Synthesizing Execution Contracts from RTL for Formally Verified Hardware Side-Channel Defenses

I. Introduction

In modern processors, instructions may execute *transiently*, meaning they may partially execute before being squashed and are ultimately prevented from updating architectural state. Such transient execution can arise from either hardware mispredictions or faults [1]. While transient execution is ubiquitous in modern processors, it introduces a class of *hardware side-channel attacks* [1]. In a hardware side-channel attack, a *transmitter* (an unsafe instruction in the victim program) modulates a *channel* (hardware resource) in an operand-dependent manner, and a *receiver* (attacker) indirectly observes the channel modulation to infer the operand value [2]. When the transmitter is transient, we call it a *transient execution attack*; Spectre [3] and Meltdown [4] are two classes of these attacks in which the transient transmitter is due to a hardware misprediction or fault, respectively.

To address transient execution attacks, various software [5], [6], [7], [8] and hardware [9], [10] defenses have been proposed. They rely on per-microarchitecture *security contracts*, comprised of a *leakage contract* and an *execution contract* [11]. Leakage contracts characterize a microarchitecture's transmitters, while execution contracts characterize transient instruction execution. Execution contracts may include *prediction contracts*, describing speculative control- and data-flow, and/or *exception contracts*, characterizing instructions that cause microarchitectural faults. Crucially, verifying that a defense against transient execution attacks is implemented correctly in a SystemVerilog processor design requires verifying that the design upholds the leakage and execution contracts that the defense relies on.

A substantial body of prior work has focused on verifying RTL implementations of leakage contracts [12], [13] or security contracts holistically [14], [15], [16]. However, no work has focused specifically on verifying RTL adherence to execution contracts. To fill this gap, we present SynthEC, an automated approach for synthesizing the execution contract of a processor from RTL. Using model checking [17] and symbolic information flow tracking [18], we demonstrate SynthEC by generating preliminary execution contracts for the open-source CVA6 [19] and RSD [20] RISC-V processors.

II. Background

Linear temporal logic (LTL) [21] is a formalism for expressing propositional properties over time and is commonly encoded in hardware verification as SystemVerilog Assertions (SVA) [22]. *Model checking* [17] tools either formally prove LTL properties or produce counterexamples as execution traces. SVA distinguishes two property types: *covers*, for which a model checker must witness a single satisfying trace or declare it unreachable, and *assertions*, which must hold across all traces or yield a counterexample.

Information flow tracking (IFT) [18] instruments a design with taint logic to detect how information propagates from designated sources to sinks. In symbolic IFT, a model checker formally verifies properties of the taint-instrumented design over symbolic inputs, enabling formal proofs of information flow properties over all possible executions.

III. SynthEC Overview

We propose SynthEC, a method to automatically generate formally verified execution contracts from a SystemVerilog processor design. SynthEC builds on RTL2MμPATH [12], a bottom-up formal verification approach to synthesize a set of verified *microarchitectural execution paths* (μPATHs), which are directed acyclic graphs representing an instruction's possible executions. Each node in a μPATH is a *performing location (PL)*, which is a control-flow state in which the instruction has exclusive write access to a subset of hardware state elements. SynthEC is based on the observation that if an instruction previously occupied a PL (i.e., it was present in the processor pipeline), has not yet been committed, and is no longer present in any PL then it must have been flushed, indicating it was executed transiently. Given a set of instructions under verification, SynthEC automatically constructs execution contracts, including:

- *Transient primitives:* the set of instructions that can cause transient execution, including *prediction primitives* that introduce speculation and *exception primitives* that trigger hardware faults.
- *Transient windows:* the number(s) of instructions that can be flushed due to transient execution induced by each transient primitive.
- *Control-flow targets:* the program counter values to which transient primitives can redirect control-flow.

IV. Methodology

First, SynthEC identifies instructions that introduce transient execution using symbolic IFT and model checking. Second, SynthEC characterizes each transient primitive's type, transient window, and control-flow targets.

A. Step 1: Identifying Transient Primitives

To identify transient primitives, SynthEC issues two symbolic instructions in the processor: I_{CTP}, the candidate tran-

sient primitive, followed by I_{OBS}, the instruction that may be flushed due to I_{CTP}, indicating I_{CTP} is a transient primitive.

To identify when I_{OBS} has been flushed, SynthEC adds verification logic tracking I_{OBS}'s execution via its occupancy in each PL. SynthEC recognizes I_{OBS} as occupying a PL when the PL's program counter register holds I_{OBS}'s PC, and the PL's valid signal is asserted. For example, I_{OBS} is in the decode stage when the following signal is asserted:

```
// check if i_obs is in the PL for decode stage
logic PL_decode_i = decode_stage.pc == iobs.pc &&
                    decode_stage.valid == 1;
```

An instruction is in-flight if it occupies any PL; it is flushed if it was in-flight in the previous cycle but is no longer in-flight in the current cycle, and has not been committed:

```
// check if i_obs is in any PL excluding commit
logic inflight = |{PL_1, PL_2, PL_3, ...};
logic past_inflight;
always @(posedge clk) past_inflight <= inflight;
logic flushed = past_inflight & !inflight & !PL_commit;
```

SynthEC checks if a taint originating at I_{CTP} can reach the `flushed` signal, indicating a functional path between them and that I_{CTP} is a transient primitive.

Architectural data-flow can result in false positive identification of transient primitives. For example, an ADD could be misclassified as a transient primitive if it is followed by a conditional branch with a true data-dependency on the result of the ADD. To avoid false positives, taint is prohibited from propagating through architectural dependencies, including the memory, register file, and forwarding signals along with the instruction cache.

B. Step 2: Characterizing Transient Primitives

Categorizing Primitive Type: Once an instruction is identified as a transient primitive, it is characterized as either a prediction or an exception primitive by refining the `flushed` signal. Exceptions redirect control-flow to a handler whose address is held in the trap vector base register (a CSR that is programmed at boot). SynthEC defines `flushed_exc` (`flushed_pred`), asserted when `flushed` is high and the next address equals (differs from) the trap vector base. It then reruns the model checker and checks if taint injected at I_{CTP} can still reach `flushed_exc` (`flushed_pred`), classifying I_{CTP} as an exception (prediction) primitive.

Determining Transient Window: For a transient primitive I_{CTP}, the transient window specifies the number of instructions executed transiently (i.e., are eventually squashed). To determine a transient primitive's window, SynthEC repeats the steps in §IV-A with the number of instructions between I_{CTP} and I_{OBS} fixed to a variable k. SynthEC sweeps over k: if an I_{OBS} can have a tainted `flushed` signal k instructions after I_{CTP}, then k is included in the transient window. The sweep concludes when taint no longer reaches `flushed` for I_{OBS}.

Identifying Control-Flow Targets: Control-flow targets are program counter (PC) values to which a transient primitive can redirect control-flow. Examples include PC+4 (the next sequential instruction address, assuming byte-addressed memory and 32-bit instructions), immediate values encoded in the instruction, the return address stack (RAS), and register file contents. SynthEC identifies control-flow targets using SVA cover properties. These properties are generated from templates to determine whether a transition to each candidate target is reachable. To distinguish sequential from speculative targets, SynthEC additionally tracks whether each target eventually commits. For transient primitives that do not redirect control-flow, such as data-flow prediction primitives, the only control-flow target is PC+4.

Instruction Type	Prediction	Window	Exception	Window
Branches	✓	{1}	?	-
CSR	?	-	✓	{1, 2}
EBREAK	?	-	✓	{1, 2}
ECALL	?	-	✓	{1, 2}
JALR	✓	{1}	✓	{1}

TABLE I: Synthesized preliminary execution contracts on CVA6. Instructions are classified as prediction and/or exception primitives with specific windows. ? indicates a timeout with a bound of 15 cycles.

V. Evaluation and Preliminary Results

We evaluate SynthEC on the RISC-V CVA6 [19] and RSD processors [20] to determine whether instructions in the RV64I ISA and M extension are transient primitives.

SynthEC instantiates SVA properties from templates which are evaluated using Cadence's Jasper property verifier (a suite of model checkers) [23]. For symbolic IFT, SynthEC uses Jasper SPV [24] and Jasper property verifier on designs augmented with cell-level taint tracking logic via CellIFT [18].

We evaluate SynthEC to identify prediction and exception primitives and compute their respective windows on CVA6, shown in Table I. SynthEC successfully identifies all branch instructions and JALR as prediction primitives. SynthEC also correctly identifies CSR instructions, EBREAK, and ECALL as exception primitives. All other instructions, e.g. arithmetic, logic, and memory instructions, are not classified as transient primitives (up to a bound using bounded model checking).

On RSD, SynthEC identifies transient primitives and their respective control-flow targets. It finds that all instructions can be predicted as a branch in the processor frontend, such that they can redirect control-flow to the predicted address from BTB. After the decode stage, control-flow divergences from non-branch instructions such as ADD and DIV are resolved, so the next valid instruction is always PC+4, meaning they cannot be phantom branches [25]. Real branches only redirect control-flow to the output of the BTB or the branch label.

VI. Related Work

Leakage contract verification: Prior work has proposed both formal and fuzzing approaches to determine whether RTL upholds proposed leakage contracts [12], [13], [26], [27], [28].

Execution contract verification: One prior work verifies the adherence of CPU to leakage contracts with respect to exceptions [29]. No prior work to our knowledge verifies adherence to prediction contracts.

Combined contract verification: Other prior work encompasses both execution and leakage contract verification via speculative non-interference properties [14], [15], [16].

References

[1] 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.

[2] 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)*. IEEE, 2018, pp. 974–987.

[3] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, and et al., "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[4] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and et al., "Meltdown: Reading kernel memory from user space," *Communications of the ACM*, vol. 63, no. 6, pp. 46–56, 2020.

[5] N. Mosier, H. Nemati, J. C. Mitchell, and C. Trippel, "Serberus: Protecting cryptographic code from spectres at compile-time," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 4200–4219.

[6] 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.

[7] 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," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–30, 2021.

[8] B. A. Shivakumar, J. Barnes, G. Barthe, S. Cauligi, C. Chuengsatiansup, D. Genkin, S. O'Connell, P. Schwabe, R. Q. Sim, and Y. Yarom, "Spectre declassified: Reading from the right place at the wrong time," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 1753–1770.

[9] 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)*. IEEE, 2019, pp. 151–164.

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

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

[12] Y. Hsiao, N. Nikoleris, A. Khyzha, D. P. Mulligan, G. Petri, C. W. Fletcher, and C. Trippel, "Rtl2mμpath: Multi-μpath synthesis with applications to hardware security verification," in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2024, pp. 507–524.

[13] S. Dinesh, M. Parthasarathy, and C. W. Fletcher, "Conjunct: Learning inductive invariants to prove unbounded instruction safety against microarchitectural timing attacks," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 3735–3753.

[14] Q. Tan, Y. Yang, T. Bourgeat, S. Malik, and M. Yan, "Rtl verification for secure speculation using contract shadow logic," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 970–986. [Online]. Available: https://doi.org/10.1145/3669940.3707243

[15] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz, "Processor hardware security vulnerabilities and their detection by unique program execution checking," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 994–999.

[16] 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," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2128–2142.

[17] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[18] 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.

[19] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, Nov 2019.

[20] S. Mashimo, A. Fujita, R. Matsuo, S. Akaki, A. Fukuda, T. Koizumi, J. Kadomoto, H. Irie, M. Goshima, and K. Inoue, "An open source fpga-optimized out-of-order risc-v soft processor," in *2019 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2019, pp. 63–71.

[21] A. Pnueli, "The temporal logic of programs," in *18th annual symposium on foundations of computer science (sfcs 1977)*. ieee, 1977, pp. 46–57.

[22] D. A. S. Committee *et al.*, "Ieee standard for systemverilog unified hardware design, specification, and verification language standard ieee 1800," *http://www. edastds. org/sv/*, 2005.

[23] Cadence Design Systems, Inc., "Cadence JasperGold Formal Verification Platform," https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html, 2026, accessed: 2026-04-21.

[24] ——, "Jasper SPV App," https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-verification-platform/security-path-verification-app.html, 2026, accessed: 2026-04-21.

[25] J. Wikner, D. Trujillo, and K. Razavi, "Phantom: Exploiting decoder-detectable mispredictions," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 49–61.

[26] K. Ceesay-Seitz, F. Solt, and K. Razavi, "μcfi: Formal verification of microarchitectural control-flow integrity," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 213–227.

[27] T. Kovats, F. Solt, K. Ceesay-Seitz, and K. Razavi, "Milesan: Detecting exploitable microarchitectural leakage via differential hardware-software taint tracking," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 2579–2593. [Online]. Available: https://doi.org/10.1145/3719027.3765066

[28] S. Dinesh, Y. Zhu, and C. W. Fletcher, "H-houdini: Scalable invariant learning," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 603–618. [Online]. Available: https://doi.org/10.1145/3669940.3707263

[29] J. Hofmann, E. Vannacci, C. Fournet, B. Köpf, and O. Oleksenko, "Speculation at fault: Modeling and testing microarchitectural leakage of {CPU} exceptions," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7143–7160.