

E. Installation

- Please follow the steps in this file to install and check software dependencies: `00-installation.md`

F. Experiment workflow

The cloned repository includes a series of instruction files at the top level. They will walk you through the evaluation of RTL2MμPATH and SYNTHLC on CVA6. Please follow these instructions in the following order: `01-setup.md`, `02-duvpl-dfg.md`, `03-rtl2mupath`, `04-synthlc.md`, `05-5instn-isa.md`, and `06-lc-table.md`.

G. Evaluation and expected results

1) `01-setup.md`: This instruction file walks through annotation preparation for CVA6 (§V-A), design augmentation (Table II), and formal environment setup for SVA property evaluation.
   We also provide instruction to configure N, the number of jobs the machine will run in parallel during SVA property evaluation steps. The estimated runtime mentioned below assumes $N = 3$.
2) `02-duvpl-dfg.md`: This instruction file walks through DUV PL derivation (§V-B1) and DFG Analysis (§V-B5), both of which are used by RTL2MμPATH to explore the execution behavior of all IUVs.
3) `{03-rtl2mupath,04-synthlc}.md`: These two instruction files compose the first experiment of this artifact. They explain how to run the *end-to-end* flow of RTL2MμPATH and SYNTHLC on a `DIV` instruction under a restricted execution assumption, which enables the entire experiment finish within seven hours (without this assumption, this may take many days). Specifically, RTL2MμPATH is configured to explore the execution of `DIV` when it is issued at the first cycle after the valid reset state (§V-B) and followed by no other valid instructions; while SYNTHLC takes in these restricted set of μPATHs, but assumes `DIV` can be preceded/followed by any of the five instructions mentioned earlier (`ADD`, `DIV`, `LW`, `SW`, `BEQ`) to synthesize a set of leakage signatures.
   This experiment illustrates the detailed flow of Fig. 6, showcasing properties automatically generated and evaluated (§V-B and §IV). It reproduces the following key results (§VII):
   - RTL2MμPATH automatically uncovers sixty-six cycle accurate μPATHs for `DIV`, a subset of all μPATHs uncovered in our full case study (§VI), but a sufficient amount to supply to and demonstrate the functionality of SYNTHLC.
   - SYNTHLC synthesizes two leakage signatures from this restricted set of `DIV` μPATHs and labels `DIV` as an intrinsic and dynamic transmitter.
   - SYNTHLC finds `DIV` is transponder for `BEQ` and `LW`/`SW` dynamic transmitters as a function of their `rs1`/`rs2` and `rs1` operands, respectively.
4) `05-5instn-isa.md`: The second experiment of this artifact steps through a reproduction of the μPATHs in Fig. 2b, 2c, and 4. To shorten runtimes, we consider a restricted RISC-V ISA that implements the following five instructions: `ADD`, `BEQ`, `LW`, `SW`, and `DIV`. This experiment can take a total of 40 hours.
5) `06-lc-table.md`: The last experiment of this artifact reproduces Fig. 8. First, we include in our dataset the full set of μPATHs synthesized by RTL2MμPATH for CVA6 (for all 72 instructions in the RV64IM ISA) in our submission-time case study (§VI), since RTL2MμPATH can take a significant amount of time to explore all 72 instructions. Second, given these μPATHs, this experiment will deploy SYNTHLC to incrementally synthesize a comprehensive set of leakage signatures (Fig. 8 columns) one at a time using SVA property generation and evaluation (§V-C1). *The flow can take over a week or more to finish depending on one's machine.* One can stop the process early to observe a partial version of Fig. 8. This experiment primarily aims to showcase the details of SYNTHLC, which is the culminating contribution in this paper.