

UC Berkeley

UC Berkeley Electronic Theses and Dissertations

Title

Synthesis-Enabled Reasoning: Scalable Automated Formal Verification for Hardware-Software Security

Permalink

<https://escholarship.org/uc/item/2qd921ph>

ISBN

9798189279942

Author

Godbole, Adwait

Publication Date

2026-05-01

Peer reviewed|Thesis/dissertation

Synthesis-Enabled Reasoning: Scalable Automated Formal Verification for
Hardware-Software Security

by

Adwait Godbole

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sanjit A. Seshia, Chair
Associate Professor Alvin Cheung
Associate Professor Chris Fletcher
Assistant Professor Yatin A. Manerkar

Spring 2026

Synthesis-Enabled Reasoning: Scalable Automated Formal Verification for
Hardware-Software Security

Copyright 2026
by
Adwait Godbole

Abstract

Synthesis-Enabled Reasoning: Scalable Automated Formal Verification for
Hardware-Software Security

by

Adwait Godbole

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Sanjit A. Seshia, Chair

Formal reasoning can prove that hardware-software systems satisfy critical correctness and security properties, but applying it requires bridging three persistent gaps between the systems we build and the tools available to reason about them: the *implementation-to-model gap*, where real systems are too complex for direct reasoning; the *cross-layer gap*, where properties span hardware and software layers that tools treat in isolation; and the *representation-to-tool gap*, where specifications natural to formulate are often incompatible with the verification engines best suited to discharge them. These gaps are bridged by hand today, by constructing the artifacts reasoning tools consume—abstractions, operational models, invariants. This manual effort forms the bottleneck. This thesis introduces *synthesis-enabled reasoning* (SER), a methodology that closes these gaps by automating the construction of reasoning artifacts through formal synthesis. By eliminating manual effort, SER improves scalability (through abstraction), soundness (by eliminating modelling errors), and applicability (through verification-friendly specifications). SER reduces verification of a complex system to a set of smaller synthesis problems over a structured space of artifacts, maintaining tractability. In the future, SER is poised to leverage the rise of neural search techniques to further advance scalability and applicability of reasoning, while preserving soundness by relying on a trusted base of verification tools.

We demonstrate SER for security and memory consistency at the hardware-software interface. In the security setting, microarchitectural optimizations such as speculative execution and shared caches create side channels exploited by attacks like Spectre and Meltdown; in the memory consistency setting, relaxed ordering in multi-core systems governs the correctness of multi-threaded software. To close the implementation-to-model gap, micro-update models are abstractions of hardware that preserve only security-relevant behavior, automatically synthesized from the source design and yielding 2–21× verification speedup on RISC-V processors. For the cross-layer gap, SemPat automatically generates the patterns

analysts use to scan software for vulnerabilities, starting from a high-level security policy, and discovers new Spectre-related patterns that prior manual catalogs missed. To bridge the representation-to-tool gap, Ax2Op converts declarative memory-consistency specifications into a form that off-the-shelf unbounded model checkers can verify, while PipeSynth infers formal memory-ordering specifications from concrete examples—before any hardware code is written. Finally, PyCaliper provides a unified Python framework that lets engineers express specifications once and target multiple verification and synthesis tools; it has been adopted by Intel Labs for secure processor verification. Together, these contributions show that synthesis can make formal reasoning tractable where direct approaches fail, with preliminary results extending the methodology to AI-assisted software development.

To Aai and Baba

Contents

Contents	ii
List of Figures	v
List of Tables	viii
1 Introduction	1
1.1 Motivating Example: Secure AES on RISC-V	2
1.2 The Formal Methods Opportunity	4
1.3 Thesis Contributions and Roadmap	5
2 Automated Verification and Synthesis	7
2.1 Formal Models	7
2.2 Formal Specifications	8
2.3 Verification Techniques	10
2.4 Abstraction	12
2.5 Automated Synthesis	12
3 Challenge: Reasoning at the Hardware-Software Interface	17
3.1 The Formal Setting	17
3.2 Hardware Security	18
3.3 Memory Consistency	21
3.4 Verification Challenges	24
3.5 Unifying Theme: Synthesis-Enabled Reasoning	26
I The Implementation-to-Model Gap	27
4 Micro-Update Models for RTL Security Reasoning	29
4.1 Balancing Expressivity and Verification Cost	29
4.2 The Micro-Update Model	32
4.3 Synthesizing MUMs from RTL	37
4.4 Results and Insights	42

4.5	Discussion	45
4.6	Related Work	46
5	Modelling and Verifying Isolation-Based Mitigations	48
5.1	Abstract Resource Model	49
5.2	Attacker Model and Security Property	50
5.3	Conditional Equality Invariants	51
5.4	Verification Conditions	51
5.5	Results	52
II	The Cross-Layer Gap	54
6	Generating Syntactic Security Patterns from Semantic Properties	56
6.1	The Semantic-Syntactic Divide	56
6.2	Attack Patterns and k -Completeness	59
6.3	The SemPat Algorithm	61
6.4	Results and Insights	65
6.5	Related Work	71
6.6	Discussion	72
III	The Representation-to-Tool Gap	73
7	Ax2Op: Converting Axiomatic to Operational Memory Models	74
7.1	The Paradigm Mismatch	74
7.2	Operational Model of Computation	75
7.3	Enabling Synthesis by Bounding Reorderings	76
7.4	Converting to Operational Models Using Axiom Automata	80
7.5	Case Studies: Ax2Op	82
7.6	Related Work	84
7.7	Discussion	85
8	PipeSynth: Synthesizing Axioms from Litmus Tests	86
8.1	The Pre-RTL Specification Problem	86
8.2	Axiom Templates	87
8.3	Challenges in μ spec Synthesis	89
8.4	Conjunct-Based SyGuS Encoding	90
8.5	Evaluation	91
8.6	Discussion	94

IV	On Infrastructural Challenges	96
9	Infrastructure for SER: PyCaliper	97
9.1	The Infrastructure Problem	98
9.2	Language Design	98
9.3	Multi-Backend Architecture	101
9.4	Illustrative Example: FIFO No-Stale-Data Property	103
9.5	Case Study: TAGE Branch Target Injection	105
9.6	Results and Impact	108
9.7	Discussion	109
V	Conclusion and Outlook	110
10	Conclusion and Outlook	111
10.1	Summary of Contributions	111
10.2	Impact	112
10.3	Limitations	113
10.4	Outlook: Synthesis-Enabled Reasoning in the Age of AI	114
10.5	Open Problems and Future Directions	118
A	Proofs for Ax2Op Operationalization	120
A.1	Axiom-Automata Lemma (Lemma 7.3)	120
A.2	Checking Extensibility (Lemma 7.2)	120
A.3	Bounding In-Progress Instructions (Lemma 7.4)	121
A.4	Bounding Active Instructions (Lemma 7.5)	122
A.5	Active Automata and the Feasibility Theorem (Theorem 7.2)	123
B	Hardware Mapping for DAWG and COLORIS	125
B.1	DAWG Cache Partitioning	125
B.2	COLORIS Page Coloring	126
	Bibliography	127

List of Figures

1.1	Thesis overview: three gaps at the hardware-software interface, the SER tools that close each gap, and the infrastructure layers that unify them.	5
2.1	The synthesizer-verifier loop: the synthesizer generates candidates from a synthesis space $\mathcal{S}$, the verifier checks whether each candidate satisfies specification φ , and counterexamples refine the search. The loop terminates when a valid artifact is found or the space is exhausted.	14
4.1	Pipeline timing for two store-word implementations and a dependent load in a 5-stage RISC-V pipeline. (a) sw_{Exe} : the store resolves at the execute stage (green), flushing the load buffer early. (b) sw_{Mem} : the store resolves at the memory stage (green), delaying the flush and allowing a dependent load (c) to progress further through the pipeline. The later flush point in (b) gives speculative instructions an additional pipeline stage in which to interact with the cache, widening the side-channel window.	30
4.2	Schematic for the Sodor 5-stage processor. Our modification, the load buffer (LB), is shown in grey.	30
4.3	Different microarchitectural behaviours (with a 1-cycle memory latency) for two store implementations. With sw_{Exe} , the load buffer retains its entry after the load-store sequence; with sw_{Mem} , the store flushes the buffer after the refill. . . .	31
4.4	Overview (A): Processor schematic for the Sodor 5-stage core with load buffer and bypassing paths (grayed).	33
4.5	Overview (B): Micro-update code blocks for decode, ALU, and load buffer operations.	33
4.6	Overview (C): Guard table mapping micro-updates to Boolean trigger conditions over pipeline state.	33
4.7	Overview (D): Example M-sets for three instructions: addi without dependency, addi with ALU-output bypass, and sw without dependency.	34
4.8	MUM synthesis pipeline	39
4.9	A sample execution trace from simulation, showing signal values for the load buffer and memory port across four cycles. At the third transition (counter 2→3), the load buffer is refilled (lb_data 10 → 30, lb_valid 0 → 1) as the lb instruction reaches the write-back stage.	41

4.10	Subtle program changes can expose vulnerabilities. (A) The Spectre BCB gadget from Kocher et al. [68]. (B, C) Compiled assembly from the original example and after changing <code>&=</code> to <code>=</code> . The accompanying waveforms show how (B) leaves the load buffer clean under <code>sw_{Exe}</code> while (C) does not.	44
5.1	Prime+Probe attack on a partitioned cache. The cache has four lines; l_0, l_1 are allocated to the victim, l_2 to the attacker, l_3 is unallocated. The attacker primes l_2 , then the victim accesses one of l_0 (case A), l_1 (case B), or l_2 (case C). On a partitioned cache, case C is blocked, so the attacker's probe of l_2 yields a hit in both A and B, leaking no information. However, the replacement/allocation policy state (cloud) may still differ between A and B, potentially leaking information through subsequent attacker accesses.	49
6.1	Motivating examples for SemPat	57
6.2	Patterns A–D matched against compiled executions of <code>victimA–victimD</code> . Each pattern embeds in the corresponding program's instruction sequence through op-code and dependency matching.	58
6.3	SemPat synthesis pipeline	60
6.4	Pattern E: discovered by SemPat for the store-to-load forwarding platform. The pattern captures a store-load-load sequence where forwarding from the STL buffer creates a timing side channel.	68
6.5	Verification times for Branch and STL speculation litmus tests using hyperproperty-based (dark) and pattern-based (light) detection. Pattern-based analysis completes in ~ 5 seconds for all tests, while hyperproperty-based analysis varies from minutes to timeouts (15-minute limit).	68
6.6	Scalability of pattern generation. (a) Varying set index width of the LSQ cache (same associativity). (b) Varying way index width (same sets). (c) Varying reuse buffer size. (d) Generation time and pattern count for the synthetic platform $M_{\text{synth}}(k)$ with varying microarchitectural depth (k) and grammar depth (d). . .	70
6.7	(F) Pattern using the default grammar with address-level predicates. (G) More precise pattern after adding the diffindex predicate, which compares set indices rather than full addresses.	71
6.8	Example program <code>test_K</code> : the store's full address differs from the first load's, but their set indices may match depending on <code>K</code> and <code>SET.W</code> , causing pattern F to produce a false positive when the store would in fact invalidate the LSQ entry. .	71
7.1	A non-refinable axiom. (a) A valid execution graph under the axiom $\forall i_1, i_2. \neg \text{hb}(i_1.S, i_2.S) \wedge \neg \text{hb}(i_2.S, i_1.S)$. (b, c) Both linearizations of (a) are invalid, since each introduces an ordering that the axiom forbids.	78
7.2	Completed prefix (CM), in-progress (IP), and not-fetched postfix (NF) of instructions during execution. Instructions flow left to right through the pipeline; the three sets partition the instruction stream at any point during a trace.	81

7.3	Experimental setup. The operational model $\mathcal{M}$ is composed with the hardware design via event-signaling ports (e.g., PC wires). Instruction inputs are symbolically constrained, enabling verification over all programs within the bound. . . .	82
8.1	PipeSynth block diagram. Rectangles are required inputs; ellipses are optional. The rounded rectangle represents PipeSynth’s output upon successful synthesis.	87
8.2	(a) A monolithic synthesis encoding contrasted with (b) the structured conjunct-based encoding for a path axiom. Red boxes represent the decisions made by the solver from groups (columns) of black boxes. The conjunct-based encoding prevents the solver from examining redundant and nonsensical candidates (e.g., $\neg a \vee b$ and $a \Rightarrow b$, or $a \wedge b$ and $b \wedge a$), dramatically reducing the search space. . .	90
8.3	Single-axiom synthesis: number of axioms successfully synthesized over time, by microarchitecture. Each query is run independently.	93
8.4	PipeSynth + Ax2Op as an end-to-end SER pipeline. Litmus tests and execution traces feed PipeSynth, which synthesizes μ spec axioms. Ax2Op converts these to operational automata for IC3/PDR verification. Counterexamples from verification feed back to refine the axioms, instantiating the synthesizer-verifier pattern (Chapter 2) at the specification level.	95
9.1	PyCaliper infrastructure: specifications (with or without synthesis holes) compile to multiple verification backends. Synthesis engines fill holes via CEGIS or incremental inductive synthesis, using verification as a subroutine. Utilities provide trace generation and design analysis as oracle calls.	101
10.1	Syzygy architecture. A. SpecMiner extracts behavioral contracts (allocation bounds, aliasing, nullability) from C program executions. B. CodeGenerator uses an LLM to produce Rust translations guided by the contracts. C. ArgTranslator converts C-to-Rust type signatures and I/O. D. EqTester validates behavioral equivalence; failures feed back to the code generator for iterative refinement.	116
A.1	Completed prefix (pCM), in-progress (IP), and not-fetched postfix (pNF) during execution. The commit-point $\text{cp}(j, \text{c})$ and fetch-point $\text{fp}(j, \text{c})$ delimit the in-progress range for each core. The second row for each core shows completed (CM , green) and not-fetched (NF , red) instructions; pCM and pNF are their maximal prefix/postfix-closed subsets.	122

List of Tables

2.1	Synthesis instantiations across the thesis.	15
4.1	MUM synthesis for Sodor 5-stage processor.	43
4.2	MUM synthesis for CVA6 processor components.	43
4.3	Information-flow analysis: MUM vs. RTL verification time on Sodor.	44
4.4	LSU timing channel analysis on CVA6: MUM vs. RTL.	45
5.1	DAWG cache partitioning verification results.	53
6.1	Pattern-to-hyperproperty mapping. A single SNI hyperproperty captures both variants in each pair, but pattern-based detection requires separate patterns for each.	57
6.2	Attack-pattern (AP) approaches: execution models and pattern variants.	58
6.3	Semantic hyperproperty (SH) approaches and their non-interference variants.	59
6.4	Default predicate grammar for SemPat pattern constraints.	63
6.5	Pattern generation times by platform and word width.	67
6.6	Precision comparison on <code>test_K</code> with set index width <code>SET_W</code>	71
7.1	Ax2Op: Multi-Vscale and Tomasulo verification results.	83
7.2	ReadValues axiom verification on Multi-Vscale (2 cores, symbolic load/store).	84
8.1	Single-axiom synthesis breakdown by template and microarchitecture (successes/total, excluding fence axioms). Comb. = combination of templates.	92
8.2	PipeSynth single-axiom synthesis results (excluding fence axioms; 3 fence axioms across FiveStage, OoO-SLR, and Peekaboo are trivially synthesized as empty axioms, yielding 42/46 overall).	92
8.3	Multi-axiom synthesis time (minutes:seconds) as axioms are progressively removed. Entries in <i>italics</i> are trivial (all conjuncts <code>True</code>). T/O = timeout at 2 hours.	93
9.1	TAGE BTI verification: SVA (Jasper) vs. BTOR2 (Boolector), 2-induction. TT = total time, ST = solving time, PT = parsing time. The Boolector backend's total time is dominated by parsing the flattened BTOR2 representation (PT column), not SAT solving (ST column).	106

9.2 Invariant synthesis for DAWG: random vs. neural strategy (20-run average). . .	107
--	-----

Acknowledgments

First and foremost, I would like to thank my advisor, Sanjit A. Seshia. Sanjit gave me complete intellectual freedom while relentlessly pushing me toward real-world impact. He trusted me to find my own path, and always had my back, shielding me from all non-technical challenges. Thanks, Sanjit.

I owe a great deal to Yatin A. Manerkar, whose patient, hands-on mentoring during my early years shaped how I approach research and writing.

I would like to thank my committee members, Chris Fletcher and Alvin Cheung, for their valuable feedback and insights that improved the quality of this dissertation. Beyond serving on my committee, I am grateful for being able to closely collaborate with their groups, and for the many resulting conversations that have shaped my thinking over the years.

My research journey began before Berkeley. I am grateful to S. Akshay and Krishna S. at IIT Bombay for my first experiences with research. I thank Parosh Abdulla and Faouzi Atig at Uppsala University, and Blaise Genest at INRIA for the many fruitful collaborations that set the trajectory for much of my later work.

Through my time here, I was lucky to mentor Chase Norman, Leiqi Ye, Alejandro Sanchez Ocegueda, Viansa Schmulbach, James Lim, and Manahil Syeda. They each taught me more about mentoring than I could have taught them about research.

I was surrounded by an extraordinary group of labmates who I learned a great deal from: Sahil Bhatia, Kai-Chun Chang, Kevin Cheang, Pei-Wei Chen, Karim Elmaaroufi, Aniruddha Joshi, Niklas Lauffer, Shaokai Lin, Federico Mora, Ameesh Shah¹, Victoria Tuck, Justin Wong, Beyazit Yalcinkaya, as well as postdocs (and former postdocs) in the group, Elizabeth Polgreen, Hazem Torfah, Hussain Sibai, Hanna Krasowski, and Edward Kim.

Interdisciplinary research is a defining theme of this dissertation, and I was fortunate to collaborate with stellar researchers across different groups at Berkeley: Kevin Laeuffer and Tianrui Wei, who always had great instincts and advice about architecture and hardware verification; Naman Jain and Manish Reddy who I worked with on Syzygy²; Alex, Yingchen Wang, Flavien Solt on TEEs; and Rutvik Chaudhary, Sushant Dinesh on security. Of course, my time here has also been shaped by discussions and advice from other faculty in adjacent groups: Koushik Sen, Sarah Chasins, Max Willsey, Raluca Ada Popa and others. The breadth of these interactions is a testament to Berkeley’s research culture and I am fortunate to have had a very rich research journey here³.

Outside of work, I shared many good moments with Jiwon, Tianrui, Layla and the rest of the swarmcycle gang, Charles, Justin, Parker, and many many others, as well as Jathushan, my flatmate for much of my stay here.

Several parts of this dissertation were shaped by close collaborations with industry, in particular Intel. At Intel, I am grateful to John Matthews, Jin Yang, Carlos Rozas, and

¹I may not have learned much from Ameesh, Federico, and Pei-Wei, but I at least could teach them how to be better at playing Pocket Tanks (though I still remain undefeated).

²a.k.a Crusty, a.k.a. DynaTranslate.

³Including a robotics paper, by virtue of which I can hazard calling myself a roboticist.

John O’Leary, whose advice left a lasting imprint. I am thankful to Brian Huffman and Fangfei Liu for close collaboration on PyCaliper and their mentorship during my time at Intel. I also thank Jeremy Casas and Zhenkun Wang for a formative internship earlier in my Ph.D. At AWS, I thank Shilpi Goel and Jim Grundy for their mentorship and support during an exciting internship in Austin. I also had the opportunity to collaborate with some amazing researchers at other universities, in particular with Somesh Jha and Tej Chajed at UW–Madison, and Varun Chandrasekaran at UIUC.

My school friends — Nachiket, Tanay, Tanmay, and Viraj — and college batchmates Nilay and Nitish have been a source of enduring friendship over the years.

I am grateful to Radhika for making the hardest stretch of this journey the brightest. Finally, none of this would have been possible without Aai, Baba, and Radhika, whose unwavering love and belief carried me through it all.

Chapter 1

Introduction

The correctness of modern computing systems rests on a tower of abstractions. Software assumes sequential instruction semantics; operating systems assume isolated address spaces; compilers assume faithful execution of the ISA. Each abstraction suppresses detail that would otherwise overwhelm both human reasoning and automated analysis, and the gaps between microarchitectural and software layers rarely receive systematic formal scrutiny. The Spectre [68] and Meltdown [76] vulnerabilities exposed the cost of this neglect: microarchitectural optimizations intended to be invisible to software—speculative execution, branch prediction, shared cache hierarchies—create side channels that violate the very isolation properties higher layers depend on. Exploiting these side channels, an attacker can extract cryptographic keys from a process that the software stack considers isolated. These vulnerabilities are not implementation accidents; they are *abstraction failures*, symptoms of a systematic gap between the abstractions we reason with and the systems we build.

This thesis develops techniques for *bridging* such gaps. The case for combining abstraction with synthesis has been made before: Seshia argued for *reducing verification to synthesis* as a systematic methodology in DAC 2012 [104] and Proceedings of the IEEE 2015 [103], and the UCLID5 verification and synthesis platform [105, 100] embodies this principle by integrating modeling, verification, and synthesis in a single environment. What this thesis adds is empirical evidence: we *demonstrate* that synthesis-enabled reasoning is indispensable for verifying complex hardware-software systems, with security properties as the most demanding application. The unifying methodology, which we call *synthesis-enabled reasoning* (SER), is defined as follows:

Synthesis-Enabled Reasoning is a methodology that combines formal methods, programming-language techniques, and AI-based methods to *automatically synthesize verification artifacts*—such as abstractions, specifications, operational models, or invariants—that bridge the gap between the system and the available formal tools. The synthesis is tractable because domain-specific structure constrains the search space. The result is sound because the synthesized artifact is formally verified (via bounded or unbounded model checking, depending on the

application).

SER is a methodology: identify the gap, characterize the artifact space using domain structure, synthesize candidates, and verify them. This pattern succeeds because hardware-software systems exhibit rich domain structure—module boundaries, instruction templates, resource allocation policies—that dramatically constrains the space of candidate artifacts. This turns an intractable search into a feasible one. Each of the following chapters instantiates this pattern for a different gap at the hardware-software interface.

1.1 Motivating Example: Secure AES on RISC-V

To illustrate the challenges that motivate SER, consider a concrete scenario: *verifying that a RISC-V processor securely and correctly executes a T-table AES-128 encryption kernel*. T-table AES is a standard software implementation of the Advanced Encryption Standard in which each round computes output bytes by indexing into four precomputed lookup tables:

```
state[i] = Te0[s[0]] ^ Te1[s[5]] ^ Te2[s[10]] ^ Te3[s[15]] ^ round_key[i]
```

Each table access `Te0[s[j]]` is a memory operation whose address depends on intermediate cipher state `s[j]`, which in turn depends on the secret key. On a RISC-V processor with speculative execution, this single line of code creates three distinct verification challenges—each an instance of a broader, well-studied problem in hardware-software verification. We illustrate these on the Sodor 5-stage pedagogical core [20] (for security analysis)—an in-order pipeline that we augment with a hypothetical load buffer to study speculative-execution effects—and the CVA6 application-class core [136] (for scalability evaluation).

1.1.1 The Implementation-to-Model Gap

Modern processors speculatively execute instructions past unresolved branches, squashing them if the prediction proves wrong. The squash restores *architectural* state but leaves *microarchitectural* traces—modified cache lines, updated branch predictor entries—that persist and create timing side channels [68]. This is the *implementation-to-model gap*: the full RTL of a processor is too complex for direct formal analysis, yet security-critical behaviors—such as which addresses enter the cache during speculation—are invisible at higher levels of abstraction.

For AES, a speculative T-table lookup `Te0[s[0]]` may populate the load buffer or cache with a key-dependent address that persists even after squash, creating a side-channel witness. More broadly, the implementation-to-model gap arises whenever a hardware design’s security-relevant behavior is buried in microarchitectural detail that exceeds the capacity of existing verification tools [21, 57].

A related instance involves branch predictors such as the TAGE predictor [108], whose tagged history tables can be poisoned across privilege boundaries. Verifying that predictor

partitioning prevents such cross-domain influence requires invariants over microarchitectural state that the ISA abstraction deliberately hides.

1.1.2 The Layer-to-Layer Gap

Ensuring that AES key material does not leak via cache timing is not a property of the hardware alone, nor of the software alone—it concerns the interaction between T-table access patterns in the software and speculative-execution behavior in the hardware. Like many relational security properties, this cross-layer requirement is naturally formalized as a *hyperproperty* [27], relating multiple executions to express requirements like “changing the secret key does not affect publicly observable timing.” Hyperproperties are not specific to the cross-layer setting—they capture a broad class of relational properties—but the cross-layer setting makes their verification especially difficult: each layer has its own abstractions and tools, and no widely-used single formalism naturally captures the property end-to-end.

This is the *cross-layer gap*: security properties that span abstraction boundaries cannot be expressed or verified within any single layer’s formalism. The gap is well known in the information-flow security literature [52], but becomes especially acute at the hardware-software interface, where semantic intent (“no secret-dependent flows”) must be translated into syntactic artifacts (automata, monitor circuits) that verification tools can consume. Cache-timing attacks on AES and hardware resource-partitioning defenses such as DAWG [67] are prominent instances of this broader challenge.

1.1.3 The Representation-to-Tool Gap

A separate class of challenges arises in multi-threaded programs. Correct concurrent execution requires store-ordering guarantees specified in a memory consistency model (e.g., RISC-V RVWMO [126]). Such ordering properties are naturally expressed *axiomatically* through happens-before relations in formalisms such as μ spec [78, 80], but the verification engines best suited to unbounded proofs—IC3/PDR [12]—require *operational* (state-machine) input.

This is the *representation-to-tool gap*: a specification written in one paradigm may be incompatible with the best available verification tools. The gap is not specific to memory models; it arises whenever the natural specification language for a domain differs from the input format of existing solvers. Pre-RTL specification is a closely related challenge: writing formal axioms demands joint expertise in formal methods and microarchitecture, yet architects more naturally produce examples—litmus tests exhibiting forbidden behaviors, execution traces from simulators such as gem5 [10]—that could in principle serve as specification sources.

The AES cache side-channel scenario illustrates the security challenges that motivate Parts I and II of the thesis. The memory consistency challenges motivate Part III. PyCaliper (Chapter 9) provides the infrastructure to express all three classes of properties in a single Python framework.

1.2 The Formal Methods Opportunity

Having identified three concrete gaps, we now consider why formal methods—rather than testing or simulation—are the right foundation for bridging them. *Formal methods* are mathematical techniques for specifying, modeling, verifying, and synthesizing computing systems; *formal verification*, the focus of much of this thesis, is the subfield that aims to *prove* that a system satisfies its specification rather than merely test for bugs. Grounded in logic and automated reasoning—SAT and SMT solving, model checking, theorem proving—formal methods offer several capabilities that are central to this thesis. Most fundamentally, formal verification provides *correctness guarantees* over all possible inputs and states, not just the subset that a test suite happens to exercise. For security-critical properties such as isolation, non-interference, and cryptographic correctness, this exhaustiveness is especially important: attackers exploit precisely those rare corner cases that testing is unlikely to cover, and formal methods can, relative to a given specification, eliminate entire vulnerability classes at once. Beyond verification, formal techniques enable *design-time bug discovery*—PipeSynth (Chapter 8) synthesizes μ spec microarchitectural specifications from litmus tests and execution traces, after which those specifications can be verified against ISA-level memory consistency models using existing μ spec-based tools (e.g., the COATCheck [80] iteration of the PipeCheck methodology) to expose specification errors before RTL exists—and, perhaps counterintuitively, can *accelerate development*: automated synthesis generates specifications from examples, reducing manual effort, while domain-specific languages such as PyCaliper (Chapter 9) lower the expertise barrier.

1.2.1 Verification and Synthesis

Formal techniques range from fully manual to fully automated. *Interactive theorem proving* offers maximum expressiveness but requires the user to guide every proof step. *Property-directed verification* automates the proof search once the user supplies a property and an appropriate abstraction. *Specification synthesis* goes further: the user provides examples (litmus tests, execution traces), and the tool generates the formal specification itself. *Invariant synthesis* is fully automated given a grammar or template, producing the inductive invariants needed to close a proof. At the most automated end, *neural and LLM-assisted synthesis* uses machine learning to suggest specifications or guide the search.

The idea of *reducing verification to synthesis* — automating invariant discovery, abstraction construction, and environment modeling through example-guided search — was articulated as a systematic methodology by Seshia [103]. That same work advocated solving the underlying synthesis problem by integrating deductive reasoning, inductive learning from examples, and data-driven AI techniques—a synthesis-by-integration view that this thesis inherits. The contributions of this thesis instantiate that approach at the hardware-software interface, spanning several points along this range from property-directed verification with expert-provided abstractions to fully automated synthesis of specifications and invariants from examples.

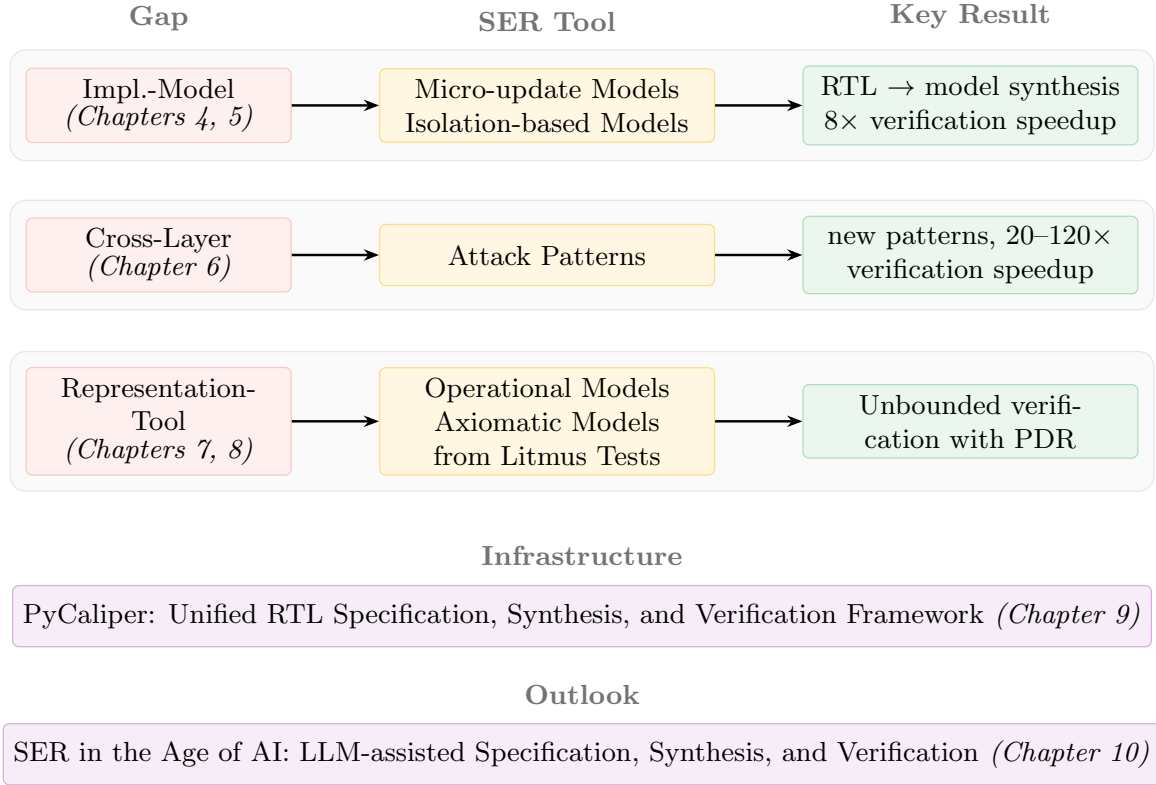


Figure 1.1: Thesis overview: three gaps at the hardware-software interface, the SER tools that close each gap, and the infrastructure layers that unify them.

1.3 Thesis Contributions and Roadmap

This thesis identifies three gaps that impede formal reasoning about hardware-software systems and shows that each can be closed through synthesis-enabled reasoning. Figure 1.1 provides an overview; the remainder of the thesis is organized as follows.

Chapters 2 and 3 lay the groundwork. Chapter 2 introduces the formal machinery—transition systems, temporal logic, verification algorithms (BMC, IC3/PDR), abstraction, and synthesis (CEGIS, SyGuS)—that the technical chapters build upon. Chapter 3 formalizes the hardware-software interface, develops running examples around the AES scenario, and identifies the verification challenges that motivate the rest of the thesis.

The technical chapters are organized into three parts, each closing one gap through synthesis. For the *implementation-to-model gap* (Part I): Chapter 4 presents micro-update models (MUMs) [48], which synthesize compact abstractions from RTL that preserve security-relevant behavior, yielding a 2–21× verification speedup (geometric mean $\sim 8\times$) on the

Sodor [20] and CVA6 [136] cores. Chapter 5 presents an abstract resource partitioning framework [49] with conditional equality invariants, enabling unbounded verification of hardware isolation mechanisms such as DAWG cache partitioning. For the *cross-layer gap* (Part II): Chapter 6 presents SemPat [47], which synthesizes k -complete attack-detection patterns from hyperproperty specifications, discovering two new Spectre-related patterns and enabling 20–120 $\times$ faster verification through pattern matching. For the *representation-to-tool gap* (Part III): Chapter 7 presents Ax2Op [46], which converts axiomatic memory-consistency models to operational automata for IC3/PDR. Chapter 8 presents PipeSynth [94], which synthesizes μ spec axioms pre-RTL from litmus tests and execution traces.

The remaining chapters address infrastructure and outlook. PyCaliper [50] (Chapter 9, Part IV) provides a Python-embedded DSL that unifies verification (SVA, BTOR2) and synthesis (CEGIS, LLM-based) backends under a single language. Chapter 10 concludes with an outlook on the intersection of SER and large language models [109], followed by a discussion of impact, limitations, and future directions.

Taken together, these contributions demonstrate that synthesis-enabled reasoning can make formal verification more automated and accessible for hardware-software security and consistency analysis.

Joint work and co-author contributions. The technical chapters of this thesis are based on jointly authored papers. The following is a high-level summary of collaborators on each contribution. *Micro-update models* (Chapter 4, [48]) is joint with Kevin Cheang, Yatin A. Manerkar, and Sanjit A. Seshia. *Conditional equality invariants for resource partitioning* (Chapter 5, [49]) is joint with Leiqi Ye (then a mentee), Yatin A. Manerkar, and Sanjit A. Seshia. *SemPat* (Chapter 6, [47]) is joint with Yatin A. Manerkar and Sanjit A. Seshia. *Ax2Op* (Chapter 7, [46]) is joint with Yatin A. Manerkar and Sanjit A. Seshia. *PipeSynth* (Chapter 8, [94]) is joint with Chase Norman and Yatin A. Manerkar. Yatin proposed and mentored the project; Chase was the first author on the paper and developed the implementation, evaluation, and most of the writing. The candidate contributed to the formal framework, axiom template design, and writing. *PyCaliper* (Chapter 9, [50]) is joint with Brian Huffman, Fangfei Liu, and Carlos V. Rozas of Intel Labs and Sanjit A. Seshia; Intel collaborators contributed the SVA backend and industrial case studies. The outlook discussion of *Syzygy* (Chapter 10, [109]) is joint with Manish Shetty (lead author), Naman Jain, Sanjit A. Seshia, and Koushik Sen.

Open-source artifacts. The tools and benchmarks developed for the technical chapters of this thesis are released as open-source reproducibility artifacts; URLs are given at the opening of each chapter.

Chapter 2

Automated Verification and Synthesis

This chapter establishes the technical foundation that the rest of the thesis builds upon. The contributions in Chapters 4–10.4 each combine verification and synthesis in domain-specific ways; this chapter introduces the general-purpose machinery they share.

We begin with *formal models*—transition systems and their symbolic representations—that capture how hardware systems behave (Section 2.1). We then introduce *specifications*—temporal logics, invariants, and hyperproperties—that express the properties we require (Section 2.2). With models and specifications in hand, we describe *verification techniques*—bounded model checking, IC3/PDR, and abstraction-based reasoning—that determine whether a model satisfies a specification (Sections 2.3–2.4). Finally, we present *automated synthesis*—the syntax-guided synthesis (SyGuS) problem family, the CEGIS algorithm that solves many SyGuS instances, and neural approaches—that automates the construction of models, specifications, and abstractions (Section 2.5).

2.1 Formal Models

A formal model defines system behavior mathematically, enabling automated reasoning about correctness and security. We introduce *transition systems* as the primary modeling formalism for this chapter; later chapters introduce additional formalisms as needed (axiomatic memory models in Chapter 7, resource partition models in Chapter 5). Transition systems are a natural fit for hardware: digital circuits advance in discrete clock cycles, with each cycle producing a new configuration of registers, memories, and control signals.

2.1.1 Transition Systems

A *transition system* $\mathcal{M} = (S, s_0, \rightarrow)$ consists of a set of states S , an initial state $s_0 \in S$, and a transition relation $\rightarrow \subseteq S \times S$. A *trace* $\tau = s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$ is a (possibly infinite) sequence of states where $s_i \rightarrow s_{i+1}$ for all i ; the set of all traces of $\mathcal{M}$ is denoted $\mathcal{L}(\mathcal{M})$.

To make this concrete, consider the Sodor 5-stage RISC-V pipeline that serves as a running example throughout this thesis. Each state encodes the program counter, the 32-entry register file, the contents of the pipeline latches between each pair of stages (fetch/decode, decode/execute, execute/memory, memory/writeback), the data memory, and the control signals governing forwarding and hazard detection. Even this modest pedagogical design has a state space far exceeding 2^{160} configurations. Transitions model one clock cycle of instruction execution: the fetch stage reads the instruction at the current PC and populates the decode latch, the decode stage reads source registers and fills the execute latch, and so on through memory access and writeback. A single transition relation captures all of this parallel activity, including pipeline stalls, flushes, and forwarding logic.

2.1.2 Symbolic Representation

Explicit state enumeration is infeasible for hardware designs whose state spaces exceed 2^{160} configurations; instead, we rely on *symbolic representations* that manipulate sets of states without enumerating them individually. Three strategies are common. *Boolean formulas* encode states as bitvector valuations and transitions as Boolean circuits, enabling SAT/SMT solvers to manipulate formulas without enumerating states. *Binary Decision Diagrams (BDDs)* provide canonical graph representations of Boolean functions, supporting efficient set operations such as union, intersection, and image computation. *Symbolic execution* runs programs on symbolic inputs and builds path constraints; while useful for software, it is less applicable to hardware verification.

Among these, Boolean formula representations with SAT/SMT backends (e.g., Z3 [89]) have become dominant for hardware verification due to the scalability of modern conflict-driven clause-learning solvers.

2.2 Formal Specifications

A formal model captures how a system behaves; a *specification* captures what we want from it. Because hardware systems evolve over discrete clock cycles, specifications must express requirements about temporal behavior—not just “what holds now” but “what must always hold” and “what must eventually happen.” Temporal logics such as LTL [99] and CTL [25] provide formal languages for such requirements; this thesis works primarily with *safety properties*, which assert that undesirable states are never reached.

Most hardware verification properties take an *implication* form: $\Box(\text{assumption} \Rightarrow \text{guarantee})$, stating that whenever the assumption holds, the guarantee must follow. This pattern recurs throughout the thesis; for example, micro-update models (Chapter 4) use guards to define when a micro-update is observable, and the property asserts that when the guard is satisfied, the abstract model faithfully refines the concrete RTL.

2.2.1 Temporal Properties vs. Invariants

Temporal properties expressed as LTL formulas may involve arbitrary combinations of $\Box$, $\Diamond$, and $\mathcal{U}$ operators. In practice, however, most hardware verification properties take a simpler form: they are *invariants*—state predicates that must hold in every reachable state. An invariant can be expressed in LTL as $\Box p$ where p is a purely state-level formula without temporal operators.

Checking whether a predicate p is an invariant requires, in principle, computing the full set of reachable states—a problem that is itself computationally intractable for large systems. The key technique for sidestepping this difficulty is *inductive invariants*. An inductive invariant I is a predicate that is strong enough to prove itself: it holds in the initial state, is preserved by every transition, and implies the property of interest. Formally, I must satisfy three conditions:

1. $s_0 \models I$ (holds initially)
2. $s \models I \wedge s \rightarrow s' \Rightarrow s' \models I$ (preserved by transitions)
3. $I \Rightarrow \varphi$ (implies the desired property)

The challenge is that the property φ itself is rarely inductive—there are typically unreachable states that violate φ but are reachable from other states satisfying φ , causing the induction step to fail. Finding the right strengthening $I \Rightarrow \varphi$ that *is* inductive requires discovering auxiliary facts about the system that rule out these unreachable states. This is where synthesis becomes essential: Chapters 7–9 present techniques that automate the construction of verification artifacts—operational models, axioms, and invariants—replacing manual trial-and-error with systematic search.

2.2.2 Hyperproperties

The temporal logics introduced above—LTL, CTL, and invariants—are expressive enough to capture many important properties, but they share a fundamental limitation: each describes the behavior of a *single* execution trace in isolation. Some of the most important security properties, however, are inherently *relational*: they constrain how two or more executions relate to each other. Non-interference, for instance, requires that changing a secret input does not affect public outputs—a statement that only makes sense when comparing two executions. *Hyperproperties* [27] generalize trace properties to capture exactly these relational requirements: they are properties of *sets* of traces rather than individual traces.

Formally, a *k-safety hyperproperty* requires that for every k -tuple of traces $(\tau_1, \dots, \tau_k)$, a relational condition $\Psi(\tau_1, \dots, \tau_k)$ holds; standard trace properties are the special case $k = 1$.

As a concrete example, *non-interference* is a 2-safety hyperproperty: two executions that start from states agreeing on public inputs must produce identical public outputs.

$$\forall \tau_1, \tau_2. \tau_1[0] =_L \tau_2[0] \implies \forall i. \mathcal{O}(\tau_1[i]) = \mathcal{O}(\tau_2[i])$$

Hyperproperties are central to security (Chapter 6): non-interference, confidentiality, and integrity are all hyperproperties. Standard model checkers verify trace properties, not hyperproperties, creating a fundamental tool gap.

This raises a practical question: how can we verify hyperproperties using standard model checkers that are designed for single-trace properties? The standard technique is *self-composition* [116]. Given a system $\mathcal{M}$, we construct the *self-composed* system $\mathcal{M} \times \mathcal{M}$ whose state is a pair (s_1, s_2) representing two simultaneous executions, with transitions that advance both copies in lockstep. The 2-safety hyperproperty then reduces to an ordinary safety property on the product system:

$$\mathcal{M} \models \text{NI} \iff (\mathcal{M} \times \mathcal{M}) \models \Box(s_1 =_L s_2 \implies \mathcal{O}(s_1) = \mathcal{O}(s_2))$$

Self-composition converts a hyperproperty into a trace property, enabling standard model checking—but at a significant cost. The product system has *twice* as many state variables as the original, roughly squaring the state space. For hardware designs that are already at the limit of verification scalability, this doubling can push the problem out of reach. Much of the work in Chapters 6 and 9 is motivated by this cost: synthesizing abstractions and invariants that make self-composed verification tractable.

2.3 Verification Techniques

Given a formal model $\mathcal{M}$ and a specification φ , verification asks whether every behavior of $\mathcal{M}$ satisfies φ —written $\mathcal{M} \models \varphi$. When the answer is yes, verification produces a *proof*; when the answer is no, it produces a *counterexample* witnessing the violation.

2.3.1 Model Checking

Model checking [26] automates this question by algorithmically exploring the state space of $\mathcal{M}$, either proving $\mathcal{M} \models \varphi$ or returning a counterexample trace.

2.3.1.1 Bounded Model Checking (BMC)

BMC [9] checks a property up to a fixed depth k by unrolling the transition relation k times and encoding the question as a single SAT query:

$$\text{SAT}(\text{Init}(s_0) \wedge \bigwedge_{i=0}^{k-1} \text{Trans}(s_i, s_{i+1}) \wedge \neg\varphi)$$

If the formula is satisfiable, the satisfying assignment is a counterexample trace witnessing a bug. If it is unsatisfiable for all k up to a completeness threshold, the property holds. The principal advantage of BMC is that it reduces verification to SAT, leveraging the remarkable scalability of modern conflict-driven clause-learning solvers.

However, BMC provides only a *bounded* guarantee: the absence of a counterexample at depth k does not prove that no counterexample exists at depth $k + 1$. To obtain an

unbounded proof via BMC, one must determine the *completeness threshold*—the depth at which the absence of counterexamples guarantees correctness. This threshold corresponds to the diameter of the reachability graph, which is typically unknown and may itself be exponential in the number of state variables. In practice, BMC is therefore most useful for bug-finding (where a counterexample at any depth constitutes a proof of failure) rather than for proving correctness.

2.3.1.2 IC3/PDR

IC3 (Incremental Construction of Inductive Clauses) and PDR (Property Directed Reachability) [12, 32] take a fundamentally different approach from BMC. Rather than searching forward from initial states for a bounded number of steps, they combine forward overapproximation with backward analysis of potential violations, systematically ruling out states that could lead to a violation by learning clauses that overapproximate the reachable state set. The key insight is that the algorithm never needs to unroll the transition relation to a fixed depth; instead, it incrementally constructs an inductive invariant that proves the property for all depths simultaneously.

The algorithm maintains a sequence of *frames* $F_0, F_1, \dots, F_k$, where each F_i overapproximates the set of states reachable in at most i steps:

1. Check whether the property can be violated in F_k
2. If a potential violation is found, analyze it and generalize it to a *blocking clause*—a learned fact that rules out not just the specific counterexample but an entire class of similar states
3. Push learned clauses forward through the frame sequence; if two consecutive frames become identical ($F_i = F_{i+1}$), the algorithm has found an inductive invariant and the property is proven

IC3/PDR provides fully automated, *unbounded* verification: when it succeeds, the result is a proof that the property holds for all execution depths, not merely up to some bound. Moreover, it often discovers inductive invariants comprising many clauses that would be impractical to construct by hand. This capability is particularly valuable for hardware verification, where the necessary invariants involve intricate relationships among pipeline stages, buffers, and control signals. Ax2Op (Chapter 7) enables IC3 for axiomatic memory models by converting them to the operational transition-system representation that IC3 requires.

2.3.2 Other Techniques

Other verification approaches include *deductive verification* (interactive proof in Coq or Isabelle/HOL, offering maximum expressiveness at the cost of manual effort) and *simulation-based validation* (testing with random or directed stimuli, scalable but incomplete). This

thesis focuses on automated model-checking techniques and does not employ deductive verification.

2.4 Abstraction

The model-checking techniques above struggle when applied directly to large hardware designs. Abstraction addresses this by replacing a complex model $\mathcal{M}$ with a simpler model $\widehat{\mathcal{M}}$ that preserves the property of interest, relying on the implication $\widehat{\mathcal{M}} \models \varphi \Rightarrow \mathcal{M} \models \varphi$ to transfer the result. Soundness is formalized through a *simulation relation* $R \subseteq S_{\text{concrete}} \times S_{\text{abstract}}$ connecting concrete and abstract states such that every concrete transition is matched by an abstract transition.

When an abstraction is too coarse, verification produces spurious counterexamples—violations that exist in the abstract model but not the concrete one. Counterexample-guided abstraction refinement (CEGAR) [24] automates the repair: it analyzes the spurious counterexample, refines the abstraction to eliminate it, and repeats.

Among the most effective abstractions are *property-directed* ones: they preserve exactly the distinctions the property requires while discarding everything else. This thesis develops two such abstractions: micro-update models (Chapter 4) abstract speculative execution to preserve information-flow paths, and conditional equality invariants (Chapter 5) abstract resource partitioning to preserve isolation guarantees. Constructing these abstractions manually is the practical bottleneck in formal verification—and is the problem that synthesis addresses.

2.5 Automated Synthesis

Verification checks whether a given artifact (specification, invariant, abstraction) is correct; *synthesis* generates the artifact in the first place. In practice, manual construction of verification artifacts is often a significant component of the cost of formal verification projects. Automated synthesis addresses this bottleneck by searching for artifacts that satisfy formal constraints.

2.5.1 The Synthesizer-Verifier Pattern

The use of synthesis to automate verification sub-problems — invariant discovery, abstraction construction, environment modeling — was proposed by Seshia [104, 103], combining inductive learning from examples with deductive reasoning and domain structure. Greben-shchikov et al. [55] took a deductive approach, synthesizing software verifiers from proof rules via constrained Horn clause solving. The roots of this idea trace back to deductive program synthesis [87].

All synthesis techniques in this thesis share a common structure: a *synthesizer* proposes candidate artifacts, and a *verifier* checks them (Figure 2.1). This pattern is an instance of

oracle-guided inductive synthesis (OGIS) [64], which abstracts the interaction as a query-response interface between a learner and one or more oracles. The pattern is parameterized along three axes.

The *synthesis space* $\mathcal{S}$ defines the set of candidate artifacts the synthesizer may explore. The space may be specified by a context-free grammar (as in SyGuS [3]), a template with designated holes (as in sketching [114]), or a learned distribution over programs (as in LLM-based synthesis). The tighter the synthesis space, the faster synthesis converges—but an overly restrictive space may exclude the correct solution. Designing the synthesis space is where domain expertise enters: the space must be expressive enough to contain the target artifact while small enough to be searchable.

The *synthesizer algorithm* determines how candidates are generated from the synthesis space. Counterexample-guided inductive synthesis (CEGIS, described below) iterates between proposing candidates and refining them using verifier feedback. Enumerative synthesis systematically explores the grammar in order of increasing complexity. Neural synthesis samples from a learned distribution, exploiting statistical priors to propose candidates that are likely close to correct. Each approach offers a different trade-off: CEGIS is complete for finite synthesis spaces but can be expensive per iteration; enumerative methods are simple but scale poorly with grammar depth; neural methods are fast but offer no completeness guarantees.

The *verifier algorithm* checks whether a candidate satisfies the specification, returning either a proof of correctness or a counterexample witnessing a violation. The verifier can be instantiated with different engines depending on the desired guarantee: BMC provides fast bounded checks suitable for bug-finding, IC3/PDR provides unbounded proofs at higher computational cost, and simulation provides rapid but incomplete validation. Crucially, the verifier’s counterexamples serve double duty: they certify that a rejected candidate is incorrect *and* provide concrete information that the synthesizer can use to avoid similar mistakes in subsequent iterations.

The interaction between synthesizer and verifier is the key to both correctness and efficiency: the verifier ensures that the final artifact is sound, while counterexample feedback steers the synthesizer away from incorrect candidates.

2.5.2 Counterexample-Guided Inductive Synthesis (CEGIS)

Solar-Lezama et al. [114] introduced *program sketching*, partial programs with designated holes, and the *counterexample-guided inductive synthesis* (CEGIS) algorithm for filling them. In sketching, the user provides the program’s high-level structure, and the synthesizer determines the low-level details (constants, expressions, choice of operations) that make the sketch satisfy a specification. CEGIS solves this by iterating between a synthesizer and a verifier. The synthesizer maintains a set of constraints $\mathcal{C}$ (initially empty or seeded with examples) and proposes a candidate completion A satisfying $\mathcal{C}$. The verifier checks whether A satisfies the full specification φ . If verification succeeds, A is returned. If it fails with

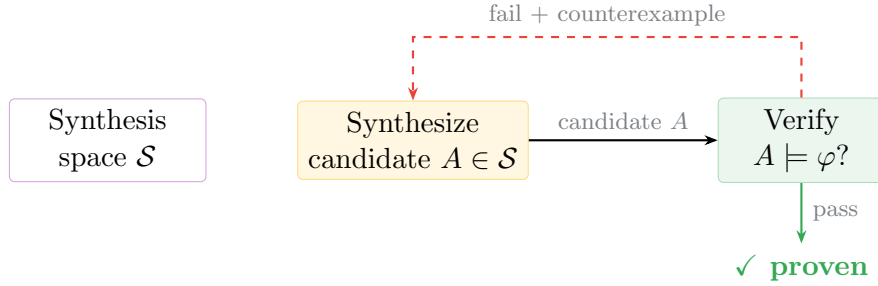


Figure 2.1: The synthesizer-verifier loop: the synthesizer generates candidates from a synthesis space $\mathcal{S}$, the verifier checks whether each candidate satisfies specification φ , and counterexamples refine the search. The loop terminates when a valid artifact is found or the space is exhausted.

a counterexample cex , the counterexample is added to $\mathcal{C}$ and synthesis repeats. The loop terminates when a valid artifact is found or the search space is exhausted.

CEGIS is effective because each counterexample permanently eliminates a class of incorrect candidates, ensuring progress. This thesis uses CEGIS in three settings: synthesizing attack patterns from semantic policies (Chapter 6), synthesizing μspec axioms from litmus tests (Chapter 8), and synthesizing inductive invariants from properties (Chapter 9).

2.5.3 Syntax-Guided Synthesis (SyGuS)

Syntax-guided synthesis [3] constrains the synthesis space $\mathcal{S}$ by requiring candidates to conform to a user-provided *grammar*. The key insight is that domain expertise, even when insufficient to write the solution directly, often suffices to constrain its syntactic structure. For example, PipeSynth (Chapter 8) defines axiom templates—path, FIFO, and sourcing—that capture the structural patterns common to memory ordering specifications. Rather than searching over arbitrary logical formulas, synthesis is restricted to conjunctions of predicates drawn from these templates, dramatically reducing the search space. Similarly, MUM guard synthesis (Chapter 4) restricts guards to Boolean combinations of trigger signal predicates, and SemPat (Chapter 6) restricts attack patterns to sequences drawn from an instruction grammar.

Sketching can be viewed as a special case of SyGuS in which the grammar is derived from the sketch template: the holes define the synthesis space, and the surrounding program structure constrains it. SyGuS generalizes this by allowing arbitrary grammars not tied to a specific program skeleton. CEGIS is one algorithm for solving SyGuS instances; solvers such as CVC5 [5] also implement enumerative and hybrid strategies. In summary: SyGuS defines the *problem* (what to synthesize, from what grammar, against what spec); CEGIS provides one *algorithm* for solving it. Most synthesis tasks in this thesis combine both.

UCLID5 [105, 100] implements this combination by reducing synthesis-mode queries to

Table 2.1: Synthesis instantiations across the thesis.

Artifact	Synthesis Space	Synthesizer	Verifier	Ch.
MUM guards	Boolean trigger predicates	SyGuS (CVC5)	SymbiYosys	4
Attack patterns	Templates + Grammar	Counterfactuals	UCLID5 (Z3)	6
Operational automata	Finite-state machines	Algorithmic	SymbiYosys	7
μ spec axioms	Axiom templates	SyGuS (CVC5)	Litmus tests	8
Inductive invariants	PyCaliper Specs.	Misc. (CEGIS/LLM)	Jasper/BTOR	9

SyGuS problems within a unified verification and synthesis environment. Zhang et al. [140, 139] applied SyGuS with design-informed grammars to synthesize auxiliary lemmas and environment invariants for hardware model checking.

2.5.4 LLM-Based Synthesis

Large language models offer a fundamentally different approach to candidate generation: instead of searching a grammar-defined space with constraint solvers, LLMs sample from a learned distribution over programs and specifications [22]. When the LLM’s training distribution aligns with the target domain, candidates may be close to correct on early attempts, converging faster than enumerative search.

The central challenge is correctness: LLM-generated artifacts may be plausible yet subtly wrong. The *verify-after-generate* paradigm addresses this by treating the LLM as a fast but unreliable candidate generator within the synthesizer-verifier pattern. The LLM proposes candidates; a formal verifier accepts or rejects each one; rejected candidates and their counterexamples can optionally be fed back to the LLM for refinement. This preserves the soundness guarantees of classical synthesis while benefiting from the speed of learned priors.

PyCaliper (Chapter 9) supports this hybrid approach: its pluggable synthesis backend can be instantiated with CEGIS (constraint-based), random search (baseline), or LLM-guided generation, all verified against the same SVA/BTOR2 backends. Chapter 10.4 explores this direction further.

2.5.5 Instantiations in *this* Thesis

Table 2.1 summarizes how the synthesizer-verifier pattern is instantiated across the thesis.

The table highlights a recurring theme: the *same* synthesizer-verifier pattern is instantiated with different synthesis spaces and algorithms for each domain. The synthesis space encodes domain knowledge (axiom templates, instruction grammars, specification holes), keeping the search tractable. The verifier provides the soundness guarantee, regardless of how the candidate was generated.

With this formal machinery in hand, the next chapter applies it to two concrete application domains—hardware security and memory consistency—formalizing the specific verification objectives that the technical chapters address.

Chapter 3

Challenge: Reasoning at the Hardware-Software Interface

Software assumes an *architectural abstraction*—a sequential ISA with well-defined memory semantics. Hardware implements this abstraction through complex microarchitecture: speculation, caching, out-of-order execution, and relaxed memory models. When the implementation deviates from the abstraction, vulnerabilities emerge.

This chapter formalizes the verification challenges at the hardware-software interface through two application domains: *hardware security* (Section 3.2) and *memory consistency* (Section 3.3). For each domain, we present concrete examples and then formalize the verification objectives using the transition system framework from Chapter 2. The formal definitions introduced here — Speculative Non-Interference (Definition 3.1), Conditional Equality (Definition 3.2), and the axiomatic and operational memory model paradigms (Definitions 3.4–3.6) — serve as the target specifications that the technical chapters develop techniques to establish.

3.1 The Formal Setting

We specialize the transition system framework from Chapter 2 to the hardware-software interface.

A hardware-software system comprises hardware H (a transition system whose state includes both architectural components — registers, memory, program counter — and micro-architectural components — caches, buffers, predictors), software P (a program mapping addresses to instructions), and an observation function $\mathcal{O}$ capturing what an attacker or environment can observe (varies by threat model).

Systems produce *execution traces*: sequences of states evolving under the hardware’s transition relation. What constitutes a “state” and how traces are structured varies by domain. For security analysis (Chapters 4–6), states include cache contents, predictor entries, and pipeline registers; traces record cycle-by-cycle state evolution. For memory consistency

(Chapters 7–8), traces are sequences of memory events with happens-before orderings. Each of the following chapters formalizes these for its specific setting.

The central verification question is: *does the hardware H satisfy specification φ for all programs P ?* In the security domain, φ is formalized as non-interference — secrets must not influence attacker observations. In the memory consistency domain, φ is formalized as memory model compliance — hardware orderings must respect the memory model. The remainder of this chapter instantiates φ for each domain.

3.2 Hardware Security

Modern processors optimize performance through speculative execution, caching, and resource sharing. These optimizations create *side channels*—observable behaviors that leak secret information. We ground our analysis in a concrete scenario: *a RISC-V processor executing a T-table AES-128 encryption kernel* (introduced in Section 1.1).

T-table AES computes each encryption round by indexing into precomputed lookup tables:

```

1 // AES round: each output column uses shifted input bytes
2 // Column 0 shown; columns 1-3 use different ShiftRows indices
3 state[0] = Te0[s[0]] ^ Te1[s[5]]
4             ^ Te2[s[10]] ^ Te3[s[15]] ^ round_key[0];
5 state[1] = Te0[s[4]] ^ Te1[s[9]]
6             ^ Te2[s[14]] ^ Te3[s[3]] ^ round_key[1];
7 // ... columns 2, 3 analogous

```

Listing 3.1: T-table AES round (simplified)

Each table lookup $\text{Te0}[s[j]]$ accesses a memory address determined by intermediate cipher state $s[j]$, which depends on the secret key through the AES SubBytes and MixColumns operations. On a speculative processor with caches, store buffers, and branch predictors, this single kernel creates multiple attack surfaces. The running examples that follow each examine a different facet of this system.

3.2.1 Running Example 1: AES T-Table Cache Side Channel

The first attack surface exploits caching and speculative execution. On a processor with a cache, speculative memory accesses leave *microarchitectural residue*, persistent changes to cache state that survive the architectural rollback following a misprediction. A cache hit (fast) versus miss (slow) reveals whether a particular address was accessed during speculation, creating a timing side channel.

The following code fragment illustrates the attack concretely:

```

1 // AES encryption code with bounds-check bypass
2 if (x < 16) { // Branch mispredicted as taken

```

```

3   s = plaintext[x] ^ key[x]; // Speculative: s depends on key
4   y = Te0[s];               // T-table lookup at key-dependent
                               index
5 }
```

Listing 3.2: Spectre-style gadget exploiting T-table AES lookup

When the branch is mispredicted, the speculative T-table lookup `Te0[s]` accesses a cache line at an index determined by `key[x]`. The cache state persists post-squash: the attacker probes which of the 16 possible cache lines (assuming 64-byte cache lines and 4-byte T-table entries) was loaded, recovering the upper bits of the secret-dependent index `s = plaintext[x]  $\oplus$  key[x]`. This is the T-table variant of the Spectre-V1 (Bounds Check Bypass) attack [68].

The precise vulnerability also depends on microarchitectural implementation choices, such as at which pipeline stage a buffer is invalidated. Chapter 4 develops a detailed load buffer example (Example 4.1) that illustrates this dependence and motivates micro-update models.

3.2.2 Running Example 2: Branch Target Injection on TAGE Predictor

Example 3.1 (Branch Target Injection on TAGE Predictor). Consider a processor equipped with an indirect-branch predictor in the TAGE family [108, 107], whose tagged history tables index by branch address and path history to predict the next indirect-branch target. Critically, these predictor tables are shared across privilege domains (user/kernel, different processes), creating a cross-domain attack surface.

A Branch Target Injection (BTI) attack [68] exploits this shared resource:

1. Attacker (in domain d_A) executes branches to poison predictor tables.
2. Poisoned entries map victim branch addresses to attacker-controlled targets.
3. Victim (in domain d_V) executes a branch; the predictor returns the attacker’s target.
4. Victim speculatively jumps to the attacker-controlled gadget.
5. Speculative execution leaks victim’s secrets via cache side-channel.

A standard mitigation is to partition predictor tables by domain, ensuring that entries from d_A cannot influence predictions for d_V .

The vulnerability arises from *resource sharing* across security domains: predictor tables are indexed by branch address and history, and entries from one domain can influence predictions in another. The mitigation, resource partitioning, must be formally verified. The formal requirement is precisely a conditional equality invariant (Definition 3.2): predictor

table entries must agree between two executions whenever they belong to the attacker’s domain in both executions, but may differ freely when they belong to the victim’s domain.

3.2.3 Formalizing Security: Speculative Non-Interference

We now formalize what it means for hardware to be “secure” against speculative side-channel attacks. Security properties are defined relative to a threat model: which state components are secret (e.g., cryptographic keys, memory contents) and which are observable by the attacker (e.g., cache hit/miss timing, predictor behavior). Two states are *low-equivalent* [52], written $s_1 =_L s_2$, if they agree on all public (non-secret) components. Each of the following chapters instantiates this threat model for its domain: Chapter 4 uses signals of interest and observable outputs; Chapter 5 uses protection domains with a distinguished attacker domain; Chapter 6 uses public and observable variable sets.

Definition 3.1 (Speculative Non-Interference (SNI)). Hardware H satisfies *Speculative Non-Interference* under observation $\mathcal{O}$ iff for all programs P and all pairs of traces τ_1, τ_2 induced by P :

$$s_1^0 =_L s_2^0 \implies \forall i. \mathcal{O}(s_1^i) = \mathcal{O}(s_2^i)$$

where $\tau_j = s_j^0 \rightarrow s_j^1 \rightarrow \dots$ and states differ only in secrets.

In words: if two executions start with identical public state, differing only in secrets, then the attacker’s observations must be identical at every step—secrets cannot influence observable behavior.

Remark 3.1 (SNI is a Hyperproperty). Definition 3.1 quantifies over *pairs* of traces—it is a 2-safety hyperproperty [27] (cf. Section 2.2.2). This has significant practical implications: standard model checkers verify properties of individual traces and cannot directly handle pairwise comparisons. Verifying SNI requires either self-composition [116] (constructing the product system $H \times H$ and checking a trace property on the product, which roughly squares the state space) or reduction to trace properties via abstraction. This state-space expansion makes direct verification prohibitively expensive for realistic designs, motivating the abstraction-based approaches developed in Chapters 4 and 6.

Remark 3.2 (Alternative formalizations of speculative-execution security). SNI is not the only formalization of security for speculative-execution attacks. Cheang et al. [21] introduce *trace-property-dependent observational determinism* (TPOD), a hyperproperty parameterized by a trace property of non-speculative behavior that compares speculative executions to their non-speculative counterparts. Other formulations include hardware-software contracts that delineate which microarchitectural side-effects software may rely upon [57] and observational-equivalence variants tailored to specific threat models. SNI is the formulation used in this thesis because it cleanly captures the relational nature of side-channel attacks; the abstraction- and synthesis-based techniques developed here apply, with minor adjustments, to the alternative formulations as well.

3.2.4 Formalizing Security: Conditional Equality

Speculative Non-Interference provides a strong, blanket guarantee—no information flows from secrets to observations under any circumstances. However, for resource partitioning schemes such as the TAGE mitigation described in Example 3.1, we need a finer notion that accounts for the intentional sharing of hardware resources across security domains. The key observation is that resources allocated to different domains should be isolated, but resources within the *same* domain are permitted to differ between the two executions being compared.

Definition 3.2 (Conditional Equality Invariant). A *conditional equality invariant* ϕ_{inv} over state pairs (s_1, s_2) has the form:

$$\phi_{\text{inv}}(s_1, s_2) \triangleq \bigwedge_{r \in \text{Resources}} (\text{Cond}_r(s_1, s_2) \implies r(s_1) = r(s_2))$$

where Cond_r specifies when resource r must be equal.

For example, TAGE partitioning requires that for predictor table entry at index i :

$$\text{Cond}_i(s_1, s_2) \triangleq \text{Domain}(i, s_1) = \text{Domain}(i, s_2) = d_{\text{attacker}}$$

“Predictor entries must be equal only if they belong to attacker’s domain in both executions.”

Chapter 5 develops a concrete instantiation of conditional equality invariants for resource partitioning (cache way-partitioning, page coloring), with specific verification conditions (base case, inductive attacker/victim steps, observation) and a filtering function that determines which policy state elements must agree.

3.3 Memory Consistency

Having formalized the security verification problem, we turn to the second application domain. Memory consistency models (MCMs) [1] define the observable orderings of memory operations in multi-threaded programs. The simplest MCM, Sequential Consistency (SC) [71], requires the results of memory operations to be consistent with a total order in which each core’s operations appear in program order. Hardware relaxes SC for performance (e.g., TSO [106], RVWMO [126]); software relies on the resulting MCM guarantees for correct synchronization. Violations cause subtle, hard-to-reproduce bugs.

The next two running examples illustrate how relaxed memory models create subtle correctness challenges for multi-threaded programs.

3.3.1 Running Example 3: Store Buffering under Relaxed Memory

Example 3.2 (Store Buffering under Relaxed Memory). Consider a dual-core processor in which each core has a private store buffer that holds committed stores before they propagate

to shared memory. Local loads can forward from the store buffer, a mechanism known as store-buffer forwarding. The following store buffering litmus test illustrates the ordering anomaly:

Core 0	Core 1
sw x1, 1, (a)	sw x2, 1, (b)
lw r1, (b)	lw r2, (a)

The question is whether the final state $r1 = 0 \wedge r2 = 0$ can occur. Under Sequential Consistency (SC), it cannot: at least one store globally commits before the other core's load executes. Under TSO [106] or RVWMO [126], however, this outcome is permitted because each load reads from memory before the other core's store has propagated.

Example 3.2 shows a classic litmus test. The outcome $r1 = 0 \wedge r2 = 0$ is *forbidden* under SC but *allowed* under relaxed models like TSO and RVWMO. In a producer-consumer scenario, this means both cores may read stale values, leading to silent data corruption.

3.3.2 Running Example 4: Message Passing with Fences

Example 3.3 (Message Passing with Fences). The message passing litmus test captures a producer-consumer pattern:

Core 0 (Writer)	Core 1 (Reader)
sw x1, 42, (data)	lw r1, (flag)
sw x2, 1, (flag)	lw r2, (data)

The intent is that the writer stores data and then sets a flag, while the reader waits for the flag and then reads the data. The question is whether the outcome $r1 = 1 \wedge r2 = 0$ can occur, that is, the reader sees the flag but reads stale data. Under SC this outcome is forbidden: if the flag store is visible, the data store must also be visible. Under RVWMO, the answer depends on fence placement; without **fence** instructions, loads can be reordered, breaking the intended synchronization pattern.

Correct synchronization requires explicit fences:

Core 0	Core 1
sw x1, 42, (data)	lw r1, (flag)
fence w,w	fence r,r
sw x2, 1, (flag)	lw r2, (data)

Example 3.3 illustrates message passing. Without fences under RVWMO, the reader may observe the flag before the data is visible, using stale or uninitialized values. The correctness depends on fence placement, and this failure is timing-dependent and non-deterministic: it may occur only under specific pipeline conditions (e.g., when the store buffer is full or when a cache miss delays propagation), making it nearly impossible to detect through testing alone. Formal verification provides stronger guarantees of correct synchronization than testing alone.

3.3.3 Formalizing Memory Consistency: Two Paradigms

Memory models can be specified in two paradigms, each with different strengths.

Definition 3.3 (Operational Memory Model). An *operational memory model* is a transition system $\mathcal{M}_{\text{op}} = (S, s_0, \rightarrow)$ whose state S includes per-core pipelines, store buffers, coherence protocol state, and global memory. Transitions model local operations, buffer drains, and coherence messages. Observable behaviors are sequences of load return values.

Definition 3.4 (Axiomatic Memory Model). An *axiomatic memory model* [2] specifies valid executions via constraints on *happens-before* orderings. Given an execution with events (loads, stores, fences), the model defines relations: program order po (intra-thread), reads-from rf (store to the load that reads its value), coherence order co (total order on writes to the same address), and the derived happens-before hb . An execution is valid iff hb is acyclic and all axioms are satisfied.

The two paradigms offer complementary strengths. Operational models are intuitive for hardware designers and directly compatible with model checkers (BMC, IC3, PDR), but they tend to be verbose and implementation-specific, making it difficult to compare the memory ordering guarantees of different designs. Axiomatic models are concise and implementation-independent, capturing ordering constraints that apply across a family of designs, but they are incompatible with most verification tools, which expect state machines rather than declarative constraints. This paradigm mismatch is the central challenge that Chapter 7 addresses.

Definition 3.5 (Memory Model Compliance). Hardware H is *compliant* with axiomatic model $\mathcal{M}_{\text{ax}}$ iff for all programs P :

$$\text{Soundness: } \forall \tau \in \mathcal{L}(H, P). \text{Obs}(\tau) \in \mathcal{L}(\mathcal{M}_{\text{ax}}) \quad (3.1)$$

$$\text{Completeness: } \forall E \in \mathcal{L}(\mathcal{M}_{\text{ax}}). \exists \tau \in \mathcal{L}(H, P). \text{Obs}(\tau) = E \quad (3.2)$$

where $\text{Obs}(\tau)$ extracts observable memory events from hardware trace.

Soundness requires that the hardware never exhibits forbidden behaviors; completeness requires that it can produce all allowed behaviors.

3.3.4 The μspec Axiomatic Framework

We use μspec [78, 80] as our axiomatic specification language. μspec axioms constrain orderings between microarchitectural events.

Definition 3.6 (μspec Axiom). A μspec axiom has the general form:

$$\forall i_1, \dots, i_n. \text{Body}(i_1, \dots, i_n)$$

where $i_1, \dots, i_n$ are instruction variables and Body is a quantifier-rich formula over instruction attributes (opcode, address, same-thread, etc.) and happens-before edges. The most

common case asserts an ordering as $\text{Pred}(i_1, \dots, i_n) \Rightarrow \text{Edge}(i_j, i_k) \in \text{hb}$, but Body may also contain existential subformulas, allowing axioms to assert, e.g., the *existence* of a supplying store for each load.

For example, a store-load ordering axiom states:

$$\forall i, j. \text{IsStore}(i) \wedge \text{IsLoad}(j) \wedge \text{SameThread}(i, j) \wedge \text{po}(i, j) \implies (i, j) \in \text{hb}$$

“Within a thread, stores happen-before subsequent loads.” This axiom holds under Sequential Consistency; under relaxed models such as TSO, store-load reordering is permitted unless a fence intervenes.

3.4 Verification Challenges

The definitions above establish *what* we want to prove in each domain. We now turn to *why* proving these properties is difficult. Despite their surface differences, hardware-security verification and memory-consistency verification share the same underlying difficulties; we present them together, in unified form, with concrete instances drawn from both domains. We identify four broad challenges.

Challenge 3.1 (Complexity of the Implementation). Modern hardware implementations contain millions of state bits spread across speculative-execution buffers, caches, predictors, and reorder buffers. Verifying any non-trivial property over the full implementation state is intractable without abstraction. Microarchitectural *state-space explosion* is the most direct manifestation: even a modest design like the Sodor 5-stage pipeline combined with a 64-line cache and a 512-entry predictor has a state space exceeding 2^{1000} configurations—far beyond the reach of BMC or IC3 without abstraction. The same complexity affects memory consistency, where verifying ordering properties across multi-core implementations requires reasoning about per-core pipelines, store buffers, and coherence protocols simultaneously. This challenge motivates the abstractions developed in Chapters 4 and 7.

Challenge 3.2 (Choosing the Right Level of Abstraction). Once we accept that abstraction is necessary, a deeper question follows: *at what level of abstraction is the attack surface—or, more generally, the property of interest—exposed?* The ISA layer is too high: it hides the microarchitectural buffers that side channels exploit and the orderings that memory consistency depends on. Full RTL is too low: verification becomes intractable, and the engineer is overwhelmed by detail unrelated to the property under analysis. The right level lies somewhere in between, and it differs across properties: a cache-timing side channel requires modeling cache state but not branch-predictor history; a Branch Target Injection attack requires the opposite; a memory consistency proof requires pipeline ordering but not functional-unit detail.

Choosing this level by hand requires deep domain expertise and risks missing security-relevant behaviors entirely. This challenge cuts across all parts of the thesis: micro-update

models (Chapter 4) expose precisely the microarchitectural buffers that speculative side channels exploit; the resource-partitioning framework (Chapter 5) exposes the policy-state and resource-allocation maps that isolation depends on; SemPat (Chapter 6) exposes the attack surface at the instruction-pattern level; and Ax2Op (Chapter 7) exposes happens-before structure at the granularity required for operational verification.

Challenge 3.3 (Model-Implementation Correspondence). An abstract model or formal specification, once chosen, is only useful if it faithfully represents the implementation it describes. Three failure modes recur across both domains.

Abstraction unsoundness: the abstract model satisfies a property that the concrete implementation violates, because the abstraction was too coarse. Proving soundness in general requires a simulation relation and an inductive invariant establishing it, which is often as hard as the original verification problem.

Pre-implementation specification drift: a formal specification written before the implementation exists may diverge from what the implementation eventually does, undermining any proofs about it. Writing axioms manually pre-RTL demands joint expertise in formal methods and microarchitecture.

Specification correctness: hand-written specifications—abstract models, axiom sets, security policies—may be *incomplete* (missing constraints allow forbidden behaviors), *overly restrictive* (extra constraints forbid allowed behaviors), or *inconsistent* (no execution satisfies them).

Chapter 4 provides correctness guarantees through S_{data} -equivalence; Chapter 8 addresses pre-RTL drift by synthesizing axioms from concrete examples; Chapter 7 enables model checking of the axioms themselves; and Chapter 9 automates the invariant synthesis on which abstraction soundness depends.

Challenge 3.4 (Specification-Form Tradeoffs). Finally, a specification language is shaped by two competing forces: it must be *natural to formulate* so that engineers can express intent, and *tractable to verify* so that tools can check it. These forces typically pull in opposite directions.

Semantic, relational, axiomatic specifications are concise and natural for human authors. Non-interference (Definition 3.1) directly captures “no information flows from secret to attacker”; a μ spec axiom (Definition 3.6) directly captures “stores happen-before subsequent loads in program order.” But standard verification techniques cannot consume them directly: hyperproperties require self-composition or relational model checking, and axiomatic specifications require operational reasoning that off-the-shelf model checkers do not support.

Syntactic, single-trace, operational specifications—automata, attack-pattern catalogs, transition systems—are directly executable by existing tools, but enumerating them by hand at the scale of a modern processor is error-prone and incomplete: attack-pattern catalogs miss variants; operational memory models drift from the implementations they describe.

This mismatch appears in both domains. In security, SNI is a hyperproperty (Remark 3.1) and cannot be checked by standard model checkers without self-composition; the equivalent

syntactic form (attack patterns) must be enumerated manually. In memory consistency, axiomatic μ spec specifications are incompatible with operational verification engines such as IC3 and PDR, and prior to this thesis no tool automatically bridged this gap. Chapters 5, 6, and 7 each address a different instance of this challenge.

3.5 Unifying Theme: Synthesis-Enabled Reasoning

The four challenges identified above share a common structural feature: in each case, the bottleneck is not running a verification engine but *constructing* the artifact the engine consumes—choosing a tractable abstraction at the right level (Challenges 3.1 and 3.2), aligning the model with the implementation (Challenge 3.3), or rendering the specification in a form the engine accepts (Challenge 3.4). This thesis adopts a single methodology, *synthesis-enabled reasoning* (SER), that closes all four by automating the construction of these reasoning artifacts through formal synthesis.

SER instantiates the synthesizer-verifier pattern introduced in Chapter 2 (Table 2.1): a synthesizer proposes candidate artifacts from a domain-structured search space, and a verifier checks each candidate, providing soundness independent of how it was generated. Domain structure—module boundaries, axiom templates, instruction grammars, partition schemata—keeps the search tractable while leaving the verifier’s guarantees intact. Each of the following chapters instantiates this pattern for one of the challenges above, demonstrating that the same methodology applies across security, memory consistency, and infrastructure.

Part I

The Implementation-to-Model Gap

The Implementation-to-Model Gap

Formal verification begins with a model, which exposes a view of the design that retains the behaviors relevant to the property under analysis while discarding detail that would otherwise make reasoning intractable. A well-chosen model exposes precisely the right information. For example, in the case of hardware security, a model may capture cache replacement decisions, pipeline interlocks, or speculative execution paths that determine whether secrets can leak, without the full complexity of the underlying RTL. With such a model, verification tools can explore all the design’s security-critical behaviors while searching through a much smaller state space as compared to the design. This is critical as modern systems such as chip designs have millions of logic bits resulting in state spaces of 2^{160} configurations or more (Challenge 3.1).

Constructing verification models by hand, however, is both labor-intensive and error-prone. A verification engineer must understand the design intimately enough to judge which signals matter for a given security property, then write and maintain a model that faithfully represents those signals across design revisions. Omit a critical signal and the model misses real vulnerabilities; include too much and the state space explodes. This is the *implementation-to-model gap*, the first gap identified in this thesis, and it reflects a fundamental tension: models must balance expressiveness against tractability, a tension that this chapter examines in detail. This part addresses the implementation-to-model gap from two angles.

Chapter 4

Micro-Update Models for RTL Security Reasoning

This chapter presents *micro-update models* (MUMs), a formalism for security verification of hardware that captures cycle-accurate behavior of security-relevant signals while abstracting away the remainder. A MUM is synthesized from the design’s register-transfer-level (RTL) description — the bit-precise specification, written in a hardware description language such as Verilog or VHDL, from which a synthesizable circuit is produced — and is provably equivalent to the RTL on the signals of interest. MUMs achieve 2–21× verification speedup on RISC-V processors. The following chapter (Chapter 5) presents a complementary approach using *conditional equality invariants* for verifying hardware isolation mechanisms. In both cases, domain-specific structure constrains the abstraction, making verification tractable while providing bounded soundness guarantees.

The material in this chapter is based on our ASPLOS 2024 paper [48]. Code and reproducibility artifacts are available at <https://github.com/adwait/asplos24-micro-update-lifting>, with a Zenodo snapshot at <https://zenodo.org/records/10552049>.

4.1 Balancing Expressivity and Verification Cost

Security verification of hardware requires a formal model that is simultaneously *expressive* enough to capture the behaviors under analysis and *tractable* enough for automated verification. Several works apply formal methods to verify software against hardware attacks [21, 58, 56, 88] and to discover new ones [118, 34]. Most of these operate on *manually crafted* abstract models that preserve only the vulnerability-relevant components of the hardware design. The expressiveness-tractability tension is resolved differently by each class of model, and each resolution comes with significant limitations.

ISA-level models [17, 111, 122, 54, 23] capture only architectural state (registers, memory, program counter) with temporal granularity at the instruction level, as do instruction-level abstraction (ILA) approaches for accelerators [63, 137]. This makes them compact

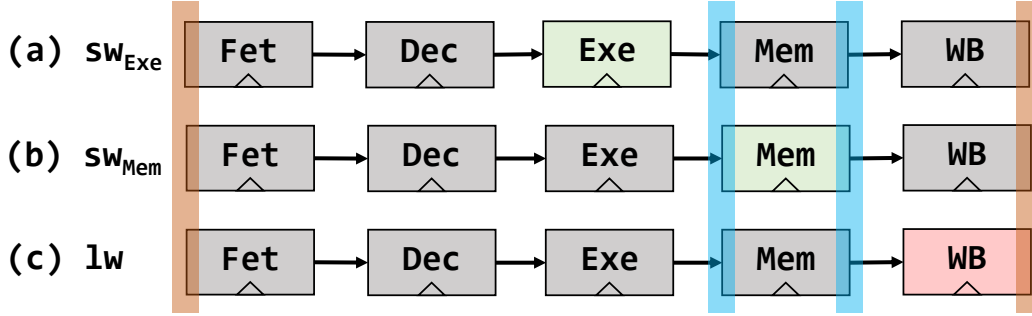


Figure 4.1: Pipeline timing for two store-word implementations and a dependent load in a 5-stage RISC-V pipeline. (a) sw_{Exe} : the store resolves at the execute stage (green), flushing the load buffer early. (b) sw_{Mem} : the store resolves at the memory stage (green), delaying the flush and allowing a dependent load (c) to progress further through the pipeline. The later flush point in (b) gives speculative instructions an additional pipeline stage in which to interact with the cache, widening the side-channel window.

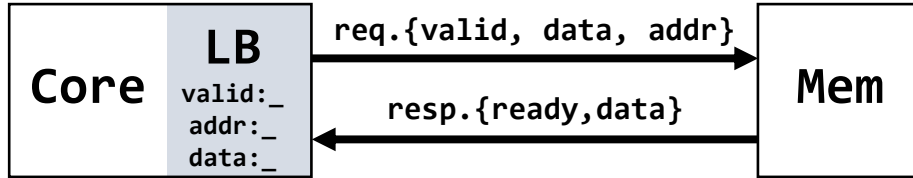


Figure 4.2: Schematic for the Sodor 5-stage processor. Our modification, the load buffer (LB), is shown in grey.

and efficient to verify, but they cannot express security-critical microarchitectural behaviors. ISA-level models can express certain software-level security contracts such as constant-time execution, which requires instruction counts and memory access sequences to be independent of secrets [18]. However, constant-time is sensitive to the microarchitectural implementation [19]: it may deem code insecure even when the underlying hardware sanitizes the relevant state, imposing overly conservative requirements. We illustrate the problem with a concrete example that serves as the primary running example throughout this chapter.

Example 4.1 (Load Buffer Side Channel). Consider the Sodor 5-stage RISC-V pipeline [20], augmented with a hypothetical *load buffer* (LB) for our analysis (Figure 4.2). The load buffer caches recent load results as $\langle addr, data, valid \rangle$ tuples, allowing subsequent loads to matching addresses to bypass memory and read cached data directly.

Now consider two implementations of the store instruction that differ only in when the load buffer is invalidated. These are illustrated in Figure 4.1. In the execute-stage variant (sw_{Exe}), the store flushes the load buffer during the execute stage; in the memory-stage variant (sw_{Mem}), the flush occurs at the memory stage. Both produce identical ISA-level

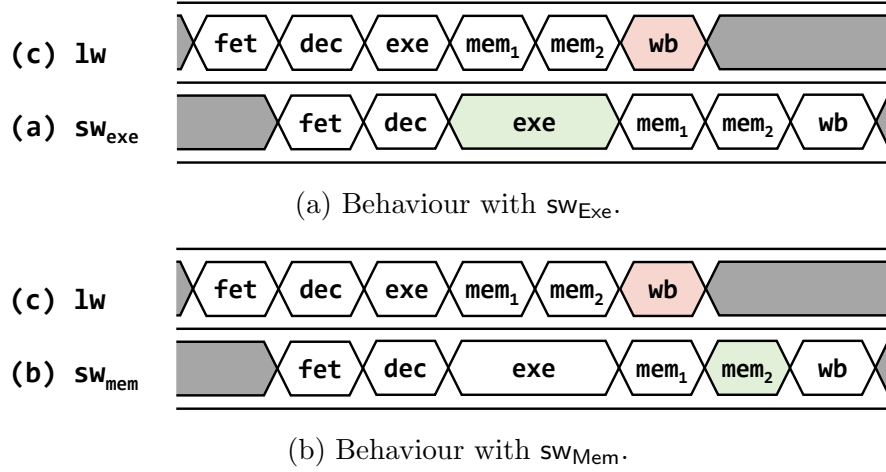


Figure 4.3: Different microarchitectural behaviours (with a 1-cycle memory latency) for two store implementations. With sw_{Exe} , the load buffer retains its entry after the load-store sequence; with sw_{Mem} , the store flushes the buffer after the refill.

results for all programs. But they differ in security, as shown by the timing waveforms in Figure 4.3: in sw_{Exe} , a subsequent speculative load may refill the buffer *after* the flush, leaving secret-dependent data observable via a timing side channel. In sw_{Mem} , the flush occurs after the refill, leaving the buffer clean.

An ISA-level model, which does not distinguish pipeline stages, cannot represent this difference at all, yet it is precisely this difference that determines whether the implementation is vulnerable.

More broadly, ISA models treat instruction sequences as compositional: the effect of executing $\llbracket i_1; i_2 \rrbracket$ is modeled as the sequential composition of executing $\llbracket i_1 \rrbracket$ and $\llbracket i_2 \rrbracket$ in isolation. This compositional assumption breaks for microarchitectural state, where the effect of two overlapping instructions depends on their relative pipeline positions. In the load buffer example, the semantics of both sw_{Exe} and sw_{Mem} are identical when executed in isolation, even with respect to the load buffer state, since both flush the buffer before finishing. The difference emerges only when a load overlaps with the store in the pipeline.

Full RTL models capture all implementation detail and can express any microarchitectural behavior. However, their state spaces grow exponentially with the number of microarchitectural resources (Challenge 3.1), making direct verification intractable for realistic designs.

Axiomatic ordering models such as $\mu spec$ (Section 3.3.4) occupy a middle ground for memory consistency verification: they express ordering constraints between microarchitectural events declaratively, without committing to a specific implementation. Since these constraints range over full executions, axiomatic models do not suffer from the instruction-isolation problem. However, they abstract away *functional behavior*, the actual values com-

puted and stored by the hardware. This is adequate for memory ordering properties but insufficient for security properties like non-interference (Definition 3.1), which require reasoning about whether secret values influence observable outputs. For example, processors can leak victim secrets through the LRU replacement state of shared caches [131]. Detecting such leaks requires a model that tracks cache contents and replacement policy at the functional level, detail that axiomatic models deliberately omit. While non-interference uniformly captures all vulnerable cases as a single property, μ spec-based security analyses [118, 88] work around this limitation by manually enumerating vulnerability patterns (exploit patterns, transmitters, disclosure gadgets). This requires deep microarchitectural expertise and risks false negatives if patterns are missed.

A second limitation is that axiomatic models are *declarative*: they define valid behaviors through constraints rather than state transitions. Checking alignment between such a model and the source RTL requires *global invariants* over full executions, which are difficult to generate and verify automatically [62, 84]. An *operational* model, by contrast, represents executions as imperative transitions on state at every cycle, enabling per-step equivalence checking with standard model checkers.

What is needed is a model that combines three properties: *finer temporal resolution* than ISA models, capturing per-cycle microarchitectural events rather than instruction-level snapshots; *functional detail* sufficient for security properties, tracking values and not just orderings; and an *operational* representation that enables direct equivalence checking against the source RTL. Micro-update models are designed to satisfy all three.

4.2 The Micro-Update Model

We now develop the micro-update model formalism. A micro-update model is defined over a set of signals and has two kinds of elements: *micro-updates*, which are imperative code snippets defining functional operations on those signals, and *guards*, which are Boolean predicates controlling when each micro-update is triggered. We illustrate the formalism through Figures 4.4–4.7: the processor architecture (Figure 4.4), the micro-update code blocks (Figure 4.5), the guard table (Figure 4.6), and example M-sets for different instructions (Figure 4.7).

Example 4.2 (Micro-update model walkthrough). Consider an `addi` instruction executing on the processor in Figure 4.4. The instruction interacts with several pipeline components by invoking multiple micro-updates. Each micro-update is executed when its corresponding guard (Figure 4.6) evaluates to true. For instance, `decode_itype` fires when `instr_queue[1]` is a load or ALU-immediate instruction.

While `instr_queue[1]` depends on a single instruction input, guards can in general depend on multiple inputs. If `rs1_addr(instr_queue[1]) == rd_addr(instr_queue[2])`, there is a data dependency between the current and previous instructions. In this case, the micro-update `alu_compute_alu_out_bypass_imm` is invoked instead of `alu_compute_rs_imm`,

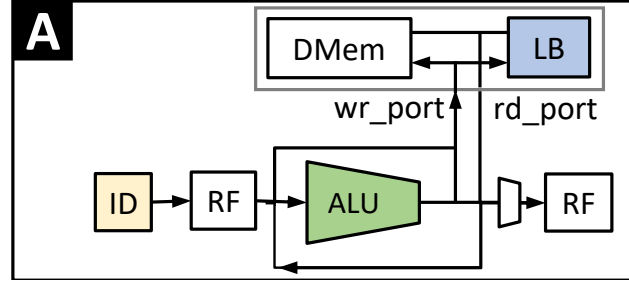


Figure 4.4: Overview (A): Processor schematic for the Sodor 5-stage core with load buffer and bypassing paths (grayed).

B <pre> mop decode_itype(): imm <- inst[31:20] rs1_addr <- inst[19:15] ... mop decode_stype(): imm <- inst[31:25] ++ inst[11:7] rs1_addr <- inst[19:15] </pre>	<pre> mop alu_compute_rs_imm(): alu_out <- aluop(rs1_data, imm) mop alu_compute_alu_out_bypass_imm(): alu_out <- aluop(alu_out, imm) mop alu_compute_rs_rs(): alu_out <- aluop(rs1_data, rs2_data) </pre>	<pre> mop lb_flush(): lb_addr <- 0 lb_data <- 0 lb_valid <- 1 mop lb_refill(): lb_addr <- mem_addr lb_data <- mem_data lb_valid <- 1 </pre>
---	--	--

Figure 4.5: Overview (B): Micro-update code blocks for decode, ALU, and load buffer operations.

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

Figure 4.6: Overview (C): Guard table mapping micro-updates to Boolean trigger conditions over pipeline state.

which reads from the register file. Figure 4.7 shows the resulting M-sets: the first column lists the micro-updates for `addi` without a dependency, and the second column shows how the M-set changes when a bypass dependency is present. The third column shows a `sw` instruction, which invokes a different set of micro-updates including `lb_flush` and `mem_write_req`.

The micro-update model addresses all three concerns identified in Section 4.1:

Finer temporal resolution. ISA-based approaches define semantics at instruction-level temporal resolution, between fetch and commit points. The micro-update model decomposes

D	addi w/o dependency	addi w/ alu_out dep.	sw without dep
	decode_itype	decode_itype	decode_stype
	regs_read	alu_compute_alu_out_bypass_imm	regs_read
	alu_compute_rs_imm	regs_write	alu_compute_rs_imm
	regs_write	...	lb_flush
			mem_write_req

Figure 4.7: Overview (D): Example M-sets for three instructions: **addi** without dependency, **addi** with ALU-output bypass, and **sw** without dependency.

the execution of each instruction into several micro-updates, evaluating guards and applying updates on a per-cycle basis. This finer resolution allows the model to discriminate between microarchitectural implementations that are architecturally indistinguishable, such as the SW_{Exe} and SW_{Mem} implementations from Example 4.1.

Imperatively modelled functional detail. Axiomatic ordering models such as $\mu spec$ can also provide finer temporal resolution through microarchitectural events, but they lack functional detail. Micro-update bodies (Figure 4.5) capture the actual values computed and stored, enabling specification and verification of security properties like non-interference [53, 116] that require reasoning about data values rather than just orderings.

Operational modeling. Because micro-updates describe imperative transitions on model state, the model supports per-cycle equivalence checking against the source RTL using standard model checkers. This contrasts with axiomatic approaches, where checking alignment with the RTL requires global invariants that are difficult to generate and verify automatically [62, 84].

We now formalize these concepts.

4.2.1 Signals of Interest

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

The S_{data} capture designer-identified components that are security-relevant based on the threat model. While automating this selection is an open challenge (discussed in Section 4.5), the key observation is that security properties have *locality*, i.e., they depend on specific microarchitectural components, not the entire design.

Example 4.3 (Running Example). For the load buffer (Example 4.1), S_{data} includes the load buffer fields (address, data, valid), pipeline control signals (stall, kill, bypass enable), and memory interface signals, roughly 20 signals out of the Sodor core’s several hundred.

4.2.2 Micro-Updates

A micro-update is an imperative code block that typically operates on small subsets of signals (Fig. 4.5). The model as a whole comprises a set of micro-updates, L . At each cycle, signals from S_{data} are updated based on the micro-updates from L that are invoked during that cycle. The body of each micro-update $t \in L$ provides the semantics $\llbracket t \rrbracket$, which defines how t transforms the state when invoked. An instruction typically performs several of these micro-updates during its lifetime, as Fig. 4.7 illustrates.

Micro-updates come in two types: *sequential* micro-updates consume values from the previous-cycle state, while *combinational* micro-updates use current-cycle values after dependencies have stabilized (analogous to sequential and combinational logic in HDLs). The sets of signals read and written by a micro-update, $Uses(u)$ and $Modifies(u)$, can be extracted from the code body (e.g., by AST traversal).

Example 4.4 (Running Example). For `lb_refill` in Figure 4.5, $Uses = \{\text{mem_addr}, \text{mem_data}\}$ and $Modifies = \{\text{lb_addr}, \text{lb_data}, \text{lb_valid}\}$. These derived sets are used to optimize synthesis (Section 4.3.2) and to define valid M-sets (Definition 4.1).

4.2.3 Trigger signals and guards

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

Example 4.5 (Running Example). Figure 4.6 shows representative guards. The guard for `lb_flush` is true when `instr_queue[2]` is a store instruction; the guard for `lb_refill` requires both a valid memory response and the absence of a flush. The guard for flushing depends on pipeline position, and as Example 4.1 showed, different implementation choices (flush at execute vs. memory stage) correspond to different guards with different security implications.

4.2.4 Execution of the micro-update model

States of the micro-update model transition system are assignments $\sigma : S_{data} \rightarrow D$. We write Σ for the set of all such assignments, $\Sigma = S_{data} \rightarrow D$. For initial state σ^{init} and input sequence $\vec{x}$, the execution of the model is the sequence of states at each cycle, denoted $\llbracket \vec{x} \rrbracket_M(\sigma^{\text{init}}) \in \Sigma^*$. Similarly, $\llbracket \vec{x} \rrbracket_D(\sigma^{\text{init}})$ denotes an execution of the source RTL design.

While each micro-update only modifies a few signals, more than one micro-update can be triggered in a single cycle. We call such a set of simultaneously triggered micro-updates an *M-set*.

Definition 4.1 (M-Set). An *M-set* is a set of micro-updates $M = \{u_1, \dots, u_k\}$ that fire simultaneously in one cycle. A valid M-set must satisfy:

1. **Disjoint modification:** $\forall u_i, u_j \in M, i \neq j : \text{Modifies}(u_i) \cap \text{Modifies}(u_j) = \emptyset$
2. **No combinational cycles:** The dependency graph among micro-updates in M is acyclic, where u' depends on u iff u' is combinational and $\text{Uses}(u') \cap \text{Modifies}(u) \neq \emptyset$

Both conditions can be checked from the uses and modifies information. The effect of triggering an M-set follows the usual sequential and combinational semantics.

Figure 4.7 illustrates M-sets for three instructions. An `addi` without dependencies triggers `{decode_itype, regs_read, alu_compute_rs_imm, regs_write}`. The same `addi` with an ALU-output dependency instead triggers `alu_compute_alu_out_bypass_imm`, bypassing the register file read. A `sw` instruction triggers a different set entirely, including `decode_stype`, `lb_flush`, and `mem_write_req`.

Definition 4.2 (Micro-Update Model). A *micro-update model* $\mathcal{M}$ over signals of interest S_{data} consists of:

- A library of micro-updates $\mathcal{U} = \{u_1, \dots, u_n\}$
- Guards $G = \{g_1, \dots, g_n\}$ controlling each micro-update
- Trigger signals S_{trig}

At each cycle, the model evaluates all guards under the current trigger signal values, collects the triggered micro-updates into an M-set, and applies them to update the state of S_{data} .

4.2.5 S_{data} -Equivalence and Soundness

The correctness criterion for MUMs is *S_{data} -equivalence*: the model must reproduce the RTL's behavior on all signals of interest, at every cycle, under all inputs.

Definition 4.3 (S_{data} -Equivalence). A MUM $\mathcal{M}$ is S_{data} -equivalent to hardware design H iff for all input sequences $\vec{x} \in I^*$ and all initial states agreeing on S_{data} :

$$\forall j \geq 0. \mathcal{M}(\vec{x}, s_0^{\mathcal{M}})(j) \equiv_{S_{data}} H(\vec{x}, s_0^H)(j)$$

where $\sigma_1 \equiv_{S_{data}} \sigma_2$ iff $\forall s \in S_{data} : \sigma_1(s) = \sigma_2(s)$.

Theorem 4.1 (Soundness of MUM Verification). *If $\mathcal{M}$ is S_{data} -equivalent to H and $Pub, Obs \subseteq S_{data}$, then:*

$$\mathcal{M} \models SNI(Pub, Obs) \implies H \models SNI(Pub, Obs)$$

More generally, any hyperproperty φ over S_{data} that holds on $\mathcal{M}$ also holds on H .

Proof sketch. Since $\mathcal{M}$ and H produce identical traces when projected onto S_{data} , any pair of traces witnessing a violation of φ on H would also witness a violation on $\mathcal{M}$. Contrapositive gives the result. $\square$

S_{data} -equivalence is stronger than simulation: it requires *exact agreement* on signals of interest at every cycle, not merely inclusion of trace sets. This per-cycle guarantee enables hyperproperty preservation, a property that simulation alone does not ensure.

4.3 Synthesizing MUMs from RTL

Manually constructing MUMs is possible but tedious—the micro-update bodies are relatively straightforward to write, but determining the precise conditions under which each fires requires a deep understanding of inter-dependencies between pipeline stages. Indeed, roughly 10 out of 42 RTL bug reports in the CVA6 processor [29] were due to imprecisely coordinated inter-dependencies of this kind.

We present a synthesis methodology that automates guard generation and validates S_{data} -equivalence. The key insight is a *functional-temporal decomposition*: the user provides the functional components (micro-update bodies, which require only local understanding of individual pipeline stages), and synthesis determines the temporal coordination (guards, which require global understanding of inter-dependencies). The approach operates in two phases: M-set generation determines *which* micro-updates fire at each cycle, and guard synthesis determines *when* they fire.

The methodology is implemented in PAUL (Python-based Atomic-Update Language), interfacing with Icarus Verilog for simulation, CVC5 [5] for syntax-guided synthesis, and SymbiYosys [134] for bounded model checking. Figure 4.8 illustrates the end-to-end pipeline, showing how these tools coordinate across the two synthesis phases and the CEGIS-style verification loop.

4.3.1 The Micro-Update Library

The micro-update library $\mathcal{U}$ used when generating a model for signals S_{data} should capture all possible single-cycle transitions of those signals. As such, these transitions depend on the design: a processor with bypassing needs micro-updates that allow the ALU output to depend on pipeline registers further along the pipeline, while one without bypassing does not.

However, the library need not be specific to a single design. Multiple designs from the same family can share micro-updates: in our experiments, the Sodor family of processors

share micro-updates for instruction decode, register file operations, and functional units, while more deeply pipelined processors require additional micro-updates for intermediate stages.

The synthesis procedure is robust to imprecise libraries. Irrelevant micro-updates (e.g., LB micro-updates in a model with only ALU operations) are filtered out during candidate M-set generation: their guard is assigned `false`. Since this phase does not invoke costly solvers, synthesis performance is not heavily affected. Incomplete libraries, on the other hand, fail during M-set generation, since no combination of available micro-updates can justify the observed simulation trace. The procedure then identifies the failing step and signals, guiding the user to add the missing micro-updates. A larger-than-necessary library does not affect correctness but may increase synthesis runtime due to the larger search space.

4.3.2 Phase 1: M-Set Generation

Given the micro-update library $\mathcal{U}$ (user-specified) and simulation traces from the RTL design, Phase 1 identifies which M-sets are triggered at each simulation step.

The first step is *trace generation*: random test programs are executed on the RTL simulator, recording the values of all signals in S_{data} at each cycle to produce a set of pre/post state pairs $\{(\sigma_i, \sigma_{i+1})\}$.

The second step is *candidate M-set extraction*. For each state pair (σ_i, σ_{i+1}) , a depth-first search (Algorithm 1) finds sets of micro-updates that transform σ_i into σ_{i+1} . The search exploits two structural properties for pruning: independent micro-updates (those with disjoint Uses and Modifies sets) can be searched in any order, and once a micro-update is chosen for a signal, all other micro-updates modifying that signal are excluded.

The third step *eliminates spurious candidates*. The DFS may produce multiple candidate M-sets consistent with a given state pair, as the following example illustrates.

Example 4.6 (Spurious M-sets). Consider the simulation trace in Figure 4.9. For the third transition (counter 2→3), the load buffer signals do not change between consecutive steps (`lb_data`, `lb_addr`, `lb_valid` remain constant). Both `lb_refill` (which writes the current memory values to the LB) and `lb_hold` (which does nothing) correctly transform the pre-state into the post-state, because the memory values happen to match the existing LB contents. The DFS produces both as candidates. However, in the actual design, `lb_refill` is only triggered when the memory-stage instruction is a load, which is not the case here. Thus `lb_refill` is a spurious candidate that must be filtered out.

We eliminate spurious candidates using two techniques.

Distinguishing oracles. Adapting the oracle-guided synthesis technique [64], given two candidate M-sets M_1, M_2 , we create two copies of the micro-update model, one invoking M_1 and the other M_2 . Let $Uses = \bigcup \{Uses(u) \mid u \in M_1 \cup M_2\}$ and $Mods = (\bigcup \{Modifies(u) \mid u \in M_1\}) \cap (\bigcup \{Modifies(u) \mid u \in M_2\})$. We invoke the hardware model checker to find a trace

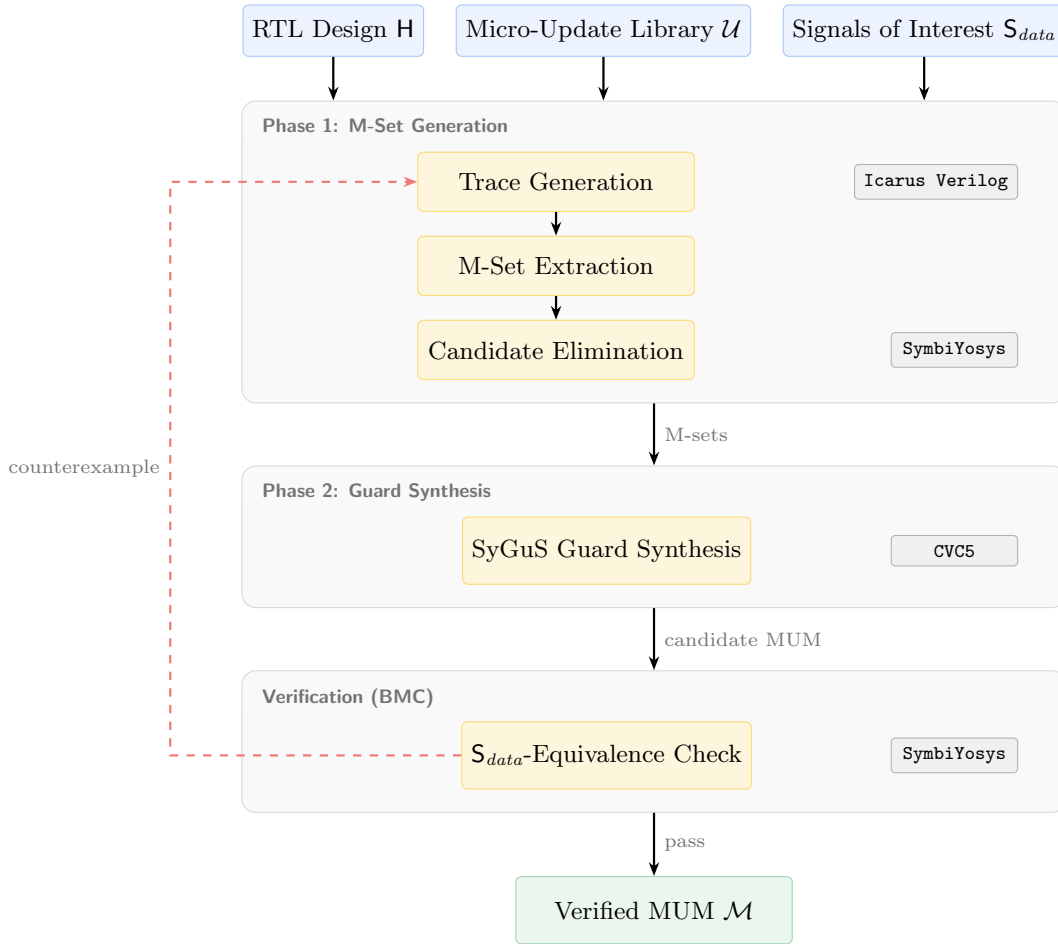


Figure 4.8: MUM synthesis pipeline. Phase 1 generates candidate M-sets from simulation traces and eliminates spurious candidates via distinguishing oracles and cover properties. Phase 2 synthesizes guards via SyGuS. The candidate MUM is verified for S_{data} -equivalence; failures feed back in a CEGIS loop.

where, at some cycle i , all used signals agree but some modified signal differs:

$$\text{assume} \left(\bigwedge_{s \in \text{Uses}} \sigma_1^i(s) = \sigma_2^i(s) \right) \quad \text{assert} \left(\bigvee_{s \in \text{Mods}} \sigma_1^{i+1}(s) \neq \sigma_2^{i+1}(s) \right)$$

Since M_1 and M_2 produce different values for some signal, only one can be the valid candidate for that transition. Continuing Example 4.6: the distinguishing oracle generates a trace where the memory-stage instruction is not a load. In this trace, only `lb_hold` applies (the LB entry remains unchanged), eliminating the spurious `lb_refill` candidate. A typical case

Algorithm 1 Candidate M-set generation via DFS

Require: Pre-state σ , post-state σ' , library $\mathcal{U}$ **Ensure:** Set of candidate M-sets consistent with (σ, σ')

```

1: function GENMSETS( $\sigma, \sigma', \mathcal{U}$ )
2:   candidates  $\leftarrow \emptyset$ 
3:   DFS( $\emptyset, \sigma$ )
4:   return candidates
5: end function
6: function DFS( $M_{\text{curr}}, \sigma_{\text{curr}}$ )
7:   if  $\sigma_{\text{curr}} = \sigma'$  on all signals in  $\mathbf{S}_{\text{data}}$  then
8:     candidates  $\leftarrow$  candidates  $\cup \{M_{\text{curr}}\}$ 
9:     return
10:  end if
11:  for  $u \in \mathcal{U}$  not conflicting with  $M_{\text{curr}}$  do
12:     $\sigma_{\text{next}} \leftarrow \sigma_{\text{curr}}$  with  $u$  applied
13:    if  $\sigma_{\text{next}}$  agrees with  $\sigma'$  on  $\text{Modifies}(u)$  then
14:      DFS( $M_{\text{curr}} \cup \{u\}, \sigma_{\text{next}}$ )
15:    end if
16:  end for
17: end function

```

where the distinguisher helps is when a functional unit operates on one of several possible inputs (e.g., bypassing versus register-file read).

Cover properties. Some execution patterns arise too rarely in random simulation to appear in traces. We specify conditions of interest as cover properties of the form $\text{cover}(\phi)$ for the model checker, which generates targeted traces where ϕ holds.

Example 4.7 (Using cover properties). Consider a bypassing path micro-update triggered when there is a data dependency between the execute-stage and fetch-stage instructions and the execute-stage destination is not x0: $\phi_{\text{bypass}} \equiv (\text{rd}(i_{\text{exe}}) = \text{rs1}(i_{\text{fet}})) \wedge (\text{rd}(i_{\text{exe}}) \neq 5'd0)$. Since this condition requires matching register fields, it holds with probability $\approx 1/32$ and may be missed during random simulation. The cover property $\text{cover}(\phi_{\text{bypass}})$ generates an execution exercising this path, which is then added to the trace corpus for future M-set generation.

4.3.3 Phase 2: Guard Synthesis

With M-sets resolved, Phase 2 synthesizes Boolean predicates (guards) that correctly trigger each micro-update. This is formulated as a Syntax-Guided Synthesis (SyGuS) problem.

The inputs are M-sets $M^1, \dots, M^k$ from Phase 1, trigger signal assignments σ_{trig}^i at each simulation step, and a Boolean indicator b_u^i recording whether micro-update u appeared in

counter	0	1	2	3
inst_exe	addi r11,r10,16	lb r5,r4(16)	addi r13,r12,16	...
inst_mem	xori r3,r2,16	addi r11,r10,16	lb r5,r4(16)	addi r13,r12,16
lb_data	10	10	30	30
lb_addr	20	20	40	40
lb_valid	0	0	1	1
mem_data	60	30	30	80
mem_addr	70	40	40	90

Figure 4.9: A sample execution trace from simulation, showing signal values for the load buffer and memory port across four cycles. At the third transition (counter 2→3), the load buffer is refilled (lb_data 10 → 30, lb_valid 0 → 1) as the lb instruction reaches the write-back stage.

the valid M-set at step i . The synthesized guards must satisfy two constraints:

1. *Negative examples:* If micro-update u did not fire at step i , its guard must evaluate false:

$$\bigwedge_i (b_u^i \vee \neg g_u(\sigma_{\text{trig}}^i))$$

2. *Exactly-one per signal:* For each signal $s \in \mathcal{S}_{\text{data}}$, exactly one micro-update modifying s fires at any given step:

$$\bigwedge_{u \neq u' \in \text{Modifying}(s)} (\neg g_u \vee \neg g_{u'}) \wedge \bigvee_{u \in \text{Modifying}(s)} g_u$$

The SyGuS solver (CVC5) searches over a context-free grammar restricting the syntax of guards to Boolean combinations of trigger signal predicates. Algorithm 2 summarizes the procedure.

4.3.4 Equivalence Verification

The synthesized MUM is verified against the RTL for $\mathcal{S}_{\text{data}}$ -equivalence using bounded model checking (SymbiYosys with Boolector/ABC backends). The BMC depth is chosen to cover all inter-instruction interactions—for a 5-stage pipeline, depth 15 suffices.

If the equivalence check fails, the counterexample trace is added to the simulation database and synthesis iterates. This CEGIS-style loop encourages convergence in practice: each iteration eliminates at least one incorrect candidate from the finite search space.

Algorithm 2 Guard synthesis via SyGuS

Require: M-sets $\{M^1, \dots, M^k\}$, trigger assignments $\{\sigma_{\text{trig}}^1, \dots, \sigma_{\text{trig}}^k\}$, grammar G
Ensure: Guards $\{g_u \mid u \in \mathcal{U}\}$ consistent with all M-sets

- 1: **for** each micro-update $u \in \mathcal{U}$ **do**
- 2: Collect positive examples $E^+ \leftarrow \{i \mid u \in M^i\}$
- 3: Collect negative examples $E^- \leftarrow \{i \mid u \notin M^i\}$
- 4: $\mathcal{C} \leftarrow \bigwedge_{i \in E^-} \neg g_u(\sigma_{\text{trig}}^i)$ ▷ Guard must be false on negatives
- 5: $\mathcal{C} \leftarrow \mathcal{C} \wedge$ mutual exclusion constraints ▷ One modifier per signal
- 6: $g_u \leftarrow \text{SyGuS}(G, \mathcal{C})$ ▷ Synthesize guard from grammar G
- 7: **end for**
- 8: **return** $\{g_u \mid u \in \mathcal{U}\}$

4.3.5 Hierarchical Synthesis

For complex designs, monolithic synthesis can be prohibitively expensive. The hierarchical approach reuses previously synthesized sub-component models.

To synthesize a model for component C with signals $\mathcal{S}_{\text{data}}$, given a previously synthesized model $\mathcal{M}_1$ for sub-component C_1 with signals $\mathcal{S}_{\text{data}1} \subseteq \mathcal{S}_{\text{data}}$:

1. Reuse $\mathcal{M}_1$ for signals in $\mathcal{S}_{\text{data}1}$
2. Synthesize guards only for new signals $\mathcal{S}_{\text{data}} \setminus \mathcal{S}_{\text{data}1}$

This is sound when C_1 's signals are not directly driven by logic outside C_1 and C_1 's trigger signals map straightforwardly to C 's trigger signals. The hierarchical approach mirrors the RTL module hierarchy, exploiting the same compositional structure that hardware designers use.

4.4 Results and Insights

We evaluate MUM synthesis and verification on two RISC-V processors: the Sodor 5-stage (in-order, RV32I) and CVA6/Ariane (6-stage, RV64). All experiments run on an Intel Core i7-10610U at 2.3 GHz with 16 GB RAM.

4.4.1 Model Generation

Table 4.1 summarizes model generation for the Sodor processor.

M-set generation takes under 5 seconds in all cases (no solver calls). Guard synthesis is the dominant cost, scaling with the number of micro-updates and grammar depth. Equivalence checking at depth 15 ranges from 10–34 minutes.

Table 4.2 shows results for the CVA6 processor.

Table 4.1: MUM synthesis for Sodor 5-stage processor.

Model Slice	$ S_{data} $	$ U $	Guard Synth	Equiv Check
I-type ALU	15	25	1m 1s	10m 9s
R-type ALU	15	29	2m 14s	11m 57s
ALU + load + store	20	37	57m	34m 24s

Table 4.2: MUM synthesis for CVA6 processor components.

Module	Guard Synth	Equiv Check	Notes
TLB	4s	6s	PLRU eviction
Write-buffer	2m 47s	1m 10s	Coalescing buffer
Store-unit	5m 17s	2m 58s	Speculative queue
Load-store-unit (mono.)	>1hr	1m 49s	Timeout
Load-store-unit (hier.)	11m 38s	1m 49s	Reuses store-unit

4.4.2 Security Analysis: Spectre Gadget Discrimination

We apply the synthesized MUM to analyze Spectre Bounds Check Bypass (BCB) gadgets [68] on the Sodor processor augmented with a load buffer. Figure 4.10(A) shows the Spectre BCB gadget. Figure 4.10(B) shows assembly fragments for the `array2` access in the gadget; Figure 4.10(C) shows the assembly after a minor modification to the gadget (changing `&=` to `=`). In (B), two ALU instructions separate the load instruction from the store instruction. In (C), they do not.

The accompanying waveforms in Figure 4.10 depict the pipeline timing of (B) and (C) on the Sodor processor with the load buffer, under the execute-stage flushing implementation (`swExe`). In (B), because two ALU instructions separate the load from the store, the store flushes the load buffer *after* the load refills it, leaving the buffer clean. In (C), with no such separation, the store flushes the buffer *before* the load refills it, leaving it tainted with secret-dependent data. Thus, if the load buffer address is attacker-observable, (B) is safe while (C) is unsafe.

This example illustrates how small microarchitectural additions like the load buffer can render hand-written abstract models imprecise. The abstract model from Cheang et al. [21], designed to check Spectre variants, would flag *both* (B) and (C) as vulnerabilities, producing a false positive. Approaches such as Spectector [58], InSpectre [56], and the automatic discovery framework of Fabian et al. [34] rely on similarly hand-written models that cannot capture the load buffer’s timing behavior. The synthesized MUM, however, correctly distinguishes the two cases by capturing the relative pipeline positions of the load and store, which determine whether the flush precedes or follows the refill.

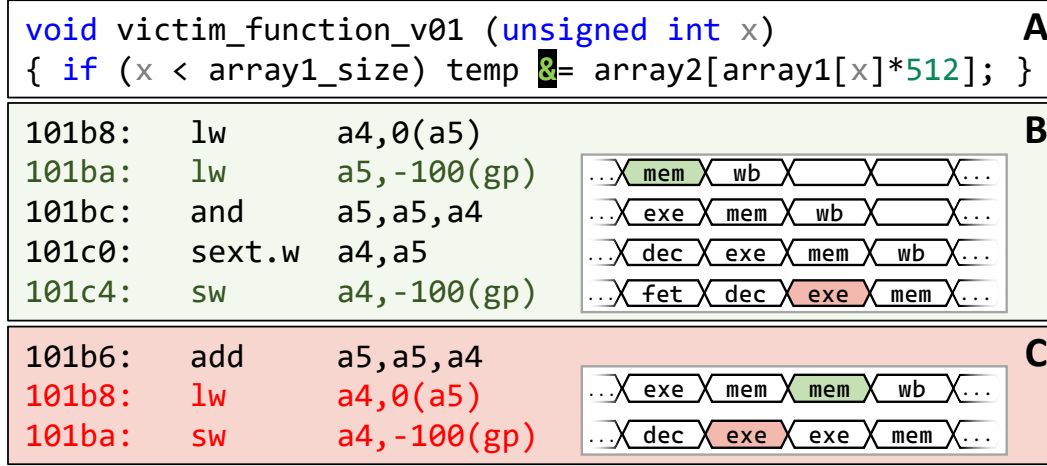


Figure 4.10: Subtle program changes can expose vulnerabilities. (A) The Spectre BCB gadget from Kocher et al. [68]. (B, C) Compiled assembly from the original example and after changing `&=` to `=`. The accompanying waveforms show how (B) leaves the load buffer clean under `swExe` while (C) does not.

Table 4.3: Information-flow analysis: MUM vs. RTL verification time on Sodor.

Sequence	Source	Result	RTL	MUM	Speedup
nop;load	mem	Safe	2m 43s	8s	20×
nop;load	mem,regs	Unsafe	3m 50s	11s	21×
nop;alu;load	mem	Safe	2m 37s	12s	13×
nop;store;load	mem	Safe	2m 30s	9s	17×
nop;load;load	regs	Unsafe	4m 15s	8m 51s	0.5×
nop;(load+alu) ² ;store	mem,regs	Safe	4m 49s	38s	8×
nop;(load+alu) ³ ;store	mem,regs	Safe	5m 11s	1m 31s	3.4×

4.4.3 Security Analysis: Information Flow

Beyond gadget discrimination, MUMs enable systematic information-flow analysis. We check whether information flows from specified source signals to the load buffer address (`lb_addr`), assumed adversary-observable, across a range of instruction sequences. Table 4.3 summarizes results for the Sodor processor.

The `nop;load;load` row is notable: the MUM is *slower* than RTL (8m 51s vs. 4m 15s). This occurs because two consecutive loads create complex interactions between the load buffer and forwarding logic, producing a large number of candidate M-sets that the DFS-based extraction must enumerate. The slowdown illustrates that MUM abstraction is not uniformly beneficial; it is most effective when the instruction sequence exercises diverse

Table 4.4: LSU timing channel analysis on CVA6: MUM vs. RTL.

Sequence	Constraint	Result	RTL	MUM	Speedup
ld;st;ld;alu	none	Unsafe	34s	17s	2×
ld;st;ld;alu	regs equal	Safe	3m 57s	48s	5×
ld ₁ ;st;ld ₂ ;alu	none	Unsafe	1m 29s	25s	3.6×
ld ₁ ;st;ld ₂ ;ld ₃	$rd_1 \neq rs1(st, ld_2)$	Unsafe	9m 4s	1m 2s	8.7×
ld ₁ ;st;ld ₂ ;ld ₃	$rd_1 \neq rs1(st, ld_2, ld_3)$	Safe	12m 36s	1m 29s	8.5×

pipeline stages (loads, stores, ALU) rather than stressing a single component.

4.4.4 Security Analysis: CVA6 Timing Channels

The CVA6 processor presents a more challenging target: its load-store unit, which can complete loads and stores out of program order, introduces timing channels through stalls that occur when a load address matches a pending store address. Table 4.4 presents selected results, illustrating how register constraints affect the security verdict. We use **ld/st** as shorthand for RISC-V load/store instructions.

4.4.5 Key Insights

Three insights emerge from the evaluation. First, MUMs achieve substantial verification speedup—2–21× across both processors—by eliminating irrelevant state while remaining equally faithful on S_{data} for the security property of interest. Second, synthesis cost is front-loaded: model generation is a one-time cost (minutes to an hour), and the resulting MUM is reused across multiple security queries, so the amortized cost is strongly favorable. Third, compositionality is essential for scaling. Hierarchical synthesis reduces CVA6 LSU synthesis from timeout (>1 hour) to 11 minutes by reusing the store-unit model, mirroring the modular structure that hardware designers already use.

4.5 Discussion

Two aspects of MUM construction currently require human input: identifying the signals of interest S_{data} and specifying the micro-update library $\mathcal{U}$. In practice, identifying S_{data} took 3–4 person-weeks for CVA6 (dominated by understanding and instrumenting the RTL), while specifying the micro-update library took 1–2 person-days. Guard synthesis and equivalence checking are fully automated.

The key to tractability is the *functional-temporal decomposition*. The MUM formalism separates the abstraction into functional components (micro-updates, which are relatively

straightforward to specify) and temporal components (guards, which capture when each operation fires). The functional part leverages domain expertise; the temporal part is synthesized automatically. Synthesizing both aspects simultaneously would be far harder.

The approach has several limitations. The current formulation assumes a finite, user-specified micro-update library; if the library is incomplete, synthesis will fail. The BMC-based equivalence check provides a bounded guarantee; extending to unbounded verification would require inductive invariants over the MUM-RTL pair. And the signals-of-interest selection may miss security-relevant signals, potentially allowing attacks on components outside S_{data} .

MUM synthesis is a canonical instance of synthesis-enabled reasoning: the system (RTL) is too complex for direct verification, so SER synthesizes a simpler, equivalent model. The synthesized MUM is not merely an approximation— S_{data} -equivalence ensures soundness. Domain structure (micro-update decomposition, module hierarchy) constrains the synthesis search, making it tractable.

Looking forward, PyCaliper (Chapter 9) provides infrastructure for expressing MUM specifications in Python and compiling to both verification and synthesis backends, including CEGIS and LLM-assisted approaches that offer complementary paths to automating the remaining manual steps.

4.6 Related Work

ISA-level models. ISA-level verification approaches [17, 111, 122, 54, 23, 4] and instruction-level abstractions [63, 137] capture only architectural state, making them compact but unable to express microarchitectural security properties. MUMs fill this gap by providing finer temporal resolution while remaining tractable.

Axiomatic microarchitectural models. The μ spec framework [78, 80] and tools in the Check suite [82, 119, 84] express microarchitectural orderings axiomatically but abstract away functional behavior. MUMs complement these by capturing the actual values computed and stored, enabling non-interference analysis. RTL2 μ spec [62] synthesizes axiomatic models from RTL; MUM synthesis addresses a different goal (functional abstraction for security analysis rather than ordering extraction).

Hardware security verification. UPEC [36, 35] verifies RTL directly against security properties using unique program execution checking, and constant-time hardware verifiers such as IODINE and Xenon [44] use solver-aided techniques. Approaches such as [21, 58, 56] verify software against manually written platform models; Buiras et al. [16] present a complementary methodology for validating such platform models. MUMs provide a more precise alternative to hand-written models: the S_{data} -equivalence guarantee ensures that security proofs on the MUM transfer soundly to the RTL.

Hardware mitigations and side-channel-resistant compilation. Hardware mitigations such as Speculative Taint Tracking [135] prevent secret values from being forwarded to transmitting instructions, while compiler-level approaches such as Blade [121] eliminate speculative leaks at the type-system level. MUMs are complementary: they provide a verification substrate on which these mitigations can be analyzed.

Rule- and transaction-based hardware modeling. Micro-updates bear a conceptual resemblance to rules in Bluespec [93]: both decompose hardware behavior into small, locally-specified units guarded by predicates. Our focus, however, is on *lifting* an existing RTL design into such a decomposition for verification, rather than on synthesizing RTL from rules.

Program slicing and lifting. MUM synthesis bears a conceptual resemblance to program slicing [128]: both extract a property-relevant subset of a larger system. The lifting methodology also connects to work in the PL and systems domains [65, 110], which extract high-level representations from low-level implementations while maintaining equivalence. MUMs extend this idea to hardware, with the functional-temporal decomposition tailored to the structure of RTL designs.

Chapter 5

Modelling and Verifying Isolation-Based Mitigations

The material in this chapter is based on our FMCAD 2023 paper [49]. Code and reproducibility artifacts are available at https://github.com/lichye/sec_resource_partitioning.

MUMs address the implementation-to-model gap for *functional* microarchitectural behavior: how signals evolve cycle by cycle. A complementary challenge arises for *isolation mechanisms*: cache partitioning, page coloring, and predictor isolation all aim to prevent cross-domain information leakage, but the full hardware state is typically too large for direct verification. This chapter introduces an abstract resource model and *conditional equality invariants* that make non-interference verification tractable for partitioned resources. We ground the development in a concrete example.

Example 5.1 (Prime+Probe on a Partitioned Cache). Consider a cache segment with four lines $\{l_0, l_1, l_2, l_3\}$ shared between a victim and an attacker (Figure 5.1). The attacker first *primes* the cache by loading a value into line l_2 . The victim then performs a memory access that touches one of l_0, l_1 , or l_2 , producing cases A, B, and C respectively. When the attacker subsequently *probes* l_2 , case C yields a miss (the victim evicted the attacker’s data) while A and B yield hits. This timing difference reveals which line the victim accessed.

Cache partitioning prevents this by allocating disjoint partitions: l_0, l_1 to the victim and l_2 to the attacker. Since the victim cannot access l_2 , case C is eliminated, and both remaining cases (A, B) produce the same hit outcome on the attacker’s probe.

However, the replacement policy state (the cloud in Figure 5.1) may still differ between A and B, even though the attacker-visible partition is identical. The victim’s access to l_0 versus l_1 may update different PLRU tree bits, and this difference could manifest after *some number* of further attacker accesses. Proving security therefore requires reasoning not just about partition contents but about the policy state as well.

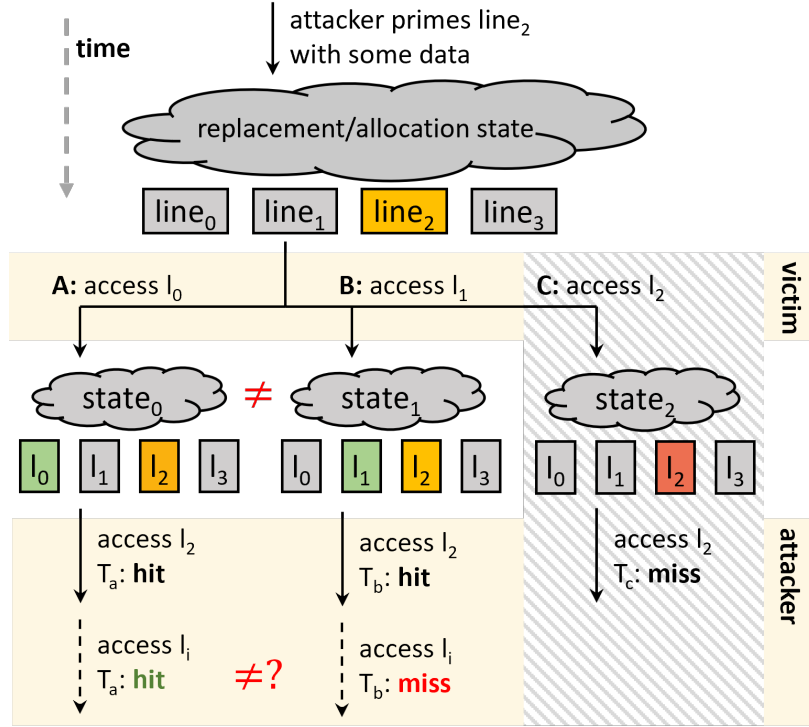


Figure 5.1: Prime+Probe attack on a partitioned cache. The cache has four lines; l_0, l_1 are allocated to the victim, l_2 to the attacker, l_3 is unallocated. The attacker primes l_2 , then the victim accesses one of l_0 (case A), l_1 (case B), or l_2 (case C). On a partitioned cache, case C is blocked, so the attacker's probe of l_2 yields a hit in both A and B, leaking no information. However, the replacement/allocation policy state (cloud) may still differ between A and B, potentially leaking information through subsequent attacker accesses.

5.1 Abstract Resource Model

We capture the essential structure of partitioned resources through an abstract model. A *resource* is a collection of *cells*, each representing one unit of the shared resource (e.g., a cache line). Cells are indexed by a set $\mathbb{I}$ and hold values from a set $\mathbb{V}$ (with $\perp \in \mathbb{V}$ representing empty). A set of *protection domains* $\mathbb{D}$ share the resource. In Figure 5.1, the victim and attacker processes are domains.

Each cell is allocated to a domain via an *allocation map* $a : \mathbb{I} \rightarrow \mathbb{D}$. We write $a \downarrow_d = \{i \in \mathbb{I} \mid a(i) = d\}$ for the set of cells allocated to domain d . The resource contents are given by $r : \mathbb{I} \rightarrow \mathbb{V}$, and a *policy state* p (composed of individual *policy elements* $E \rightarrow \mathbb{V}_p$) governs replacement and eviction.

Three functions define the policy behavior:

$$\begin{aligned} f_{\text{evict}} &: \mathbb{A} \times \mathbb{P} \times \mathbb{D} \rightarrow \mathbb{I} \\ f_{\text{updM}} &: \mathbb{A} \times \mathbb{P} \times \mathbb{D} \times \mathbb{I} \rightarrow \mathbb{P} \\ f_{\text{updH}} &: \mathbb{A} \times \mathbb{P} \times \mathbb{D} \times \mathbb{I} \rightarrow \mathbb{P} \end{aligned}$$

where $\mathbb{A} = \mathbb{I} \rightarrow \mathbb{D}$ is the set of allocations and $\mathbb{P}$ is the set of policy states. The function f_{evict} , given the current allocation, policy state, and accessing domain, identifies the cell chosen for replacement. The functions f_{updM} and f_{updH} produce the new policy state after a miss or hit, respectively.

The overall configuration is a tuple $c = \langle a, r, p \rangle$. Initial configurations have the form $\langle a, r_{\text{init}}, p_{\text{init}} \rangle$ where a is arbitrary, the resource is empty ($r_{\text{init}} = \lambda i. \perp$), and p_{init} is the initial policy state.

5.1.1 Access Semantics

At each step, a domain $d \in \mathbb{D}$ performs an access with argument $\alpha \in \mathbb{V}$, denoted $d:\alpha$. The semantics are defined by the following pseudocode:

```

1  if (∃i. r(i) = α ∧ a(i) = d) :
2      // Hit: let i be the hitting index
3      p ← fupdH(a, p, d, i)
4  else:
5      // Miss
6      i ← fevict(a, p, d)
7      r(i) ← α
8      p ← fupdM(a, p, d, i)
    
```

Listing 5.1: Semantics of access $d:\alpha$ from configuration $c = \langle a, r, p \rangle$.

On a hit, only the policy state is updated. On a miss, the eviction function selects a cell, the cell contents are replaced, and the policy state is updated. We write $\delta(c, d:\alpha)$ for the resulting configuration.

5.1.2 Executions

A trace is a sequence of configurations $\pi = c_0 \cdot c_1 \cdots$ where c_0 is an initial configuration and at each step j , some access $d_j:\alpha_j$ is performed: $\delta(c_j, d_j:\alpha_j) = c_{j+1}$. We write $\text{op}_\pi[j]$ for the operation at step j and Π for the set of all traces.

5.2 Attacker Model and Security Property

We consider a timing-based attacker operating in domain $d^\#$ that can observe the hit/miss outcome of its own accesses. Two configurations c_1, c_2 are *indistinguishable* to the attacker

if every attacker access produces the same outcome in both:

$$\phi_{\text{indist}}(c_1, c_2) \equiv \forall \alpha. \text{isHit}(c_1, d^\# : \alpha) \iff \text{isHit}(c_2, d^\# : \alpha)$$

where $\text{isHit}(c, d^\# : \alpha) = \exists i. r(i) = \alpha \wedge a(i) = d^\#$.

Security requires that victim behavior never influences attacker observations. We formulate this as a non-interference property [53, 27]. Two traces π_1, π_2 have the same *public inputs* ($\pi_1 =_L \pi_2$) if the attacker allocation is identical ($a(\pi_1[0]) \downarrow_{d^\#} = a(\pi_2[0]) \downarrow_{d^\#}$) and the attacker accesses agree at every step. The non-interference property requires that for all such trace pairs, the attacker observations are identical at every step:

$$\forall \pi_1, \pi_2 \in \Pi. \pi_1 =_L \pi_2 \implies \forall j. \phi_{\text{indist}}(\pi_1[j], \pi_2[j])$$

5.3 Conditional Equality Invariants

Direct verification of non-interference requires reasoning about all pairs of traces, a hyper-property [27]. We reduce this to inductive verification using relational verification [6] and *conditional equality invariants*, which are relational invariants $\phi_{\text{inv}}(c_1, c_2)$ relating states from two traces.

A natural candidate is to require that the two configurations are fully equal, but this is too strong: victim domains may use the resource differently in the two traces, so cells allocated to victims *should* differ. Only attacker-allocated cells must agree. More subtly, the policy state need not be fully equal either. In Example 5.1, the victim’s access to l_0 versus l_1 produces different PLRU tree updates, yet the scheme may still be secure because these differences never propagate to attacker-observable outcomes.

The conditional equality invariant captures exactly this nuance. It enforces three conditions: the attacker allocation agrees ($a_1 \downarrow_{d^\#} = a_2 \downarrow_{d^\#}$), the contents of attacker-allocated cells agree ($\forall i \in a_1 \downarrow_{d^\#}. r_1(i) = r_2(i)$), and a *filtered subset* of the policy state agrees:

$$\phi_{\text{inv}}(c_1, c_2) \equiv \underbrace{a_1 \downarrow_{d^\#} = a_2 \downarrow_{d^\#}}_{\text{same attacker allocation}} \wedge \underbrace{\forall i \in a_1 \downarrow_{d^\#}. r_1(i) = r_2(i)}_{\text{same attacker contents}} \wedge \underbrace{\forall e \in \text{filter}(a \downarrow_{d^\#}). p_1(e) = p_2(e)}_{\text{conditional policy equality}}$$

The filtering function $\text{filter} : 2^\mathbb{I} \rightarrow 2^E$ selects which policy elements must agree based on the attacker allocation. For instance, in a PLRU cache, the PLRU tree bits corresponding to attacker-allocated ways must agree, but bits governing victim-only ways need not. Crucially, filter depends on $a \downarrow_{d^\#}$, which varies at runtime, so the invariant cannot be expressed as a static equality. This is why conditional equality invariants are more expressive than pure equality-based relational invariants.

5.4 Verification Conditions

To prove non-interference, it suffices to find a conditional equality invariant ϕ_{inv} that is inductive and implies indistinguishability. Concretely, ϕ_{inv} must hold for initial configurations

with identical attacker allocations (*base case*), be preserved when both traces perform the same attacker access (*induction, attacker steps*), be preserved when the two traces perform arbitrary non-attacker accesses (*induction, non-attacker steps*), and imply indistinguishability (*sufficiency*):

$$\begin{aligned}
a_1 \downarrow_{d^\#} = a_2 \downarrow_{d^\#} &\implies \phi_{\text{inv}}(\langle a_1, r_{\text{init}}, p_{\text{init}} \rangle, \langle a_2, r_{\text{init}}, p_{\text{init}} \rangle) && (\text{base}) \\
\forall \alpha. \phi_{\text{inv}}(c_1, c_2) &\implies \phi_{\text{inv}}(\delta(c_1, d^\# : \alpha), \delta(c_2, d^\# : \alpha)) && (\text{ind-}d^\#) \\
\forall d_1, d_2 \neq d^\#. \forall \alpha_1, \alpha_2. \phi_{\text{inv}}(c_1, c_2) &\implies \phi_{\text{inv}}(\delta(c_1, d_1 : \alpha_1), \delta(c_2, d_2 : \alpha_2)) && (\text{ind-non-}d^\#) \\
\phi_{\text{inv}}(c_1, c_2) &\implies \phi_{\text{indist}}(c_1, c_2) && (\text{indist})
\end{aligned}$$

If all four conditions hold, then by induction on trace length, any two traces agreeing on attacker inputs produce indistinguishable attacker observations at every step.

5.5 Results

We evaluate the conditional equality framework on two previously proposed partitioning schemes: DAWG [67] (hardware-based way partitioning) and COLORIS [133] (OS-level page coloring). Experiments run on an Intel i7-13700k at 5.2 GHz with 20 GB RAM.

5.5.1 DAWG: Dynamically Allocated Way Guard

DAWG [67] proposes secure *way partitioning* of set-associative caches. Privileged software allocates cache ways to processes based on resource utilization, and the hardware enforces isolation between ways allocated to mutually untrusting domains.

We implemented both Pseudo-Least Recently Used (PLRU) and Not Recently Used (NRU) eviction policies. In standard caches, PLRU employs a pointer to a metadata tree to identify the victim way, while NRU uses per-way access bits. In DAWG, updates to the metadata tree and victim-way identification are constrained by the domain partitioning: the policy must ensure that metadata updates for one domain do not modify the metadata section governing another domain’s ways, as doing so could alter eviction order across domains and create a side channel.

We implemented a Verilog module for the allocation and replacement policy component at the core of the DAWG design, performed self-composition (encoding the two-safety hyperproperty as a single-trace safety property [116]), and verified using SymbiYosys [134] with three approaches: k -induction (with conditional equality invariants), BMC, and PDR.

Table 5.1 shows the results. The conditional equality invariant enables k -inductive proofs completing in under 3 seconds, over three orders of magnitude faster than BMC at depth 20. PDR succeeds for PLRU but times out for NRU, illustrating that invariant-guided k -induction can, in our experiments, outperform even unbounded engines on certain policy structures.

Table 5.1: DAWG cache partitioning verification results.

Policy	Method	Runtime	vs. k -Ind
PLRU	k -induction ($k=10$)	2.8s	$1\times$
PLRU	BMC ($d=12$)	42.6s	$15\times$
PLRU	BMC ($d=20$)	145m 43s	$3100\times$
PLRU	PDR	24m 10s	$520\times$
NRU	k -induction ($k=10$)	2.7s	$1\times$
NRU	BMC ($d=12$)	1m 12s	$27\times$
NRU	BMC ($d=20$)	179m 16s	$4000\times$
NRU	PDR	Timeout	—

5.5.2 COLORIS: OS-Level Page Coloring

COLORIS [133] is an OS-level partitioning system that assigns *page colors* to memory pages based on address bits. By allocating pages with different colors to each process, COLORIS ensures that processes use different cache sets, providing coarse-grained cache isolation.

COLORIS was originally designed for performance (reducing cache contention) rather than security [133]. In scenarios where an attacker and victim share the same page color and cache set, side-channel leaks remain possible. We modified the page allocator in our model to ensure that each process’s page colors are disjoint, eliminating this vulnerability.

We developed a UCLID5 [105] model of a 4-way associative cache with the secure page allocator. BMC became infeasible beyond 3 steps due to the complexity of the combined hardware-software model. The k -inductive proof with conditional equality invariants completed in 16 minutes 33 seconds, demonstrating that the abstract resource model generalizes beyond hardware-only partitioning to combined hardware-software schemes.

5.5.3 Discussion

The invariant-based approach provides *unbounded* proofs (all possible traces), whereas BMC is limited to fixed depths, a qualitative difference rather than merely a speedup. The abstract resource model (index set, allocation map, policy functions) captures the essential structure shared by a wide class of resource partitioning schemes, not just caches, as evidenced by its application to both hardware way-partitioning and OS-level page coloring. The conditional equality framework has informed verification efforts for constant-time and stale-data properties of partitioned microarchitectures.

Part II

The Cross-Layer Gap

The Cross-Layer Gap

Security properties inherently span abstraction layers. A side-channel vulnerability involves microarchitectural state (cache lines, predictor entries, reuse buffers), ISA semantics (instruction ordering, memory operations), and software-level reasoning (which programs are vulnerable and how to detect them). Yet verification tools are layer-specific: hardware model checkers reason about RTL, software analyzers reason about programs, and neither naturally handles properties that span both.

This part addresses the *cross-layer gap*. The core challenge is that security architects specify policies *semantically*—“no information flow from secrets to attacker-observable outputs”—while much practical program analysis operates *syntactically*, matching instruction sequences against known attack patterns. Semantic specifications are general and precise but computationally expensive to verify; syntactic patterns are efficient but must be enumerated by hand, inevitably missing variants. We present *SemPat*, which bridges these two levels by automatically synthesizing k -complete attack patterns from hyperproperty specifications, achieving the completeness of semantic verification with the efficiency of syntactic pattern matching.

The material in this chapter is based on our CCS 2024 paper [47].

Chapter 6

Generating Syntactic Security Patterns from Semantic Properties

The material in this chapter is based on our CCS 2024 paper [47]. Code and reproducibility artifacts are available at <https://github.com/adwait/ccs2024-sempat>, with a Zenodo snapshot at <https://zenodo.org/records/12770008>.

6.1 The Semantic-Syntactic Divide

Speculative execution vulnerabilities can be detected through two fundamentally different approaches, each with complementary strengths and weaknesses. We ground the discussion in the Spectre Bounds Check Bypass (BCB) vulnerability [68] and its variants (Figure 6.1).

In Spectre-BCB, an attacker induces a call to `victimA` (Figure 6.1) with an out-of-bounds index `i`. While architecturally this is an illegal access, branch misprediction allows `arr1[i]` and then `arr2[arr1[i] << CL_INDEX]` to be accessed speculatively. The second load leaves a residue in the cache that depends on the accessed address; this residue survives speculative rollback and can be observed via timing (e.g., Prime+Probe [77]). Function `victimC` shows a variant targeting a computation-unit side channel: the loaded value feeds a floating-point multiply whose execution time is data-dependent [102], leaking the secret through timing rather than cache state. Over the years, many such variants have been demonstrated, varying both in the software gadget structure and the microarchitectural mechanism exploited.

6.1.1 Attack Pattern-based Analysis

One family of approaches detects vulnerabilities by matching program executions against *attack patterns*, small execution fragments indicating the presence of an exploit. Pattern A (Figure 6.1) matches executions where a load (A2) is speculatively executed following a branch (A1), and a subsequent load’s address (A3) depends on the value loaded by A2.

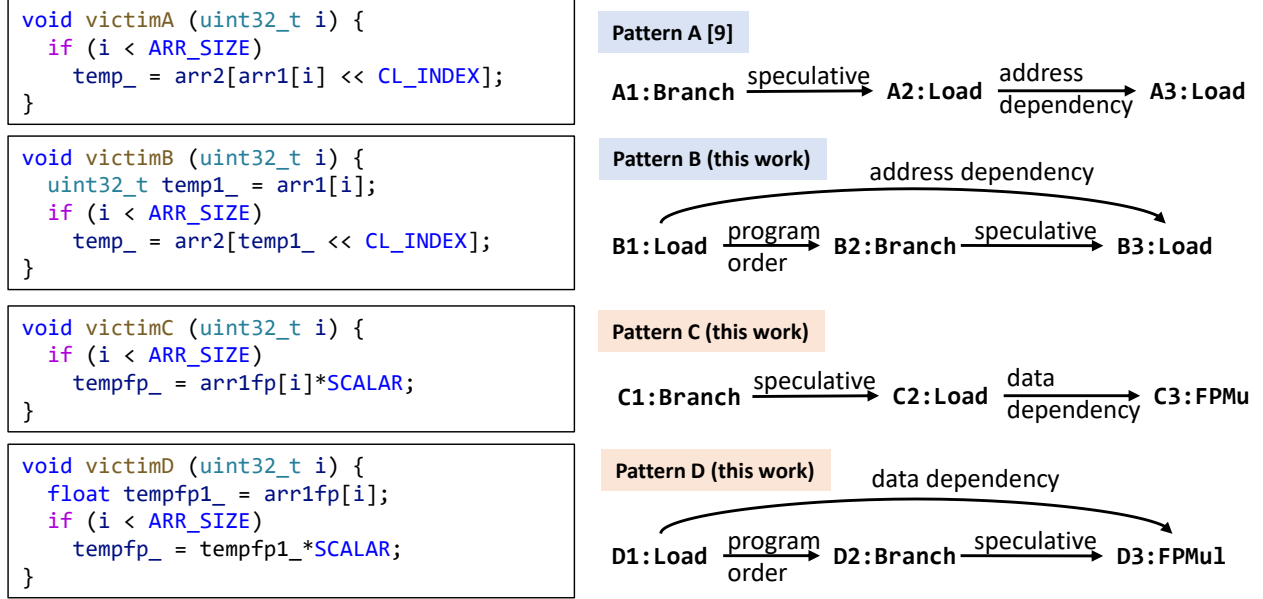


Figure 6.1: Four victim programs (left) and their attack patterns (right). Patterns A and B exploit cache side channels via speculative loads; Patterns C and D exploit computation reuse via speculative multiplies. Different gadget structures require different patterns, even when exploiting the same vulnerability. Table 6.1 maps patterns to their unifying hyperproperties.

Table 6.1: Pattern-to-hyperproperty mapping. A single SNI hyperproperty captures both variants in each pair, but pattern-based detection requires separate patterns for each.

Hyperproperty	Patterns	V_{pub}	V_{obs}
H_{AB} (cache)	A, B	regs, mem[arr1], mem[arr2], cache	cache.tags
H_{CD} (comp. reuse)	C, D	regs, mem[arr1fp], ...	#mul invocations

Table 6.2 summarizes existing approaches. Patterns defined at the architectural level abstract away complex microarchitectural details, resulting in simpler verification queries that scale to larger programs. However, patterns are tied to gadget-specific structure. Function `victimB` (Figure 6.1) is a modified version of `victimA` where the load `arr1[i]` is performed non-speculatively, before the branch. Due to this load-branch inversion, Pattern A does not match `victimB`, even though both exploit the same Spectre-BCB mechanism. Pattern B (which reorders the branch with the first load) is needed to capture `victimB`. Similarly, Patterns C and D target the computation-reuse side channel with different orderings.

Figure 6.2 shows how each pattern embeds in its corresponding compiled program. Pattern matching is fast (~ 5 seconds per program) and, because it avoids self-composition, is expected to scale better to larger programs. However, each structural variant requires a

Table 6.2: Attack-pattern (AP) approaches: execution models and pattern variants.

Approach	Execution Model	Pattern Variant
CheckMate [118]	μ hb graphs	Bad execution patterns
Cats vs. Spectre [73]	cat-based event structures	$\text{sec} \rightarrow^* \text{Obs}$ paths
Axiomatic HW/SW [88]	Leakage containment models	Transmitters

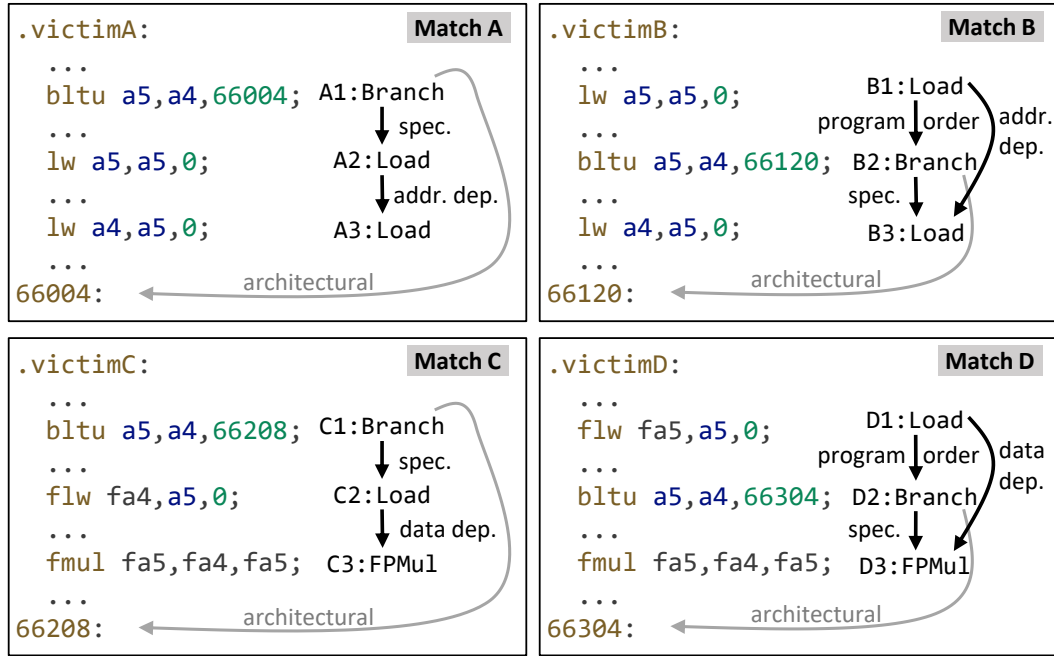


Figure 6.2: Patterns A–D matched against compiled executions of `victimA`–`victimD`. Each pattern embeds in the corresponding program’s instruction sequence through opcode and dependency matching.

separate pattern, and pattern generation has historically been manual [19, 34]. Every new microarchitectural feature (computation reuse, store-to-load forwarding) potentially introduces new variants that existing pattern sets do not cover.

6.1.2 Semantic Hyperproperty-based Analysis

The alternative family of approaches checks non-interference properties (such as SNI, Definition 3.1) directly against a formal platform model that includes microarchitectural detail. Table 6.3 summarizes existing approaches and their NI variants. A single hyperproperty can capture multiple exploit variants: the SNI hyperproperty H_{AB} (Figure 6.1) identifies both

Table 6.3: Semantic hyperproperty (SH) approaches and their non-interference variants.

Approach	NI Variant
Cheang et al. [21]	Trace-property dependent obs. determinism
Spectector [58]	Speculative non-interference
InSpectre [56]	Conditional non-interference
HW/SW Contracts [57]	Contract-conditioned NI
Fabian et al. [34]	Speculative NI

`victimA` and `victimB` as exploits, since both violate the same non-interference property regardless of gadget structure. Similarly, H_{CD} captures both computation-reuse variants. While the hyperproperty changes with the platform’s microarchitectural features (e.g., speculation mechanism, side channel), it is robust to variances in program structure.

However, semantic verification requires self-composition, which doubles the state space. Incorporating detailed microarchitectural state (cache tags, reuse buffers) further inflates the verification cost. In practice, even short programs (5–10 instructions) can take minutes to hours on realistic platform models [58].

6.1.3 Bridging the Divide

The question that motivates SemPat is whether we can combine the completeness of semantic verification with the efficiency of syntactic pattern matching: given a non-interference hyperproperty that uniformly captures a vulnerability class, can we automatically derive all the patterns needed for scalable program analysis? Figure 6.3 shows the SemPat pipeline that does so.

6.2 Attack Patterns and k -Completeness

We formalize the notion of attack patterns and the completeness guarantee that bridges the two paradigms. Throughout this section, we work with a platform model M that defines the microarchitectural state (registers, memory, caches, predictors, reuse buffers), a set of operations Ops (load, store, branch, multiply, etc.), and a transition semantics T specifying how state evolves under each operation. Speculative and non-speculative semantics are distinguished as T and T_{ns} respectively; the non-interference property $NI(\text{init}, V_{\text{pub}}, V_{\text{obs}})$ requires that secret inputs do not influence attacker-observable outputs under T , where init constrains initial states, V_{pub} identifies public state, and V_{obs} identifies observable state (cf. Definition 3.1).

Definition 6.1 (Attack Pattern). An *attack pattern* $\rho = (\pi, \psi)$ consists of:

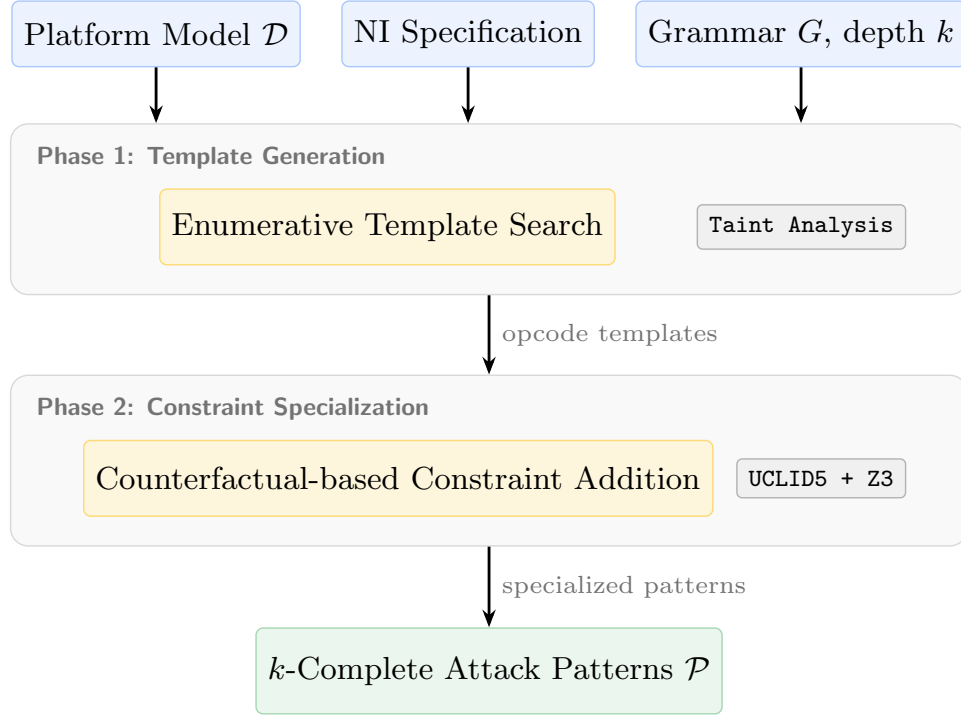


Figure 6.3: SemPat pipeline: a platform model, NI specification, and predicate grammar feed a two-phase algorithm. Phase 1 generates opcode templates; Phase 2 specializes each into a precise attack pattern. The output is a k -complete pattern set.

- A *template* $\pi = \text{op}_0 \cdot \text{op}_1 \cdots \text{op}_k$: a sequence of instruction opcodes
- A *constraint* ψ : a Boolean formula over instruction attributes (registers, dependencies, addresses) restricting when the pattern indicates a vulnerability

Definition 6.2 (Pattern Embedding). A pattern $\rho = (\pi, \psi)$ *embeds* in an execution trace $\tau = \sigma_0, \sigma_1, \dots$ at indices $(i_1 < i_2 < \cdots < i_k)$ iff:

1. Opcodes match: $\text{opcode}(\sigma_{i_j}) = \text{op}_j$ for all j
2. Constraint satisfied: $(\sigma_{i_1}, \dots, \sigma_{i_k}) \models \psi$

A program P *matches* pattern ρ if there exists an initial state σ_0 such that the execution of P from σ_0 embeds ρ .

A subtle but critical issue arises when relating pattern size to program size. A long program may violate non-interference through only a small subsequence of instructions—

those that are transitively related through data or control dependencies. This motivates the *skeleton* concept.

Definition 6.3 (NI-Violation Skeleton). For a program $P = \sigma_0 \sigma_1 \dots$ and execution trace τ , define:

- $\text{dep}_\tau(i) \subseteq V$: the variables that instruction σ_i depends on
- $\text{lw}_\tau(v, i)$: the last instruction that writes variable v before index i
- $\text{idep}_\tau(i) = \bigcup_{v \in \text{dep}_\tau(i)} \text{lw}_\tau(v, i)$: all instruction dependencies of σ_i

A subsequence $i_1, \dots, i_k$ is *closed under dependencies* if for all j , $\text{idep}(i_j) \subseteq \{i_1, \dots, i_k\}$. Program P has an *NI-violation skeleton of size k* , written $P \not\models_k \text{NI}$, if there exist traces with a dependency-closed subsequence of length k that violates NI.

The key observation is that fixing a skeleton size does *not* bound the program size: a long program can have a small skeleton. For example, a 1000-instruction program may violate NI through just 3 instructions (a branch, a load, and a dependent load) whose transitive dependencies close within those 3 instructions.

The property connecting the two paradigms is *k-completeness*:

Definition 6.4 (*k-Completeness*). A pattern set $\mathcal{P}$ is *k-complete* for a non-interference specification $\text{NI}(\text{init}, V_{\text{pub}}, V_{\text{obs}})$ on platform M iff: for all programs P ,

$$P \not\models_k \text{NI}(\text{init}, V_{\text{pub}}, V_{\text{obs}}) \implies \exists \rho \in \mathcal{P}. P \models \rho$$

That is, every NI violation with a skeleton of size at most k is detected by some pattern in $\mathcal{P}$.

k-completeness provides a soundness guarantee for the syntactic approach: if no pattern in a *k-complete* set matches a program, then the program satisfies non-interference for all violation skeletons up to size k . The patterns may still have false positives (matching programs that are actually secure), but they have *no false negatives* up to skeleton depth k . The skeleton-based formulation is strictly more powerful than bounding program length: it detects violations in arbitrarily long programs, provided the violating subsequence is small.

6.3 The SemPat Algorithm

SemPat automatically generates a *k-complete* pattern set from a platform model M , a non-interference specification, and a predicate grammar G . The algorithm has two phases.

Algorithm 3 `GenerateTemplates`(M, NI, d)

Require: Platform model M , NI property $\text{NI}(\text{src}, \text{obs}, \text{init})$, depth d

Ensure: Set of violating opcode templates

```

1:  $acc \leftarrow \emptyset$ 
2: procedure TEMPLATEHELPER( $\pi$ )
3:   if  $|\pi| < d$  then
4:     for each opcode  $op \in \text{Ops}$  do
5:       if  $op \cdot \pi$  propagates taint from  $\text{src}$  to  $\text{obs}$  then
6:         if  $op \cdot \pi \not\models \text{NI}$  then ▷ Semantic check
7:            $acc \leftarrow acc \cup \{op \cdot \pi\}$ 
8:         end if
9:       end if
10:      TEMPLATEHELPER( $op \cdot \pi$ )
11:    end for
12:  end if
13: end procedure
14: TEMPLATEHELPER( $\epsilon$ )
15: return  $acc$ 

```

6.3.1 Phase 1: Template Generation

The first phase enumerates all opcode templates (up to depth d) that can potentially violate non-interference. Algorithm 3 shows the procedure: a recursive DFS over operation sequences applies two filters at each step. First, a *syntactic taint analysis* overapproximates checks whether the template propagates taint from secret inputs to observable outputs; templates without taint propagation cannot violate non-interference and are pruned without invoking the solver. Second, for templates that pass the taint filter, a *semantic NI check* via self-composition (UCLID5/Z3) determines whether there exist concrete operand assignments making the template violate NI. Only semantically violating templates are retained as output.

We state this formally:

Lemma 6.1 (Template Generation Soundness). *For $k \leq d$, if $\text{inst}_1 \cdots \text{inst}_k \not\models \text{NI}$ where instruction $\text{inst}_i = \text{op}_i(\omega_i)$, then $\text{op}_1 \cdots \text{op}_k \in \text{GenerateTemplates}(M, \text{NI}, d)$.*

Proof sketch. The proof is a direct consequence of Algorithm 3 and the soundness of taint-based overapproximation. If a template w does not propagate taint from V_{pub}^C to V_{obs} , then it does not violate $\text{NI}(V_{\text{pub}}, V_{\text{obs}}, \Sigma_{\text{init}})$. Contrapositive: any NI-violating sequence passes the taint filter and is retained. $\square$

Table 6.4: Default predicate grammar for SemPat pattern constraints.

Predicate Atom	Meaning	RISC-V Encoding
$\text{datadep}(i_1, i_2)$	Data operand of i_2 depends on result of i_1	Last write to $i_2.\text{rs1}$ or $i_2.\text{rs2}$ is by i_1
$\text{addrdep}(i_1, i_2)$	Address operand of i_2 depends on result of i_1	Last write to $i_2.\text{rs1}$ is by i_1
$\text{sameaddr}(i_1, i_2)$	Same memory address	$i_1.\text{addr} = i_2.\text{addr}$
$\text{diffaddr}(i_1, i_2)$	Different memory address	$i_1.\text{addr} \neq i_2.\text{addr}$
$\text{srcdata}_r(i)$	Data operand read from register r	$i.\text{rs1} = r$ or $i.\text{rs2} = r$
$\text{srcaddr}_r(i)$	Address operand read from register r	$i.\text{rs1} = r$
$\text{destreg}_r(i)$	Result written to register r	$i.\text{rd} = r$
$\text{speculates}(i)$	Instruction initiates speculation	i sets the spec variable
$\text{lowoperands}(i)$	Operands independent of secrets	$\sigma_1(i.\text{ops}) = \sigma_2(i.\text{ops})$
$\text{lowresult}(i)$	Result independent of secrets	$\sigma_1(i.\text{rd}) = \sigma_2(i.\text{rd})$

6.3.2 Predicate Grammar

The constraint ψ in a pattern (π, ψ) is a conjunction of *predicate atoms* drawn from a grammar G . Each predicate has a fixed arity and is applied to instructions from the template. Table 6.4 shows the default grammar.

The grammar exposes a precision-robustness tradeoff: architectural predicates (register dependencies, data flow) produce patterns portable across implementations but with more false positives, while microarchitectural predicates (set indices, buffer state) yield more precise patterns tied to specific designs. We explore this tradeoff in Section 6.4.

6.3.3 Phase 2: Constraint Specialization

For each template π from Phase 1, Phase 2 synthesizes a constraint ψ that precisely characterizes when the template indicates a vulnerability. Algorithm 4 shows the procedure, which iterates over predicate atoms using *counterfactual addition*.

The key observation underlying counterfactual addition is:

Lemma 6.2 (Counterfactual Soundness). *For a pattern (π, ψ) with $|\pi| = k$ and atom f : if for all programs $P = i_1 \cdots i_k$,*

$$P \models (\pi, \psi \wedge \neg f) \implies P \models \text{NI}$$

then for all P : $(P \models (\pi, \psi) \wedge P \not\models \text{NI}) \implies P \models (\pi, \psi \wedge f)$.

That is, if adding $\neg f$ eliminates all violations, then every violating program must satisfy f , so conjoining f preserves completeness. The algorithm collects all atoms by applying predicates from G to template positions (line 6), then iterates: for each atom, if the counterfactual test passes (line 3), the atom is added to the constraint (line 4); otherwise it is skipped (line 5).

Algorithm 4 `ConstraintSpecialize`(M, NI, π, G)

Require: Platform M , NI property, template π , grammar G

Ensure: Set of patterns

```

1:  $acc \leftarrow \emptyset$ 
2:  $L \leftarrow \text{ApplyPredicates}(\pi, G)$  ▷ All atoms from  $G$  applied to  $\pi$ 
3: procedure CONSHelper( $\psi, i, L$ )
4:   if  $i > |L|$  then
5:      $acc \leftarrow acc \cup \{(\pi, \psi)\}$  ▷ Exhausted all atoms
6:   else if  $\forall P. P \models (\pi, \psi \wedge \neg L[i]) \implies P \models \text{NI}$  then
7:     CONSHelper( $\psi \wedge L[i], i + 1, L$ ) ▷ Add  $L[i]$ 
8:   else
9:     CONSHelper( $\psi, i + 1, L$ ) ▷ Skip  $L[i]$ 
10:  end if
11: end procedure
12: CONSHelper(true, 0,  $L$ )
13: return  $acc$ 

```

When no single atom passes the counterfactual test, the algorithm generalizes to *multiple counterfactuals*:

Lemma 6.3 (Multiple Counterfactuals). *For pattern (w, ϕ) , atoms $\{f_i\}_i$, and for any $C = \text{inst}_1 \cdots \text{inst}_{|w|}$:*

$$\begin{aligned}
 (C \models (w, \phi \wedge \bigwedge_i \neg f_i) \implies C \models \text{NI}) &\implies \\
 ((C \models (w, \phi) \wedge C \not\models \text{NI}) \implies \bigvee_i (C \models (w, \phi \wedge f_i))) &
 \end{aligned}$$

Proof. Consider a partially specialized template (w, ϕ) and a set of atoms $\{f_i\}_i$. Define sets of programs: $S = \{C \mid C \models (w, \phi)\}$; for each f_i , let $S_i = \{C \mid C \models (w, \phi \wedge f_i)\}$; and $T = \{C \mid C \models \text{NI}\}$. Then the set $U = S \setminus \bigcup_i S_i$ consists of programs satisfying $C \models (w, \phi \wedge \bigwedge_i \neg f_i)$.

Suppose the claim is false. Then there exists a program C such that (a) $C \in S \cap \overline{T}$, and (b) $C \notin \bigcup_i S_i$. Thus $C \in (S \cap \overline{T} \cap \overline{\bigcup_i S_i})$, implying by the definition of U that $C \in U \cap \overline{T}$. However, this means that the antecedent of the claim (which corresponds to $\overline{U} \cup T$) does not hold, a contradiction. $\square$

Lemma 6.2 is the special case with a single atom. Considering multiple counterfactuals $\{f_i\}$ results in a disjunctive branching (bolded $\bigvee_i$) over atoms: **ConstraintSpecialize** recursively invokes **ConSHelper** on $(w, \phi \wedge f_i)$ for each i . This is sound as the collection of patterns together covers all violating executions.

Theorem 6.1 (k -Completeness of SemPat). *Let $W = \text{GenerateTemplates}(M, \text{NI}, d)$ and $P_i = \text{ConstraintSpecialize}(M, \text{NI}, w_i, G)$ for each $w_i \in W$. Then $\bigcup_i P_i$ is d -complete.*

Proof. Suppose **ConstraintSpecialize** is invoked with arguments (w, NI, M, G) . We claim that in Algorithm 4, at each recursive call of **ConsHelper** with arguments (ϕ, i, L) , the following property holds:

$$\begin{aligned} \forall C. \quad & (C \models (w, \text{true}) \wedge C \not\models \text{NI}) \\ \implies & \exists (w, \phi') \in \text{acc} \cup \{(w, \phi)\}. C \models (w, \phi') \end{aligned}$$

In words, the partially specialized patterns maintained in the accumulator together form an overapproximation of the violating programs for template w .

Base case: At the start, **ConsHelper**(true, 0, L) immediately implies the property.

Inductive step: Let the property hold for **ConsHelper**(ϕ, i, L). Then, either we (a) add (w, ϕ) to acc , (b) add $L[i]$ to ϕ and recurse, or (c) skip over $L[i]$ and recurse. The property holds immediately in cases (a) and (b). In case (c), we note that the set of programs C satisfying (w, ϕ) and violating **NI** is identical to those satisfying $(w, \phi \wedge L[i])$, due to Lemma 6.2. A similar argument holds for the multi-counterfactual case.

Therefore, by induction, the property holds for all recursive calls of **ConsHelper**, and hence the final set of patterns in acc (when there are no more calls to **ConsHelper**) is a d -complete set of patterns for w . $\square$

6.3.4 Running Example: Computation Reuse

Consider a platform with a *computation reuse buffer*—a cache of recently computed results. When a multiply instruction’s operands match a buffered entry, the result is reused without invoking the multiplier. The multiplier invocation count is attacker-observable (via timing).

The NI specification: memory contents (secrets) should not influence the multiplier count (observable output).

SemPat’s template generation discovers that sequences of length ≤ 2 cannot violate NI (the reuse buffer starts empty, so no reuse occurs with fewer than three operations). At depth 3, the template `mul--load--mul` violates NI: the first multiply primes the reuse buffer, the load brings a secret-dependent value into a register, and the second multiply either hits or misses the reuse buffer depending on whether the loaded value matches the primed operands.

Constraint specialization adds two atoms: `datadep(load, mul2)` (the second multiply must depend on the loaded value) and `highresult(load)` (the loaded value must be secret-dependent). The resulting pattern is precise: it matches only programs where the reuse buffer creates a secret-dependent timing difference.

6.4 Results and Insights

We evaluate SemPat on two platform designs, described below. All experiments use UCLID5 [105] with Z3 [89] on an Intel i9-10900X with 64 GB RAM.

```

1 type reentry_t = record { op1: word_t, op2: word_t, result: word_t };
2 var reuse_buf : [instr_ind_t]reentry_t;
3
4 operation mulop (rd, rs1, rs2) {
5   op1 = regfile[rs1]; op2 = regfile[rs2];
6   // Check for reuse
7   if (∃ i. reuse_buf[i].op1 == op1 &&
8       reuse_buf[i].op2 == op2) {
9     result = reuse_buf[i].result; // Reuse result
10  } else { // Otherwise invoke multiplier unit
11    result = multiplier(op1, op2);
12    mulcount = mulcount + 1;
13  }
14  // Replace reuse_buf entry (at index ind_)
15  reuse_buf[ind_] = reentry_t {op1, op2, result};
16  regfile[rd] = result;
17 }

```

Listing 6.1: M_{CR} : computation reuse platform model.

6.4.1 Platform Designs

The *computation reuse* platform (M_{CR} , Figure 6.1) models dynamic reuse of results of previous high-latency computations, based on the instruction reuse scheme of Sodani and Sohi [113]. The model includes a reuse buffer (`reuse_buf`) that records operands and results of multiply operations; future operations with matching operands reuse the buffered result without invoking the multiplier. The multiplier invocation count (`mulcount`) is the attacker-observable output.

The *store optimization* platform (M_{STL} , Figure 6.2) models silent stores [75, 74] and store-to-load forwarding via an LSQ cache (load-store queue cache) [75]. On a load, the LSQ cache is checked for a store to the same address; if found, the load is served from the buffer. On a store, the buffer is checked for a matching entry; if the value matches, the store is squashed (silenced). The memory access count (`lscount`) is the attacker-observable output.

Both platforms are extended with speculation primitives (branch speculation for M_{CR} , store-to-load forwarding speculation for M_{STL}) and composed with the speculative non-interference property from Definition 3.1.

6.4.2 Pattern Discovery

We evaluated SemPat on two platform variants. Table 6.5 summarizes the generation times across word widths.

On the branch speculation platform, SemPat discovered patterns C and D (cache side channel via computation reuse under speculation) in 69 seconds at 32-bit word width. On the store-to-load forwarding platform, SemPat discovered pattern E (Figure 6.4) in 92 seconds

```

1 type lsqc_entry_t = record { valid: boolean,
2   is_load: boolean, addr: word_t, data: word_t }
3 var lsqc : [lsqc_index_t]lsqc_entry_t;
4 var lptr : lsqc_index_t; // Pointer into LSQC
5 var lscout : int; // Number of memory accesses
6
7 operation store (rs2, rs1, imm) {
8   addr = addrgen(regfile[rs1], imm);
9   data = regfile[rs2];
10  if (lsqc[lptr].valid && lsqc[lptr].addr == addr
11    && lsqc[lptr].data == data) {
12    // Store squashing (silent store)
13  } else { // Perform memory access
14    mem[addr] = data; lscout += 1;
15    lsqc = ... // Update the lsqc
16  }
17 }
18 operation load (rd, rs1, imm) {
19   addr = addrgen(regfile[rs1], imm)
20   if (lsqc[lptr].addr == addr && lsqc[lptr].valid){
21     data = lsqc[lptr].data; // Data forwarding
22   } else { // Perform memory access
23     data = mem[addr]; lscout += 1;
24     lsqc = ... // Update the LSQC
25   }
26   regfile[rd] = data;
27 }
    
```

 Listing 6.2: M_{STL} : store optimization platform model with LSQC cache.

Table 6.5: Pattern generation times by platform and word width.

Platform	ww=16	ww=20	ww=24	ww=32
Branch + CR	—	—	—	69s
STL + CR	92s	334s	timeout (1 hr)	

at 16-bit word width; generation timed out at higher word widths due to solver scalability. Patterns C, D, and E were *not present in prior manual enumerations* [19, 34], demonstrating that automated synthesis can discover patterns targeting microarchitectural features not covered by prior manual enumerations.

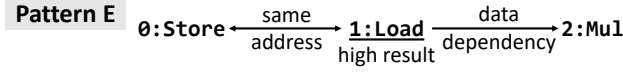


Figure 6.4: Pattern E: discovered by SemPat for the store-to-load forwarding platform. The pattern captures a store-load-load sequence where forwarding from the STL buffer creates a timing side channel.

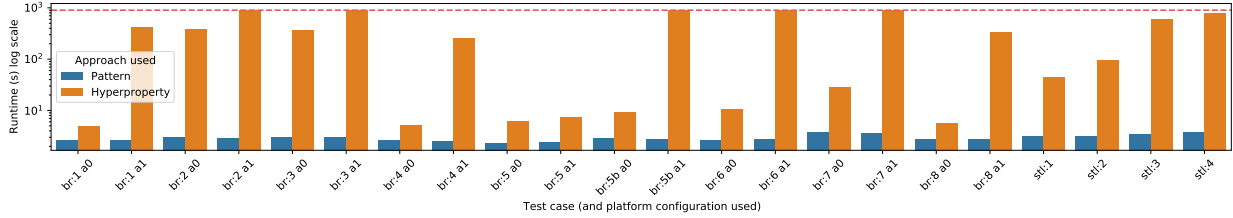


Figure 6.5: Verification times for Branch and STL speculation litmus tests using hyperproperty-based (dark) and pattern-based (light) detection. Pattern-based analysis completes in ~ 5 seconds for all tests, while hyperproperty-based analysis varies from minutes to timeouts (15-minute limit).

6.4.3 Verification Speedup

We compared pattern-based and hyperproperty-based detection on a suite of 13 litmus tests: 9 SpectreV1-CR variants and 4 SpectreV4-CR variants, adapted from Kocher [68] and Daniel et al. [30] by replacing the secret-dependent load with a multiply targeting the computation-unit side channel. Of these, 12 are unsafe and 1 is safe.

Figure 6.5 compares the two approaches. Pattern-based analysis completed all tests in approximately 5 seconds each, while hyperproperty-based analysis ranged from minutes to timeouts at a 15-minute limit, a speedup of $20\text{--}120\times$ on tests that completed under both approaches. Both methods agreed on all verdicts, consistent with the k -completeness guarantee: no false negatives occurred in the pattern-based analysis, and the single safe program was correctly not matched by any pattern.

The pattern approach was also more robust to microarchitectural variation. For some SpectreV1-CR tests (e.g., **br:1**, **br:4**), the hyperproperty check converged on a non-associative cache configuration but timed out on an associative one. Since pattern-based analysis operates over architectural state, it is resilient to such microarchitectural differences.

Binary analysis methodology. To apply generated patterns to compiled programs, we disassemble the binary using `riscv64-unknown-elf-objdump` and extract the control flow graph (CFG). The CFG is unrolled to a fixed depth from the function entry point. Branch instructions are instrumented with assumptions corresponding to the branch condition, allowing the branch to be bypassed when speculating: for example, “`if (x == 0) stmt`” be-

comes “`assume(x == 0 || spec); stmt`”. Each unrolling is analyzed independently. For pattern-based analysis, all subsequences matching the pattern template are identified, and for each, an SMT query checks whether the pattern constraint is satisfied. For hyperproperty-based analysis, the property is encoded over a self-composition [116] of the platform model executing the unrolled program. Both analyses are performed using UCLID5.

6.4.4 Scalability

Generation time scales primarily with two parameters: microarchitectural model size and grammar depth. Figure 6.6(a,b) shows the effect of varying set and way index widths on the LSQ cache platform. Way index width has a larger impact than set count: a larger number of sets increases bitvector widths (handled efficiently by word-level SMT reasoning), while larger associativity increases the number of explicit case-splits in the encoding. Figure 6.6(c) shows a similar trend for reuse buffer size.

Figure 6.6(d) isolates the effect of grammar depth on a synthetic parameterized platform $M_{\text{synth}}(k)$ with k cascaded buffers. When grammar depth equals platform depth ($d = k$), only one pattern is generated. At $d = k + 1$, redundant instruction sequences require 2-counterfactual branching; at $d = k + 2$, 3-counterfactuals. This branching causes exponential growth, but generation remains feasible (under 10 minutes) for realistic configurations.

6.4.5 Grammar Design and False Positives

The grammar controls a fundamental precision-robustness tradeoff. To illustrate, consider an LSQ cache variant with set-indexed invalidation: a store invalidates all entries with the same set index. With the default grammar (address-level predicates), SemPat produces pattern F (Figure 6.7, left). This pattern flags code where a store’s full address differs from the first load’s, but their set indices (an address slice) may still match, causing a false positive when the store would in fact invalidate the LSQ entry.

Augmenting the grammar with a diffindex predicate, which compares set indices rather than full addresses, produces the more precise pattern G (Figure 6.7, right). This pattern correctly avoids the false positive.

Figure 6.8 illustrates a concrete false positive. Pattern F flags `test_K` due to a match on instructions `Ld0`, `St0`, `Ld1`. While their full addresses differ, the set indices (an address slice) may be identical when $K > \text{SET_W} + 2$, causing the store to invalidate the LSQ entry and making the program safe. Pattern G, using the diffindex predicate, flags `test_K` only when $K \leq \text{SET_W} + 2$. Table 6.6 summarizes.

The choice of grammar is a design decision: architectural predicates (register dependencies) produce patterns portable across implementations, while microarchitectural predicates (set indices) yield more precise patterns tied to specific designs.

Example 6.1 (AES Scenario: Cache Side-Channel Patterns). The cache side-channel attacks on T-table AES (Section 1.1) exemplify the class of vulnerabilities that SemPat can

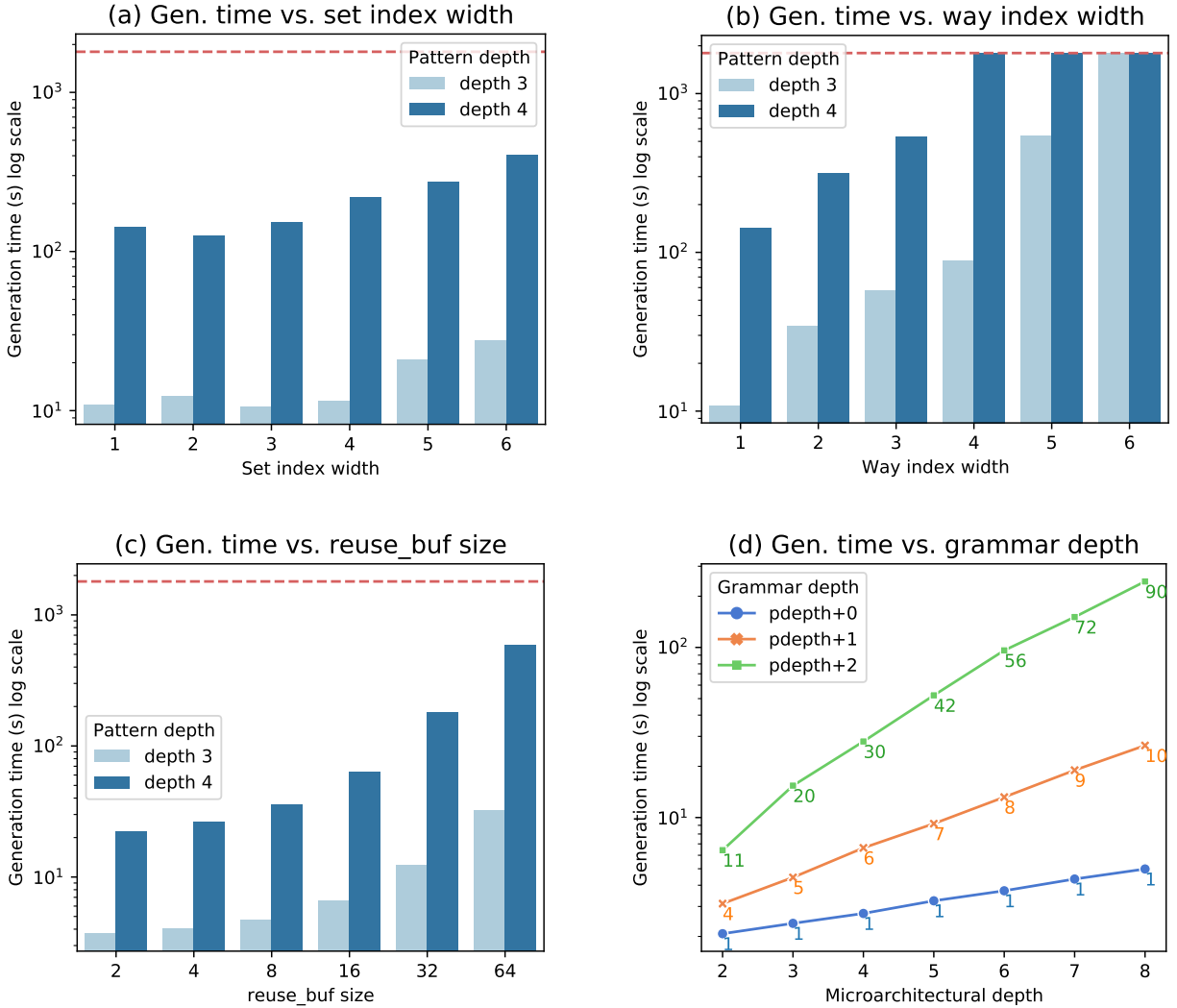


Figure 6.6: Scalability of pattern generation. (a) Varying set index width of the LSQ cache (same associativity). (b) Varying way index width (same sets). (c) Varying reuse buffer size. (d) Generation time and pattern count for the synthetic platform $M_{\text{synth}}(k)$ with varying microarchitectural depth (k) and grammar depth (d).

target. Patterns A and B (Figure 6.1) capture speculative cache leaks: a branch followed by data-dependent loads whose addresses enter the cache during speculation. In the AES context, these correspond to a mispredicted bounds check followed by a T-table lookup $\text{Te0}[\text{s}[\text{j}]]$ that loads a key-dependent cache line. Given a platform model that faithfully captures cache side-channel behavior, SemPat’s k -completeness guarantee would ensure that all such cache-based Spectre patterns up to skeleton depth k are detected, including Pattern B, which reorders the load before the branch and was absent from prior manual enumer-



Figure 6.7: (F) Pattern using the default grammar with address-level predicates. (G) More precise pattern after adding the diffindex predicate, which compares set indices rather than full addresses.

```

void test_K (uint32_t idx) {
    // Address (A) = (arr1+idx)
    _temp = arr1[idx];    // Ld0: LSQC Index = A[SET_W+1:2]
    arr1[idx+(1<<K)] = 0; // St0: LSQC Index = (A+(1<<K))[SET_W+1:2]
    _temp1 = arr2[_temp]; // Ld1
}
    
```

Figure 6.8: Example program `test_K`: the store’s full address differs from the first load’s, but their set indices may match depending on `K` and `SET_W`, causing pattern F to produce a false positive when the store would in fact invalidate the LSQ entry.

Table 6.6: Precision comparison on `test_K` with set index width `SET_W`.

Check	$K > SET_W + 2$	$K \leq SET_W + 2$
Hyperproperty	Safe	Unsafe
Pattern F (default grammar)	Unsafe (false positive)	Unsafe
Pattern G (+diffindex)	Safe	Unsafe

ations. Applying SemPat to a cache side-channel platform model to generate these patterns is straightforward but was not part of the current evaluation.

6.5 Related Work

Pattern-based detection. Prior work on pattern-based detection manually defines patterns: CheckMate [118] identifies bad execution patterns in μ hb graphs, Cats vs. Spectre [73] uses `cat`-based event structures, and Axiomatic HW/SW Contracts [88] defines transmitters in leakage containment models. SemPat complements these by automatically generating patterns that are provably complete up to skeleton depth k .

Semantic hyperproperty verification. SemPat takes as input the non-interference specifications developed by approaches such as Cheang et al. [21], Spectector [58], InSpectre [56], and the automated detection framework of Fabian et al. [34]. Hardware-software contracts [57, 125] provide a related abstraction at the hardware-software interface. SemPat’s contribution is orthogonal: it converts these semantic specifications into efficient syntactic patterns.

Symbolic software analysis. Specialized symbolic analysis tools target specific leakage models (e.g., constant-time execution): KLEESpectre [124] and SPECUSYM [59] extend symbolic execution for speculative vulnerabilities, while relational symbolic execution tools [30, 37] check two-safety properties directly. See Geimer et al. [42] for a systematic evaluation. SemPat is orthogonal to these: its goal is pattern generation, not binary analysis, and it is not limited to a specific leakage model.

Hardware security verification. Several approaches verify hardware RTL directly: IO-DINE [45] checks constant-time properties, Revizor [95] uses fuzzing-based testing, and Huzzle [66] combines hardware fuzzing with information-flow analysis. These extract security-relevant hardware abstractions but must be paired with software analysis for end-to-end guarantees; SemPat’s patterns provide the software-side complement.

6.6 Discussion

SemPat demonstrates that synthesis can bridge abstraction layers: the synthesized patterns bring microarchitectural semantics (the platform model, speculation behavior) to bear on software analysis (pattern matching over programs). Pattern matching itself is a standard program analysis technique; what makes it effective for cross-layer security is the synthesized pattern set that encodes microarchitectural information in a form that software-level tools can consume.

Three aspects of the results merit emphasis. First, automated synthesis discovers patterns for microarchitectural features beyond those targeted by prior manual enumerations: SemPat found two new Spectre-related patterns involving computation reuse, a microarchitectural feature not covered by prior manual enumeration, with completeness up to depth k guaranteed by construction. Second, the semantic heavy lifting is amortized into one-time pattern generation; once generated, patterns enable up to two orders of magnitude faster program analysis compared to direct hyperproperty checking. Third, the grammar is a user-facing knob for navigating the precision-robustness tradeoff, allowing the same framework to produce either portable architectural patterns or precise implementation-specific ones.

A promising direction is integrating SemPat’s pattern-based analysis with PyCaliper’s specification infrastructure (Chapter 9), enabling patterns to be expressed and checked within the same framework used for invariant synthesis and verification.

Part III

The Representation-to-Tool Gap

Chapter 7

Ax2Op: Converting Axiomatic to Operational Memory Models

Formal specifications are only useful if verification tools can consume them. The *representation-to-tool gap* arises when a specification is expressed in one paradigm—for example, as axiomatic happens-before constraints—but the most powerful verification engines expect another, such as operational state machines. This gap is not a deficiency of either paradigm; it reflects the fact that the most natural representation for *authoring* specifications (concise, declarative axioms that architects can review and compare across designs) is often different from the most natural representation for *verification* (explicit state machines that model checkers can consume).

This chapter presents Ax2Op, which automatically converts axiomatic memory consistency specifications (μspec) to operational automata, unlocking IC3, BMC, and PDR for happens-before specifications. Together with PipeSynth (Chapter 8), which synthesizes μspec axioms pre-RTL from litmus tests and execution traces, they form a pipeline from architect intuitions through formal specifications to unbounded verification proofs, with manual RTL instrumentation as the remaining step.

The material in this chapter is based on our FMCAD 2022 paper [46]. Code and examples are available at <https://github.com/adwait/axiomatic-operational-examples>.

7.1 The Paradigm Mismatch

Three challenges from Chapter 3 converge in this chapter. The most fundamental is *paradigm incompatibility* (Challenge 3.4): axiomatic memory models (Definition 3.4) specify valid executions via happens-before constraints that are concise and implementation-independent, but IC3, PDR, and other property-directed verification techniques expect operational models (Definition 3.3) with explicit state and transitions. Prior to Ax2Op, tools in the PipeCheck-/CCICheck-/COATCheck-/TriCheck suite [78, 82, 80, 119] verified μspec axioms by enumerating bounded litmus-test executions. PipeProof [83] went further: it performs refinement

proofs of a μ spec microarchitectural model against an ISA-level memory consistency specification, with all-program coverage rather than per-litmus-test reasoning. What was still missing was *operationalization*: converting an axiomatic specification into a finite-state operational model suitable for IC3/PDR and for direct composition with RTL, so that property-directed model checkers can be brought to bear on μ spec specifications. A closely related challenge is the *pre-RTL specification gap* (Challenge 3.3): writing μ spec axioms (Definition 3.6) demands joint expertise in formal methods and microarchitecture, yet architects more naturally express design intent through concrete examples—litmus tests exhibiting forbidden behaviors, execution traces from simulators—than through formal axioms. Finally, *axiom correctness* (Challenge 3.3) is difficult to ensure for hand-written axioms, which may be incomplete, overly restrictive, or mutually inconsistent.

The common theme is that specifications exist in representations unsuited to available tools. Synthesis bridges the gap by transforming representations: from axiomatic to operational (Ax2Op), or from behavioral examples to axiomatic (PipeSynth).

Example 7.1 (Store Ordering on Multi-Core RISC-V). Consider a multi-threaded program where one core produces data and stores it to shared memory, while another core consumes it. Correct behavior requires store-ordering guarantees: the consumer must not read stale or uninitialized values. These ordering constraints are naturally expressed as μ spec happens-before axioms (“a store to address a happens-before a load from a on another core”). But verifying that a pipelined processor enforces these orderings requires IC3 or PDR, tools that expect operational models, not axiomatic constraints. Ax2Op bridges this mismatch. While our evaluation uses Multi-Vscale, Tomasulo, and SDRAM Controller benchmarks, the store-ordering axioms are architecture-independent; the same store-ordering constraints apply to any RISC-V implementation that enforces the same memory model.

7.2 Operational Model of Computation

Recall from Section 3.3.3 that memory consistency models come in two flavors.

Axiomatic models define valid executions via constraints on happens-before orderings: an execution is valid if its μ hb graph is acyclic and all axioms are satisfied. The μ spec framework [78, 80] (Definition 3.6) specifies each axiom as a formula whose outermost quantifier prefix ranges over instruction variables. The typical form $\forall i_1, i_2. \text{ProgramOrder}(i_1, i_2) \Rightarrow \text{hb}(i_1.\text{Fetch}, i_2.\text{Fetch})$ asserts in-order fetch using only universal quantifiers, but μ spec axioms may also introduce existential quantifiers within the body—for example, to express that for every load there *exists* a store that supplies its value—and the construction in this chapter handles both quantifier types. The axiomatic tradition has a long history at the ISA level, from Alglave et al.’s *herd* framework [2] to Pulte et al.’s ARMv8 semantics [101]; μ spec extends this tradition to the *microarchitectural* level, where axioms constrain pipeline stages rather than ISA-visible orderings.

Operational models define valid executions via explicit state machines. We formalize these as multi-tape transducers following our FMCAD 2022 paper [46]. An operational model

$\mathcal{M} = (\mathcal{Q}, \Delta, q_{\text{init}}, q_{\text{final}})$ is parameterized by a set of cores $\mathcal{C}$, stages $\mathcal{S}$, and a history parameter $h \in \mathbb{N} \cup \{\infty\}$. The set $\mathcal{Q}$ is a finite set of control states, $\Delta \subseteq \mathcal{Q} \times (\text{Instr} \cup \{-\})^{|\mathcal{C}|} \times \mathcal{Q} \times \text{Act}$ is the transition relation, and $q_{\text{init}}, q_{\text{final}}$ are the initial and absorbing final states. A configuration is a triple $\gamma = (U, q, V)$ where $U, V : \mathcal{C} \rightarrow \text{Instr}^*$ represent the tape contents to the left and right of the head, and $q \in \mathcal{Q}$ is the control state. Actions include $\text{right}(c)$ (advance tape head for core c), stay (silent), $\text{copy}(c, i, s)$ (generate event $i.s$ for core c), and $\text{drop}(c, i)$ (remove instruction i). A run is a sequence of configurations starting from $\gamma_{\text{init}}(P)$; an accepting run ends in q_{final} . The traces of $\mathcal{M}$ on program P , denoted $\text{traces}(\mathcal{M}, P)$, are the event sequences labeling accepting runs, viewed as linear μhb graphs.

A model is *finite-state* if $\mathcal{Q}$ is finite and *bounded-history* if $h \in \mathbb{N}$. For effective verification, we require both: precisely such models can be compiled to RTL and composed with hardware designs for model checking.

The paradigm mismatch is significant. Axiomatic models are often easier to write, a pipeline’s store-buffer forwarding rule is a single axiom, and easier to compare across designs, since axioms are implementation-independent. But the rich ecosystem of operational verification tools (BMC, IC3, PDR) remains inaccessible to axiomatic specifications. We now formalize the conversion problem.

7.2.1 Soundness, Completeness, and the Formal Problem

Let $\mathcal{M}$ be an operational model and $\mathcal{A}$ an axiomatic semantics. For a program P , let $\text{traces}(\mathcal{M}, P)$ denote the set of traces and $G \models_P \mathcal{A}$ denote that μhb graph G is valid under $\mathcal{A}$.

Definition 7.1 (Soundness). $\mathcal{M}$ is *sound* w.r.t. $\mathcal{A}$ if for any program P , each trace in $\text{traces}(\mathcal{M}, P)$ is a linearization of some μhb graph that is valid under $\mathcal{A}$.

Definition 7.2 (Completeness). $\mathcal{M}$ is *complete* if for any program P and valid μhb graph $G \models_P \mathcal{A}$, $\text{traces}(\mathcal{M}, P)$ contains all linearizations of G .

Formal Problem Statement. Given an axiomatic semantics $\mathcal{A}$, a set of cores $\mathcal{C}$ and stages $\mathcal{S}$, generate a finite-state, bounded-history model $\mathcal{M} = (\mathcal{Q}, \Delta, q_{\text{init}}, q_{\text{final}})$ satisfying soundness and completeness.

7.3 Enabling Synthesis by Bounding Reorderings

Can every axiomatic specification be converted to an equivalent finite-state operational model? The answer is no:

Theorem 7.1 (Infeasibility of General Conversion). *For a single-core program P with m instructions, there is no model $\mathcal{M} = (\mathcal{Q}, \Delta, q_{\text{init}}, q_{\text{final}})$ that is sound and complete w.r.t. the axioms $\mathcal{A}^\#$ below and P , and such that $|\mathcal{Q}| < O(2^m/m)$, even with $h = \infty$.*

The axioms $\mathcal{A}^\#$ have two stages **S**, **T** and two rules: (1) $\forall i. \text{hb}(i.\mathbf{S}, i.\mathbf{T})$, requiring each instruction's **S** stage to precede its **T** stage, and (2) $\forall i_1, i_2. \text{hb}(i_1.\mathbf{S}, i_2.\mathbf{S}) \Rightarrow \text{hb}(i_1.\mathbf{T}, i_2.\mathbf{T})$, requiring the **T**-stage ordering to mirror the **S**-stage ordering.

Proof. Consider a single-core program $\mathcal{P}$ with the instruction stream $\mathcal{I} = i_0, \dots, i_{m-1}$. Now consider any permutation π of $\{0, \dots, m-1\}$. Then, the execution represented by $i_{\pi(0)}.\mathbf{S}, i_{\pi(1)}.\mathbf{S}, \dots, i_{\pi(m-1)}.\mathbf{S}, i_{\pi(0)}.\mathbf{T}, i_{\pi(1)}.\mathbf{T}, \dots, i_{\pi(m-1)}.\mathbf{T}$ is a valid execution of the program, and by completeness, is also generated by the operational model. Let the configuration reached by the model after generating the events $i_{\pi(0)}.\mathbf{S}, i_{\pi(1)}.\mathbf{S}, \dots, i_{\pi(m-1)}.\mathbf{S}$ be (U_π, q_π, V_π) . Now suppose that $|\mathcal{Q}| < \mathcal{O}(2^m/m)$. Then by the pigeonhole principle, there are two permutations $\pi_1 \neq \pi_2$ such that $(U_{\pi_1}, q_{\pi_1}, V_{\pi_1}) = (U_{\pi_2}, q_{\pi_2}, V_{\pi_2})$. This holds since the head has only m possible locations on the input tape. Since $i_{\pi_1(0)}.\mathbf{T}, \dots, i_{\pi_1(m-1)}.\mathbf{T}$ is accepted from $(U_{\pi_1}, q_{\pi_1}, V_{\pi_1})$, we get that $i_{\pi_1(0)}.\mathbf{S}, \dots, i_{\pi_1(m-1)}.\mathbf{S}, i_{\pi_2(0)}.\mathbf{T}, \dots, i_{\pi_2(m-1)}.\mathbf{T}$ is accepted by the model. However this execution does not satisfy axiom (2) since $\pi_1 \neq \pi_2$ and hence the ordering of **S** and **T** events must be reversed for some pair of instructions. $\square$

Corollary 7.1. *There does not exist a finite-state operational model (even with $h = \infty$) that is sound and complete with respect to $\mathcal{A}^\#$.*

Corollary follows since we require an operational model with a (fixed) finite state space that is sound and complete for all programs.

This negative result means that conversion requires restricting either the class of axioms or the class of executions. We identify three sufficient conditions.

7.3.1 Refinability

Since operational models produce totally ordered traces (linearizations of the partial-order μhb graphs that axiomatic models reason about), we need linearizations of valid graphs to themselves be valid. For two μhb graphs $G = (V, E)$ and $G' = (V', E')$, we say G' *refines* G , written $G \sqsubseteq G'$, if $V = V'$ and $E^+ \subseteq E'^+$ (the transitive closure of E is contained in that of E').

Definition 7.3 (Refinability). An axiomatic semantics $\mathcal{A}$ is *refinable* if for any program P and μhb graph G such that $G \models_P \mathcal{A}$, we have $G' \models_P \mathcal{A}$ for all linear graphs G' satisfying $G \sqsubseteq G'$.

Without refinability, a valid partial-order execution could have the property that *all* of its linearizations are invalid. Consider the axiom $\forall i_1, i_2. \neg \text{hb}(i_1.S, i_2.S) \wedge \neg \text{hb}(i_2.S, i_1.S)$ with a single stage S . As Figure 7.1 illustrates, the empty-edge graph (a) on two instructions satisfies this axiom (neither event is ordered before the other), but both linearizations (b) and (c) introduce an ordering between the two S events, violating the axiom. This axiom is not refinable.

Most practical μspec axioms are refinable because their consequents require the *presence* of edges rather than their absence: adding edges to a valid graph preserves validity.

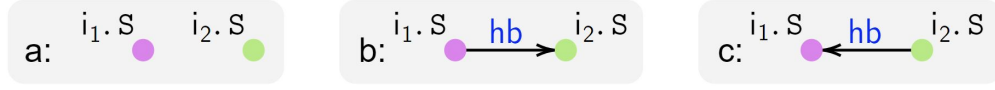


Figure 7.1: A non-refinable axiom. (a) A valid execution graph under the axiom $\forall i_1, i_2. \neg \text{hb}(i_1.S, i_2.S) \wedge \neg \text{hb}(i_2.S, i_1.S)$. (b, c) Both linearizations of (a) are invalid, since each introduces an ordering that the axiom forbids.

Lemma 7.1 (Decidability of Refinability). *Given a universal axiomatic semantics, we can decide whether it is refinable.*

Proof. Consider a universal axiomatic semantics $\mathcal{A}$, a program $\mathcal{P}$ and a graph G , such that $G \models_{\mathcal{P}} \mathcal{A}$ but that does not satisfy refinability, i.e., there is some linearization G' such that $G \sqsubseteq G'$ and $G' \not\models_{\mathcal{P}} \mathcal{A}$. We show that there exists a “small” subgraph G_1 and a linearization $G'_1 \sqsubseteq G'_1$ such that $G_1 \models_{\mathcal{P}_1} \mathcal{A}$ but $G'_1 \not\models_{\mathcal{P}_1} \mathcal{A}$ for a subprogram $\mathcal{P}_1$ of $\mathcal{P}$.

Consider an axiom $\text{AX} = \forall i_1 \dots \forall i_k \phi(i_1, \dots, i_k)$ from $\mathcal{A}$ that the graph G' violates. By the semantics of validity, there must be a set of k instructions $\mathbb{I} = \{i_1, \dots, i_k\}$ such that $\phi(i_1, \dots, i_k)$ does not hold in G' . Let the program comprised of these instructions (maintaining the cores and the $<_r$ order) be $\mathcal{P}_1$. Now we can take the projection of G and G' on the nodes $\{\text{node}(i_j.\text{st}) \mid j \in [k], \text{st} \in \text{Stages}\}$ (and taking a transitive closure on the edges), to get the graphs G_1 and G'_1 , which are executions of the program $\mathcal{P}_1$.

We have that any atoms over the instructions $\mathbb{I}$ hold in G iff they hold in G_1 , and similarly hold in G' iff they hold in G'_1 . This follows since we maintain the $<_r$ orderings between the instructions from $\mathbb{I}$ in program $\mathcal{P}_1$. Hence, $G_1 \models_{\mathcal{P}_1} \text{AX}$ but $G'_1 \not\models_{\mathcal{P}_1} \text{AX}$.

Let K be the largest arity of all axioms from $\mathcal{A}$. This gives us the desired result: if the semantics is not refinable, there exists a “small” program (of size at most K) which is a witness to non-refinability. Then it suffices to search over all programs of size K for a non-refinability witness. These programs are finitely many and hence the procedure terminates. $\square$

7.3.2 t -Reordering Bound

For same-core instructions i_1, i_2 in a trace σ , let $\text{diff}_r(i_1, i_2) = \lambda(i_2) - \lambda(i_1)$ where $\lambda(i)$ is the program-order index of i . For instruction i in trace σ , define $\text{start}(i)$ as the index of the first event of i in σ , $\text{end}(i)$ as the index of the last event, and $\text{pfxend}(i) = \max\{\text{end}(i') \mid i' \leq_r i\}$ as the prefix-closed end index (the latest event among i and all program-order predecessors). Two instructions i_1, i_2 are *coupled* (written $\text{coupled}(i_1, i_2)$) if the intervals $[\text{start}(i_1), \text{pfxend}(i_1)]$ and $[\text{start}(i_2), \text{pfxend}(i_2)]$ overlap.

Definition 7.4 (t -Reordering Bounded Traces). A trace is t -reordering bounded if for any pair of same-core instructions i_1, i_2 : (1) if $i_2.s_2 \xrightarrow{\text{hb}} i_1.s_1$ (out-of-order completion), then $\text{diff}_r(i_1, i_2) < t$; and (2) if $\text{coupled}(i_1, i)$ and $\text{coupled}(i, i_2)$ for some i , then $|\text{diff}_r(i_1, i_2)| < t$.

Condition (1) bounds the depth of reorderings: an instruction cannot be reordered with another that precedes it by $\geq t$ indices. Condition (2) is a starvation-freedom property: instructions on a core cannot be stalled while arbitrarily many instructions execute on other cores. Both are realistic: processors have finite reorder buffers, and practical systems guarantee bounded latency (e.g., through retirement windows and pipeline drain mechanisms).

7.3.3 Extensibility

The t -reordering bound alone is insufficient for bounded-history synthesis. The following example illustrates why.

Example 7.2 (Need for extensibility). Consider a single-stage semantics $\mathcal{S} = \{S\}$ with predicate p and axiom $\forall i_0, i_1, i_2. (p(i_0, i_1, i_2) \wedge i_0 <_r i_1) \Rightarrow \neg \text{hb}(i_1.S, i_0.S)$. For a program with $h + 2$ instructions $i_0 \cdot i_1 \cdots i_{h+1}$, the predicate p may or may not hold for the triple (i_0, i_1, i_{h+1}) , depending on the specific instructions. If it holds, the ordering $i_1.S \xrightarrow{\text{hb}} i_0.S$ is forbidden; if not, it is permitted. A bounded-history model with parameter h must schedule $i_0.S$ before the tape head reaches i_{h+1} , but it cannot determine whether $p(i_0, i_1, i_{h+1})$ holds until i_{h+1} is read. The model is therefore either unsound (always reorders) or incomplete (never reorders).

Extensibility resolves this by ensuring that validity can be checked incrementally.

Definition 7.5 (Extensibility). An axiomatic semantics $\mathcal{A}$ satisfies *extensibility* if for any programs P and P' with $P' \preceq P$ (prefix), and residual $P'' = P \oslash P'$: if $G' \models_{P'} \mathcal{A}$ and $G'' \models_{P''} \mathcal{A}$, then $G' \triangleright G'' \models_P \mathcal{A}$, where $G' \triangleright G''$ composes the two graphs by adding edges from sinks of G' to sources of G'' .

Extensibility ensures that the operational model can process instructions incrementally without unbounded lookahead. Intuitively, validity of a partial execution can be checked by considering only the prefix program, not the entire future.

Lemma 7.2 (Decidability of Extensibility). *Given a universal axiom, we can decide whether it satisfies extensibility.*

7.3.4 t -Completeness and Feasibility

The negative result (Theorem 7.1) rules out full completeness. We relax to *t -completeness*:

Definition 7.6 (t -Completeness). $\mathcal{M}$ is *t -complete* w.r.t. $\mathcal{A}$ if for each program P and $G \models_P \mathcal{A}$, $\text{traces}(\mathcal{M}, P)$ contains all t -reordering bounded linearizations of G .

Theorem 7.2 (Feasibility). *For a refinable universal axiomatic semantics satisfying extensibility, synthesis of finite-state, bounded-history operational models satisfying soundness (Definition 7.1) and t -completeness (Definition 7.6) is feasible.*

The microarchitectural axioms in our case studies (pipeline ordering, store-buffer forwarding, cache coherence) satisfy all three conditions, and we expect this to hold broadly. The restrictions rule out pathological axioms requiring unbounded lookahead or memory, axioms that would not arise from realistic hardware designs. Full proofs of Theorems 7.1 and 7.2 and all supporting lemmas are given in Appendix A.

7.4 Converting to Operational Models Using Axiom Automata

The conversion algorithm constructs an *axiom automaton* for each μspec axiom. The key idea is to reduce axiom compliance checking to language membership in a regular language: fixing the non-hb atoms in an axiom (the *context*) reduces it to a constraint purely on happens-before edges, which a finite automaton can check on-the-fly as events are produced.

7.4.1 Contexts

Let $\text{preds}(a)$ denote the non-hb atoms appearing in axiom a (e.g., $\text{IsRead}(i)$, $\text{SameCore}(i, j)$). A *context* is an assignment $c : \text{preds}(a) \rightarrow \mathbb{B}$. Each concrete instruction assignment s determines a unique context $c(s)$. Fixing a context reduces the axiom to a constraint purely on happens-before edges: the non-hb atoms evaluate to constants, leaving only hb atoms as free variables. This is the key reduction that enables automaton construction.

7.4.2 The Axiom-Automata Lemma

For a k -ary axiom $\forall i_1, \dots, i_k. \phi(i_1, \dots, i_k)$, the symbolic event alphabet is $E_{\text{symb}}(a) = \{i_j.s \mid j \in [k], s \in \mathcal{S}\}$. We write $E_{\text{symb}}(a)^{\text{perm}}$ for the set of all permutations of these symbolic events.

Lemma 7.3 (Axiom-Automata). *Given an axiom a and context c , there exists a finite-state automaton $\mathcal{A}_{a[c]}$ over alphabet $E_{\text{symb}}(a)$ with language*

$$\mathcal{L}(\mathcal{A}_{a[c]}) = \{w \in E_{\text{symb}}(a)^{\text{perm}} \mid s(w) \models \phi(i_1, \dots, i_k)[s] \text{ for all } s \text{ agreeing with } c\}$$

This lemma reduces axiom compliance checking to language membership in a regular language. The automaton for a given context is constructed by enumerating all possible orderings of the (finitely many) symbolic events and accepting those satisfying the axiom's happens-before constraints under that context. Since the number of symbolic events per axiom is $k \times |\mathcal{S}|$, a small constant determined by the axiom's arity and pipeline depth, the resulting automaton has a bounded number of states independent of program length.

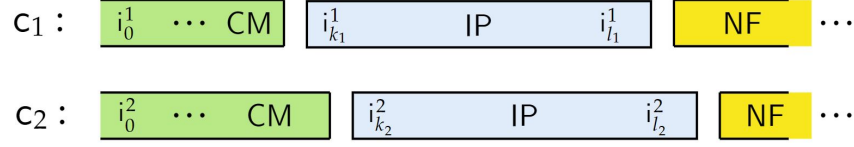


Figure 7.2: Completed prefix (CM), in-progress (IP), and not-fetched postfix (NF) of instructions during execution. Instructions flow left to right through the pipeline; the three sets partition the instruction stream at any point during a trace.

7.4.3 Concretization and Composition

For each concrete instruction assignment $s : \{i_1, \dots, i_k\} \rightarrow I(P)$ (where $I(P)$ is the instruction set of program P), the symbolic automaton $\mathcal{A}_{a[c(s)]}$ is *concretized* by replacing symbolic events with concrete ones, yielding $\mathcal{A}_{a,s}$. The set of all concretized automata for axiom a and program P is $\mathcal{A}_{a,I(P)} = \{\mathcal{A}_{a,s} \mid s : I_{\text{symb}}(a) \rightarrow I(P)\}$. The automata for different assignments and axioms are composed via intersection: the combined automaton accepts an event sequence if and only if all axioms are satisfied under all applicable assignments.

A challenge arises: $|\mathcal{A}_{a,I(P)}| \sim |I(P)|^k$ grows with program size, so the naive composition is not finite-state. The key insight enabling finite-state construction is that only a bounded number of automata are *active* at any point during execution.

7.4.4 Bounding Active Instructions

At any point j during an execution trace, partition the instructions of program P into three sets (Figure 7.2): $\text{CM}(j)$, the completed prefix (all events generated, prefix-closed under $\leq_r$); $\text{NF}(j)$, the not-yet-fetched postfix (no events generated, postfix-closed); and $\text{IP}(j) = I(P) \setminus (\text{CM}(j) \cup \text{NF}(j))$, the in-progress instructions. Instructions i_1, i_2 are *coupled* if their in-flight intervals overlap; k -coupled instructions are those reachable through coupling chains of length k . The k -*active* instructions $\text{AC}_k(j)$ are those from $\text{CM}(j) \cup \text{IP}(j)$ that are k -coupled with some instruction in $\text{IP}(j)$.

Lemma 7.4 (In-progress bound). *For t -reordering bounded traces, $|\text{IP}(j)| \leq |C| \cdot t$.*

Lemma 7.5 (Active instructions bound). *For each k , there is a program-independent bound b_k such that $|\text{AC}_k(j)| \leq b_k$ for all j .*

These bounds are crucial: a k -ary axiom automaton $\mathcal{A}_{a,s}$ is active only when all k instructions in assignment s are active. Since the number of active instructions is bounded by b_k , the number of simultaneously active automata is bounded by $\binom{b_k}{k}$, independent of program length. This yields a finite-state composition.

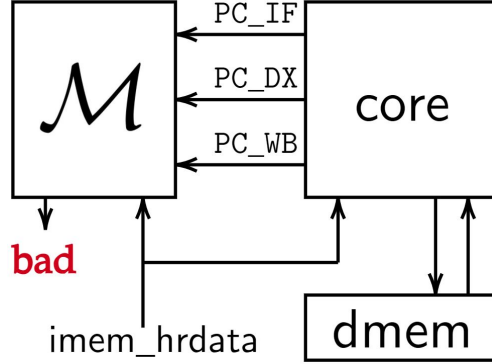


Figure 7.3: Experimental setup. The operational model $\mathcal{M}$ is composed with the hardware design via event-signaling ports (e.g., PC wires). Instruction inputs are symbolically constrained, enabling verification over all programs within the bound.

7.4.5 Deployment

The composed axiom automata are compiled to RTL and synchronously composed with the hardware design, using SymbiYosys [134] with Boolector [91] (BMC) and ABC [14] (PDR) as backend solvers. The axiom automata generation is automatic; the compilation to RTL and composition with the hardware design were performed manually in our experiments. Any violating behavior of the hardware drives the composed automaton into a rejecting state; the safety property $\neg \text{bad}$ is verified by the model checker. This enables both bounded verification (BMC at chosen depth) and unbounded proofs (IC3/PDR), a capability that was not available for axiomatic specifications prior to Ax2Op.

7.5 Case Studies: Ax2Op

We evaluate Ax2Op on three hardware designs—Multi-Vscale [81] (in-order multi-core), Tomasulo [117] (out-of-order), and an SDRAM Controller [61] (memory controller)—using SymbiYosys [134] with Boolector [91] (BMC) and ABC [14] (PDR) as backend solvers, on an Intel Core i7 machine with 16 GB RAM. Figure 7.3 shows the experimental setup: the operational model $\mathcal{M}$ is compiled to RTL and synchronously composed with the hardware design. Event-signaling ports (e.g., program counter wires) exposed from the design drive transitions in $\mathcal{M}$; the safety property $\neg \text{bad}$ asserts that $\mathcal{M}$ never reaches a rejecting state.

7.5.1 Multi-Vscale: Pipeline Axioms

Multi-Vscale [81] is a multi-core extension of the 3-stage in-order Vscale processor. We verify two pipeline axioms on a single core:

$$1 \quad \boxed{\text{ax1: } \forall \text{ i1. } (\text{hb}(\text{i1.Fet}, \text{i1.DX}) \wedge \text{hb}(\text{i1.DX}, \text{i1.WB}))}$$

Table 7.1: Ax2Op: Multi-Vscale and Tomasulo verification results.

Design	Axiom Class	PDR Time	BMC-20 Time
Multi-Vscale	ALU-R pipeline	1m 46s	14m 30s
Multi-Vscale	ALU-I pipeline	2m 11s	11m 31s
Multi-Vscale	Load + Store	2m 18s	13m 35s
Tomasulo	In-order commit + pipeline	~2m (BMC-20)	—
Tomasulo	Dependency ordering (axDep)	Bug found (BMC-20)	—
Tomasulo	axDep (after fix)	~6m 30s (BMC-20)	—
SDRAM Controller	Pipeline + refresh ordering	~8m (PDR)	—

```
2 ax2:  $\forall i1, i2. i1 <_r i2 \Rightarrow \text{hb}(i1.\text{Fet}, i2.\text{Fet})$ 
```

Listing 7.1: Multi-Vscale pipeline axioms

The first axiom requires pipeline stages to proceed in Fetch–DX–WB order; the second enforces in-order fetch. With $h = 3$ and $t = 3$ (matching the 3-stage pipeline), we perform PDR proofs over symbolic instruction streams. Table 7.1 summarizes the results.

7.5.2 Multi-Vscale: Multi-Core ReadValues

We configure Multi-Vscale with 2 cores, both initialized with symbolic load and store operations, and verify the ReadValues axiom:

```
1 RV:  $\forall i1, \exists i2, \forall i3. \text{IsRead}(i1) \Rightarrow$ 
2    $(\text{DataInit}(i1) \vee (\text{IsWrite}(i2) \wedge \text{SameAddr}(i1, i2)$ 
3      $\wedge \text{hb}(i2.\text{DX}, i1.\text{DX}) \wedge \text{ValEq}(i1, i2)$ 
4      $\wedge ((\text{IsWrite}(i3) \wedge \text{SameAddr}(i1, i3)) \Rightarrow$ 
5        $(\text{hb}(i3.\text{DX}, i2.\text{DX}) \vee \text{hb}(i1.\text{DX}, i3.\text{DX}))))$ 
```

Listing 7.2: ReadValues axiom

This axiom states that every read instruction obtains either the initial value or the value of the most recent same-address write, with no intervening write. It is not a universal axiom (it contains $\exists i_2$), so Theorem 7.2 does not apply. However, for bounded programs with $|I|$ instructions, we construct $|I|^2$ concrete automata (two universally quantified variables) and convert the existential into a finite disjunction. Table 7.2 shows the results.

By keeping instructions symbolic, we effectively prove correctness for *all* programs within the bound $|I|$. Though the theoretical results apply to universal axioms, this demonstrates how axiom automata can be applied to arbitrary axioms by bounding $|I|$.

Table 7.2: ReadValues axiom verification on Multi-Vscale (2 cores, symbolic load/store).

$ I $	$ AA $	BMC depth	Time
4	16	12	3m 10s
6	36	16	15m 48s
8	64	20	1h 58m

7.5.3 Tomasulo: Bug Finding

The Tomasulo processor [117] is an out-of-order design with stages Fetch, Dispatch, Issue, Execute, Writeback, and Commit. The dependency-ordering axiom **axDep** enforces that consecutive dependent instructions execute in order:

```

1 axDep:  $\forall i_1, i_2. (i_1 <_r i_2 \wedge \text{Cons}(i_1, i_2)$ 
2  $\wedge \text{DepOn}(i_1, i_2)) \Rightarrow \text{hb}(i_1.E, i_2.E)$ 

```

Listing 7.3: Tomasulo dependency axiom

where $\text{Cons}(i_1, i_2) \equiv \text{pc}(i_1) + 4 = \text{pc}(i_2)$ and $\text{DepOn}(i_1, i_2) \equiv \text{dest}(i_1) = \text{src1}(i_2) \vee \text{dest}(i_1) = \text{src2}(i_2)$.

BMC at depth 20 on the operationalized axiom produced a counterexample revealing a bug in the Register Alias Table (RAT). When instruction i_0 commits, the entry $\text{RAT}[\text{dest}(i_0)]$ is reset. If another instruction i_1 with $\text{dest}(i_0) = \text{dest}(i_1)$ is issued in the same cycle, a third instruction i_2 depending on i_1 reads i_0 's result instead of i_1 's, violating **axDep**. After fixing the RAT reset logic, re-verification completed in $\sim 6\text{m } 30\text{s}$.

7.5.4 SDRAM Controller: Pipeline and Refresh Ordering

To demonstrate the versatility of the approach beyond processor pipelines, we evaluate on an SDRAM Controller [61], which interfaces a processor host with an SDRAM device via a ready-valid interface for read/write requests. We verify pipeline-stage axioms for write (4-stage) and read (5-stage) operations, as well as refresh-ordering axioms ensuring that once a host-level read or write is underway, no refresh operation intervenes before it completes. A PDR-based unbounded proof for the pipeline and refresh axioms completes in $\sim 8\text{m}$ (Table 7.1).

7.6 Related Work

Axiomatic memory models. Axiomatic (declarative) models for memory consistency have been developed at the ISA level [2, 126], the microarchitectural level [78, 82, 80], and the programming-language level [8, 7, 120, 127, 70]. Equivalent operational models have

been constructed by hand for Power [2], ARMv8 [101], release-acquire [69], C++ [92], and TSO [106, 96], with proofs of equivalence conducted manually or via theorem provers. Ax2Op automates this conversion, eliminating most of the manual effort while providing the same soundness guarantees.

Logic-automata connections. At an abstract level, our axiom automata construction is inspired by classic connections between logics and automata [15, 130]. There is a large body of work on synthesizing operational implementations and monitors from temporal specifications, commonly in LTL [86] and its variants [39, 60]. Ax2Op performs a similar conversion but for a different logic: μ spec constrains partial orders over a known set of events, while LTL constrains totally ordered traces with potentially varying event sets. These differences require the novel concepts (contexts, active instructions, t -bounded reordering) developed in this chapter.

Hardware MCM verification. RTLCheck [84] compiles μ spec constraints to SystemVerilog assertions checked per-program. PipeProof [83] performs all-program refinement proofs of a μ spec microarchitectural model against an ISA-level memory consistency specification—the first such automated refinement between a pair of axiomatic models—but its output remains an axiomatic refinement and does not yield an operational artifact for property-directed model checking. RTL2 μ spec [62] performs the reverse conversion, synthesizing μ spec axioms from RTL implementations. Ax2Op complements these: it converts a μ spec specification into an operational automaton that can be composed with RTL and analyzed by IC3/PDR, enabling unbounded property-directed verification.

7.7 Discussion

Prior to Ax2Op, μ spec specifications could be verified in two ways. RTLCheck [84] compiled μ spec constraints to SystemVerilog assertions checked per-program, providing only per-program (and bounded) coverage. PipeProof [83] provided all-program refinement proofs between a μ spec microarchitectural model and an ISA-level memory consistency specification—but the resulting proof object remained axiomatic and could not be fed to property-directed model checkers or composed with RTL for IC3/PDR. Operationalization closes this loop: by producing a finite-state operational counterpart for the axiomatic specification, Ax2Op makes IC3 and PDR available for μ spec, yielding unbounded property-directed proofs that compose with RTL.

The feasibility restrictions (t -reordering bound, extensibility, refinability) are not arbitrary: they correspond to properties that well-designed hardware inherently satisfies (finite reorder buffers, incremental execution). Despite the theoretical generality of the construction, the generated axiom automata have modest state spaces: PDR proofs complete in minutes for multi-core designs.

Chapter 8

PipeSynth: Synthesizing Axioms from Litmus Tests

The previous chapter showed how Ax2Op converts axiomatic μspec specifications into operational models, enabling unbounded verification via IC3 and PDR. But where do the μspec axioms come from? Writing them by hand requires joint expertise in formal methods and microarchitecture, yet architects more naturally express design intent through concrete examples: litmus tests exhibiting forbidden behaviors, and execution traces from simulators showing how the pipeline actually operates. This chapter presents PipeSynth, which synthesizes μspec axioms from these behavioral examples, enabling formal specification *before an RTL implementation exists*.

The material in this chapter is based on our ASPLOS 2023 paper [94]. The reproducibility artifact is archived at <https://doi.org/10.5281/zenodo.7592847>.

8.1 The Pre-RTL Specification Problem

The traditional hardware design workflow discovers bugs late: architects write RTL, verification engineers find bugs, and the design iterates. Formal verification can catch bugs earlier, but only if formal specifications exist. For memory consistency, writing μspec axioms requires deep expertise: the axiom for store-buffer forwarding must precisely capture when a load can read from a store buffer entry versus global memory, accounting for address matching, ordering constraints, and intervening writes. Prior approaches either operate at the ISA level only (MemSynth [11]) or require an RTL implementation (RTL2 μspec [62], ILA [137]), making them inapplicable before the design is committed.

PipeSynth inverts this workflow. It takes three inputs:

1. A **partial μspec specification**: a set of known axioms plus placeholder functions (templates) for the axioms to be synthesized.

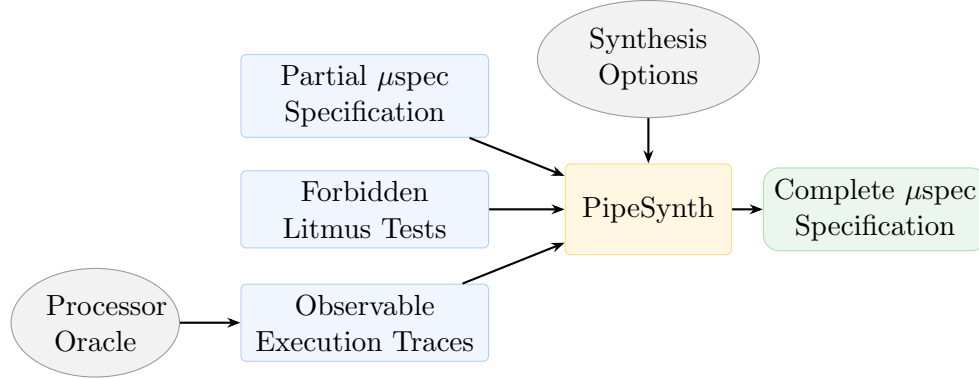


Figure 8.1: PipeSynth block diagram. Rectangles are required inputs; ellipses are optional. The rounded rectangle represents PipeSynth’s output upon successful synthesis.

2. **Forbidden litmus tests:** short programs whose outcomes are not allowed under the target memory model (e.g., “store buffering cannot produce $r1 = 0 \wedge r2 = 0$ under SC”). We generate these using `litmustestgen` [79], which produces all boundary litmus tests up to a bound for a given ISA-level memory model.
3. **Execution traces:** timestamped event sequences from architectural simulators (e.g., `gem5` [10]) or RTL simulators (e.g., `Verilator` [112]) showing *how* the microarchitecture executes allowed behaviors. A typical trace fragment looks like:

```

pc:0x8   stage:IF   t:4   unit:core0
pc:0x1c  stage:IF   t:5   unit:core1
pc:0x18  stage:DX   t:5   unit:core1
pc:0x8   stage:DX   t:6   unit:core0

```

Each line records an instruction (by program counter), the pipeline stage it reaches, the cycle, and the core. These traces provide the synthesis procedure with a witness for permissibility of allowed litmus tests.

Figure 8.1 shows PipeSynth’s high-level organization. From these inputs, PipeSynth synthesizes μspec axioms that are consistent with all provided behaviors: forbidden tests produce cyclic μhb graphs (unobservable), while allowed tests produce acyclic graphs matching the provided traces. PipeSynth encodes the synthesis problem in SyGuS [3] and invokes CVC5 [5] to solve it. If the solver returns a result, a completed specification was found; if it reports failure, no axiom matching the given templates can satisfy all constraints.

8.2 Axiom Templates

PipeSynth searches over three axiom template classes that cover the vast majority of axioms in our evaluated microarchitectures. These three axiom classes were originally identified by

PipeCheck [78].

Path axioms govern ordering within a single instruction's pipeline traversal:

$$\forall i. P(i) \implies \bigwedge_j \text{AddEdge}(i.s_j, i.s_{j+1})$$

where $P(i)$ is a conjunction of predicates on instruction type (not involving **AddEdge**) and the consequent specifies the pipeline stages the instruction traverses. Listing 8.1 shows a concrete example: the **ReadsPath** axiom constrains all read instructions to traverse stages $\text{IF} \rightarrow \text{DX} \rightarrow \text{WB}$ in order.

```

1 Axiom "ReadsPath":
2 forall microops "i",
3   IsRead(i) => (AddEdge((i, IF), (i, DX))
4     /\ AddEdge((i, DX), (i, WB))).

```

Listing 8.1: Path axiom: **ReadsPath** from Multi-Vscale.

FIFO axioms enforce ordering between pairs of instructions:

$$\forall i, j. (\neg \text{SameMicroop}(i, j) \wedge P(i, j)) \implies Q(i, j)$$

where $P(i, j)$ is a conjunction that may include program order, same-core, address predicates, and **AddEdge** (allowing disjunctions of edges via the equivalence $P \Rightarrow (e_1 \vee e_2) \equiv (P \wedge \neg e_1) \Rightarrow e_2$), and $Q(i, j)$ is a conjunction of positive **AddEdge** occurrences. Listing 8.2 shows the **DecodeExecute** axiom enforcing in-order DX stages.

```

1 Axiom "DecodeExecute_stage_is_in_order":
2 forall microops "i",
3 forall microops "j",
4   (~SameMicroop i j /\
5     AddEdge((i, IF), (j, IF))) =>
6     AddEdge((i, DX), (j, DX)).

```

Listing 8.2: FIFO axiom: in-order DX stages from Multi-Vscale.

Sourcing axioms specify reads-from relationships with a double quantifier alternation ($\forall\text{-}\exists\text{-}\forall$):

$$\forall i. P(i) \implies \exists j. (\neg \text{SameMicroop}(i, j) \wedge Q(i, j) \wedge \neg \exists k. R(i, j, k))$$

where Q identifies a valid source write and R specifies intervening writes that would invalidate the sourcing. Listing 8.3 shows the **Read_Values** axiom: every read that does not obtain its value from the initial memory state must have a source write with no intervening same-address write.

```

1 Axiom "Read_Values":
2 forall microops "i",
3   (IsRead(i) /\ ~DataFromInitialStateAtPA(i)) =>

```

```

4  ∃ microop "w",
5  IsWrite(w) /\ ReadsFrom(i, w) /\ (
6    forall microop "w'",
7      WriteOnSameAddr(w, w') =>
8      (AddEdge(w', w) \/ AddEdge(i, w'))
9  ).

```

Listing 8.3: Sourcing axiom: `Read_Values` (details elided for brevity).

These three templates cover 45 of 46 axioms across four evaluated microarchitectures. Some axioms conform to a *combination* of templates despite not appearing to do so. Consider a constraint of the form $(a \wedge F_1) \vee (b \wedge F_2)$: if exactly one of a and b is always true (a *discriminator*), the constraint decomposes into two axioms $(a \Rightarrow F_1)$ and $(b \Rightarrow F_2)$, each of which may fit a built-in template. For instance, the predicates $\{\text{DataFromInitialStateAtPA}(i), \text{SameData}(i, j)\}$ form a discriminator for a load i and a store j : a load either reads from the initial memory state or from some write, but not both. This decomposition explains how axioms that superficially resist template matching are covered. The single uncovered axiom (`Reads` from Peekaboo) requires 4+ quantifier alternations, a pattern not seen in other designs and addressable via custom templates.

8.3 Challenges in μspec Synthesis

Three challenges make μspec axiom synthesis non-trivial.

8.3.1 SyGuS for Quantified Expressions

Virtually all μspec axioms begin with quantifiers (`forall` or `exists`). However, the SyGuS standard (and hence CVC5 [5]) does not allow grammars with quantifiers, and synthesis with quantifiers requires specialized heuristic-based techniques [38]. While finite quantification can in principle be unrolled into conjunctions (for $\forall$) or disjunctions (for $\exists$), all conjuncts must be identical for this to work, requiring context-sensitivity that is impossible for the context-free grammars mandated by SyGuS solvers. PipeSynth solves this by synthesizing only the quantifier-free body of each axiom and specifying the constraint as a finite conjunction/disjunction over the synthesized expression.

8.3.2 Poor Performance of a Naive Encoding

A naive SyGuS encoding—translating the quantifier-free axiom body into a single monolithic synthesis function—produces an infeasible search space. The grammar includes all logical connectives ($\wedge$, $\vee$, $\Rightarrow$, $\neg$) and all μspec predicates, resulting in a tree of expressions with massive redundancy: the solver considers both $\neg a \vee b$ and $a \Rightarrow b$ (semantically equivalent) and both $a \wedge b$ and $b \wedge a$ (order-equivalent). Empirically, a monolithic encoding failed to terminate even after 20 hours for a single Vscale load axiom.

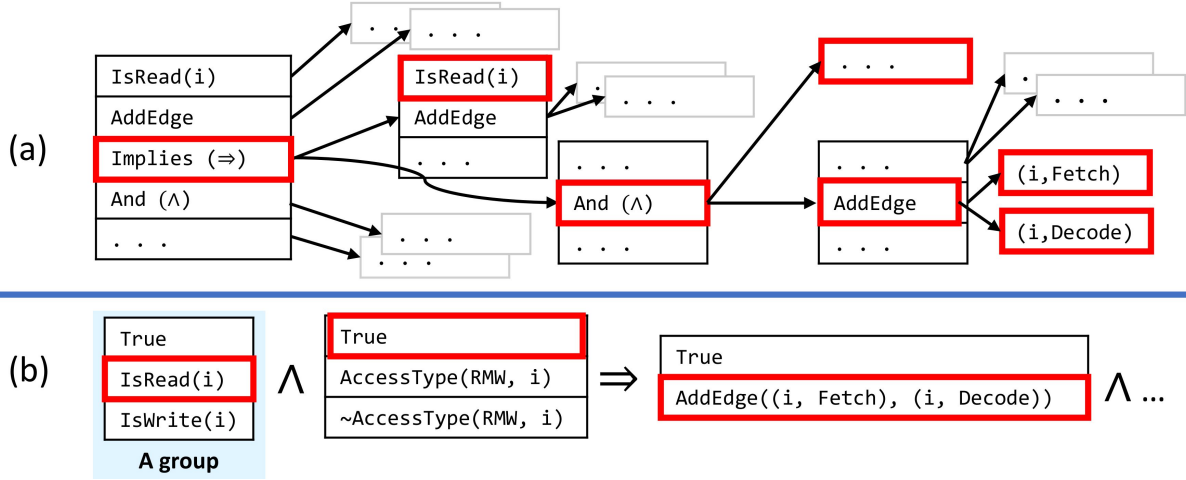


Figure 8.2: (a) A monolithic synthesis encoding contrasted with (b) the structured conjunct-based encoding for a path axiom. Red boxes represent the decisions made by the solver from groups (columns) of black boxes. The conjunct-based encoding prevents the solver from examining redundant and nonsensical candidates (e.g., $\neg a \vee b$ and $a \Rightarrow b$, or $a \wedge b$ and $b \wedge a$), dramatically reducing the search space.

8.3.3 Execution Trace Requirement

Forbidden litmus tests constrain what the microarchitecture *cannot* do but not *how* it executes allowed behaviors. Distinct microarchitectural behaviors (distinct μ hb graphs) may have identical ISA-level observations. Without microarchitectural information, a synthesis procedure could produce axioms that are technically correct but unreasonably restrictive, for example, enforcing that every instruction commits before any subsequent instruction is fetched. PipeSynth solves this by augmenting litmus tests with timestamped execution traces from architectural simulators (gem5 [10]) or RTL simulators (Verilator [112]), ensuring that synthesized axioms are permissive enough to admit the observed behaviors.

8.4 Conjunct-Based SyGuS Encoding

PipeSynth addresses the naive encoding problem (Section 8.3) with a *conjunct-based encoding* that exploits the structure of μ spec axioms. Figure 8.2 contrasts the monolithic and conjunct-based encodings visually. The key insight: both antecedents and consequents of practical axioms are *conjunctions* of predicates drawn from small, well-structured groups. Each conjunct position becomes a separate SyGuS synthesis function whose grammar is the corresponding predicate group; the SyGuS solver selects one predicate per position to form the overall conjunction.

Predicates are organized into *groups* according to two rules that eliminate redundant and

nonsensical candidates from the search space:

1. **Mutually exclusive or implication-related predicates belong to the same group.** For example, $\{\text{IsRead}(i), \text{IsWrite}(i), \text{True}\}$ forms a group because at most one non-trivial predicate can hold for a given instruction. Placing them in the same group ensures the solver never considers conjunctions like $\text{IsRead}(i) \wedge \text{IsWrite}(i)$, which is always false, or $\text{ProgramOrder}(i, j) \wedge \text{SameCore}(i, j)$, where the second conjunct is implied by the first.
2. **The same predicate applied to different instruction variables belongs to different groups.** For instance, $\text{IsRead}(i)$ and $\text{IsRead}(j)$ are placed in separate groups because they constrain independent instructions and can appear simultaneously in a conjunction.

Every group also includes True , which allows the solver to effectively remove a conjunct (since $a \wedge \text{True} = a$), so the number of active conjuncts is determined by the solver rather than fixed in advance. Assigning groups to conjunct positions also eliminates ordering redundancy: the solver examines $\text{IsRead}(i) \wedge \text{AccessType}(\text{RMW}, i)$ but not the semantically equivalent $\text{AccessType}(\text{RMW}, i) \wedge \text{IsRead}(i)$.

Under this grouping, the synthesis problem takes a particularly tractable form: for each conjunct position, select one predicate from the assigned group such that all forbidden litmus tests produce cyclic μhb graphs and all allowed execution traces produce acyclic graphs matching the trace ordering. This encoding reduces the search space by orders of magnitude compared to a naive SyGuS grammar.

8.5 Evaluation

8.5.1 Methodology

We evaluate PipeSynth on four microarchitectures, all implementing TSO except Multi-Vscale (which implements SC): Multi-Vscale [81] (3-stage, in-order, single-cycle memory with arbiter), FiveStage [80] (5-stage with per-core store buffers), OoO-SLR (5-stage with store buffers and speculative load reordering, similar to gem5.uarch from COATCheck [80]), and Peekaboo [80] (5-stage with store buffers, invalidation-before-use, prefetching, livelock prevention, and the ViCL abstraction [82]).

We generate 18 forbidden boundary litmus tests (up to 4 instructions) from ISA-level TSO/SC axioms using `litmustestgen` [79]. For each forbidden test, we create a permitted variant by relaxing one ordering constraint. Execution traces are generated differently per architecture: for Multi-Vscale, we instrument the RTL and use Verilator [112] to produce timestamped event traces. For the other three architectures, we use their complete μspec specifications, run the permitted tests, and use the resulting acyclic μhb graphs as execution traces.

Table 8.1: Single-axiom synthesis breakdown by template and microarchitecture (successes/total, excluding fence axioms). Comb. = combination of templates.

Template	Vscale	FiveStage	OoO-SLR	Peekaboo
Path	2/2	0/0	0/0	1/1
FIFO	7/7	7/7	6/6	10/10
Sourcing	0/0	1/1	1/1	1/1
Comb.	1/1	1/2	1/2	0/1
Other	0/0	0/0	0/0	0/1

Table 8.2: PipeSynth single-axiom synthesis results (excluding fence axioms; 3 fence axioms across FiveStage, OoO-SLR, and Peekaboo are trivially synthesized as empty axioms, yielding 42/46 overall).

Microarchitecture	Axioms Synthesized	Success Rate	Max Time
Multi-Vscale (3-stage, in-order)	10/10	100%	83s
FiveStage (5-stage, store buffer)	9/10	90%	<2 hrs
OoO-SLR (OoO, spec. load reorder)	8/9	89%	<2 hrs
Peekaboo (5-stage, non-trad. coherence)	12/14	86%	45 min
Total (excl. fences)	39/43	91%	

PipeSynth produces SyGuS files processed by CVC5 (version 1.0.2 for solving, with a bugfix for a soundness error in SyGuS solving). Each synthesis query is allowed 2 hours on a cluster node with 16 GB RAM.

8.5.2 Single-Axiom Synthesis

Table 8.2 summarizes single-axiom synthesis results, and Table 8.1 provides a per-template breakdown.

Of the 4 failed axioms, 3 require combining multiple templates (a fixable engineering limitation) and 1 requires 4+ quantifier alternations beyond the template vocabulary. PipeSynth also found 14 axioms to be unnecessary for satisfying the provided litmus tests and traces, synthesizing them as empty (**True**) axioms in under 1 minute each. Some were unnecessary because their features (e.g., fences) were absent from the test suite; others added “convenience” edges that did not contribute to μ hb cycles in any forbidden test.

Figure 8.3 shows single-axiom synthesis times across the four microarchitectures. All Multi-Vscale axioms complete within 83 seconds; FiveStage and OoO-SLR axioms complete within 2 hours; Peekaboo’s longest axiom takes approximately 45 minutes.

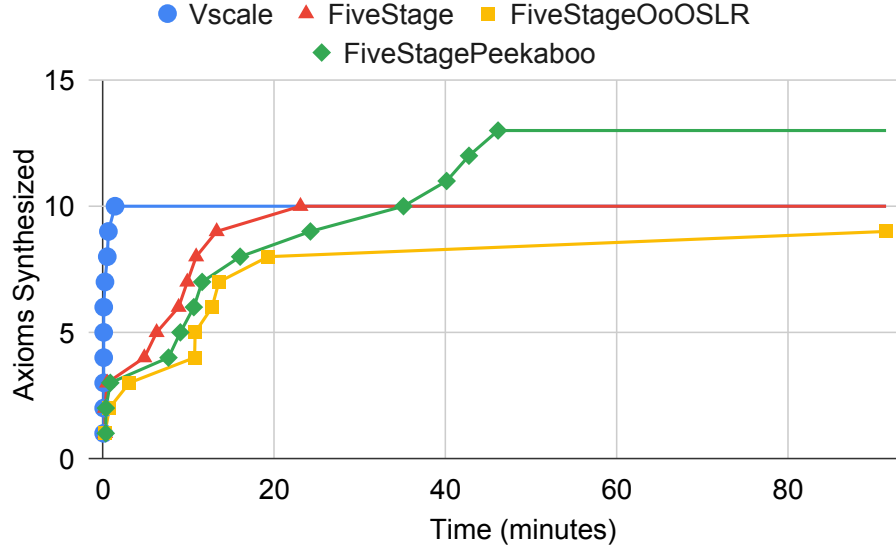


Figure 8.3: Single-axiom synthesis: number of axioms successfully synthesized over time, by microarchitecture. Each query is run independently.

Table 8.3: Multi-axiom synthesis time (minutes:seconds) as axioms are progressively removed. Entries in *italics* are trivial (all conjuncts **True**). T/O = timeout at 2 hours.

# Removed	Vscale	FiveStage	OoO-SLR	Peekaboo
1	<i>0:03</i>	<i>0:12</i>	<i>0:12</i>	<i>0:17</i>
2	<i>0:03</i>	<i>0:12</i>	<i>0:12</i>	<i>0:17</i>
3	<i>0:03</i>	<i>0:10</i>	15:24	<i>0:16</i>
4	0:07	12:03	19:48	27:04
5	0:07	9:13	14:22	47:30
6	0:06	21:05	98:20	52:49
7	1:01	16:13	T/O	106:58
8	1:52	T/O	T/O	69:56
9	2:08	T/O	T/O	90:41
10	58:44	T/O	T/O	T/O

8.5.3 Multi-Axiom Synthesis

Multi-axiom synthesis—synthesizing several axioms simultaneously rather than one at a time—tests PipeSynth’s scalability. Table 8.3 shows synthesis times as axioms are progressively removed from each microarchitecture.

For Multi-Vscale, PipeSynth synthesizes the complete 10-axiom specification in under 59

minutes. For larger designs, synthesis times out beyond 7–9 simultaneous axioms, because the joint search space grows combinatorially. Times do not increase monotonically: removing axioms can produce a more flexible query where solutions need not uphold the exact orderings of the original specification, enabling faster convergence.

The practical strategy is incremental synthesis: synthesize axioms one at a time, fixing previously synthesized axioms as constraints for subsequent rounds. This greedy approach may miss globally optimal axiom sets but scales to all evaluated architectures.

The results validate the pre-RTL workflow: litmus tests and execution traces contain sufficient information to recover the vast majority of formal specification axioms consistent with those examples, with PipeSynth synthesizing 42 of 46 axioms from behavioral examples alone. The conjunct-based encoding is essential for scalability: it exploits the algebraic structure of μspec axioms (conjunctions of predicates from small, well-structured groups), reducing synthesis from infeasible (20+ hours for a naive SyGuS encoding) to tractable (under 2 hours per axiom). This is a recurring Synthesis-Enabled Reasoning (SER, the central methodology of this thesis) principle: domain-specific structure constrains the search space, making synthesis feasible where unconstrained search is not.

8.6 Discussion

Ax2Op (Chapter 7) and PipeSynth address the same fundamental problem — specifications in the wrong form for available tools — but from opposite directions. Ax2Op transforms *down* the abstraction stack, from declarative axioms to operational automata that model checkers can consume. PipeSynth transforms *up* the abstraction stack, from concrete behavioral examples to formal axioms that architects can review and compare across designs. A complementary effort, RTL2 μspec [62], synthesizes μspec axioms directly from RTL implementations rather than from litmus tests; it addresses a related but distinct problem—extracting the model that an implementation *actually realizes*—whereas PipeSynth synthesizes the model that a design *should realize*, from architect-provided behavioral examples. Together, Ax2Op and PipeSynth compose naturally: PipeSynth’s output (μspec axioms) is Ax2Op’s input, forming a complete pipeline from architect intuitions to unbounded verification proofs (Figure 8.4). The combined pipeline demonstrates that the right representation for authoring need not be the right representation for verification; synthesis bridges the gap, enabling each audience to work in its preferred representation.

The feasibility theory (Theorems 7.1 and 7.2) provides a deeper insight: not all representations can be interconverted. The t -reordering bound, extensibility, and refinability requirements characterize the boundary between feasible and infeasible conversion—a boundary that corresponds to the distinction between realistic and unrealizable hardware designs. These theorems illustrate a broader principle: not all representation transformations are possible, and understanding the structural conditions that make them feasible is a recurring theme.

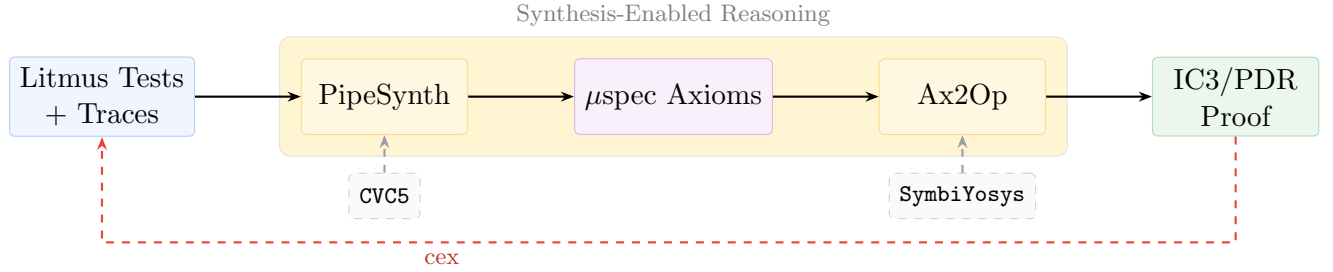


Figure 8.4: PipeSynth + Ax2Op as an end-to-end SER pipeline. Litmus tests and execution traces feed PipeSynth, which synthesizes μ spec axioms. Ax2Op converts these to operational automata for IC3/PDR verification. Counterexamples from verification feed back to refine the axioms, instantiating the synthesizer-verifier pattern (Chapter 2) at the specification level.

Looking forward, PyCaliper (Chapter 9) provides a unified framework in which both axiomatic and operational specifications can be expressed, compiled to verification backends, and subjected to synthesis—further lowering the barrier to the kind of representation transformations that Ax2Op and PipeSynth automate.

Part IV

On Infrastructural Challenges

Chapter 9

Infrastructure for SER: PyCaliper

The preceding chapters presented abstractions (micro-update models, conditional equality invariants, attack patterns) and synthesis techniques (guard synthesis, axiom synthesis, pattern generation) as separate contributions. Each required its own specification format, verification backend, and synthesis infrastructure. This fragmentation reflects the state of the field: formal methods tools are siloed, with each tool requiring specifications in its own format and offering its own interface.

This chapter presents **PyCaliper**, a Python-embedded domain-specific language for RTL specifications that unifies verification and synthesis under a single framework. PyCaliper addresses the infrastructure problem underlying SER: specifications must be consumable by *both* verification engines (to check correctness) and synthesis engines (to generate artifacts). A single specification, written once in Python, compiles to SVA for commercial verification (Cadence Jasper), BTOR2 for open-source model checking (SymbiYosys, Boolector), and CEGIS grammars for invariant synthesis.

A key design principle is *distribution-awareness*: PyCaliper’s Python syntax aligns with what large language models have been trained on, enabling AI-assisted specification synthesis. Because Python is the most widely represented programming language in LLM training corpora [43]—accounting for a substantial fraction of code on GitHub, Stack Overflow, and documentation sites—LLMs have strong priors for generating syntactically and semantically correct Python code, priors that transfer directly to PyCaliper specifications. This positions PyCaliper not just as a verification tool but as infrastructure for the SER paradigm described in Chapter 10.4.

The material in this chapter is based on our CAV 2025 paper [50]. Code is available at <https://github.com/pycaliper/pycaliper>, documentation at <https://pycaliper.github.io>, with a Zenodo snapshot at <https://zenodo.org/records/15163517>.

9.1 The Infrastructure Problem

Formal verification accounts for a substantial fraction of chip development costs [41], yet the tools that verification engineers use are fragmented along multiple axes. Commercial tools use SystemVerilog Assertions (SVA); academic tools such as EBMC [31], Pono [85], and AVR [51] use BTOR2, SMT-LIB, or custom formats. Writing specifications once for use across tools requires manual translation that is error-prone and unsustainable. The problem is compounded by a verification-synthesis disconnect: verification backends check whether a specification holds, while synthesis backends search for specifications that hold—dual tasks that existing tools treat as unrelated, since SVA has no synthesis support and synthesis frameworks cannot consume hardware specifications directly. Finally, SVA’s steep learning curve—its nested implication syntax, implicit clocking, and sequence operators—creates a usability barrier for the software engineers increasingly involved in hardware verification.

PyCaliper addresses these challenges through a single design decision: embed specifications in Python and compile to multiple backends through a shared intermediate representation.

9.2 Language Design

9.2.1 SpecModule and Signal Types

The core abstraction is the `SpecModule`—a Python class corresponding to an RTL module. Each `SpecModule` declares the signals it reasons about, using signal types that mirror SystemVerilog constructs while remaining idiomatic Python:

- `Logic(width)`: bitvector signals (registers, wires)
- `LogicArray(dims, width)`: multi-dimensional arrays (register files, memories)
- `Struct(fields)`: structured signals (SystemVerilog structs)
- `Group(signals)`: hierarchical signal bundles

Signal declarations in a `SpecModule` mirror the RTL module’s ports and internal signals. PyCaliper’s elaboration phase uses Python introspection to resolve hierarchical signal paths (e.g., `tx_fifo.wr_ptr`) to flat RTL references.

9.2.2 Inductive Specification Framework

Specifications in PyCaliper follow a three-component inductive structure:

Definition 9.1 (PyCaliper Specification). A PyCaliper specification (`input`, `state`, `output`) over transition system T_{RTL} is valid iff:

$$\forall S, S'. \text{input}(S) \wedge \text{state}(S) \wedge T_{\text{RTL}}(S, S') \implies \text{state}(S') \wedge \text{output}(S')$$

where `input`(S) specifies global assumptions, `state`(S) specifies inductive state invariants, and `output`(S) specifies assertions.

This formulation directly supports k -inductive proofs: assume the specification holds for k consecutive states, then prove it holds for the $(k+1)$ -th state. The parameter k is exposed to the user via an `unroll(k)` decorator.

9.2.3 Two-Trace Specifications for Security

Security hyperproperties require reasoning about *pairs* of traces. PyCaliper supports this through a *miter* (self-composition) construction with dedicated operators:

- `self.eq(x)`: signal x is equal in both miter copies ($A.x = B.x$)
- `self.when(expr)(x)`: conditional equality ($\text{expr}(A) \wedge \text{expr}(B) \implies A.x = B.x$)
- `self.inv(expr)`: single-trace invariant ($\text{expr}(A)$)

The `self.when` operator directly expresses conditional equality invariants (Definition 3.2), connecting PyCaliper to the DAWG verification framework from Chapter 5.

9.2.4 Synthesis Holes

PyCaliper specifications can contain *holes*—placeholders that synthesis engines fill automatically:

- `ceqhole(expr, signals)`: synthesize a subset $S \subseteq \text{signals}$ such that $\bigwedge_{s \in S} \text{self.when}(\text{expr})(s)$ is inductive. Used for conditional equality invariant synthesis.
- `ctrlhole(ctrl, dep)`: synthesize a lookup table mapping control signal values to dependent signal values. Used for functional specification synthesis.

Holes transform specifications into synthesis problems, much like sketching [114] or syntax-guided synthesis (SyGuS) [3]: the language itself encodes the search space, and synthesis backends fill the holes while verification backends check the result.

9.2.5 Extensibility: Parameterization and Refinement

PyCaliper specifications are reusable across designs through two mechanisms. *Parameterization* allows a single `SpecModule` to be instantiated with varying arguments—for example, the FIFO specification in Section 9.4 accepts depth and element width through a configuration dictionary, so the same class covers FIFOs of any size. Parameterized specifications can be packaged into Python libraries, analogous to Verification IPs but open-source and extensible.

Refinement enables compositional reasoning by relating two specifications under a mapping. This mechanism supports assume-guarantee reasoning [98, 123]: a module’s specification serves simultaneously as a *guarantee* to its clients and an *assumption* for its implementation, so that proofs compose across module boundaries without requiring monolithic verification of the full design. Given specifications $(\text{input}_1, \text{state}_1, \text{output}_1)$ for module M_1 and $(\text{input}_2, \text{state}_2, \text{output}_2)$ for module M_2 , a refinement proof under relation $R(S_1, S_2)$ establishes two verification conditions:

$$\begin{aligned}\Phi_I &\equiv \text{input}_1(S_1) \wedge \text{state}_1(S_1) \wedge R(S_1, S_2) \implies \text{input}_2(S_2) \wedge \text{state}_2(S_2) \\ \Phi_O &\equiv \text{state}_2(S_2) \wedge \text{output}_2(S_2) \wedge R(S_1, S_2) \implies \text{state}_1(S_1) \wedge \text{output}_1(S_1)\end{aligned}$$

That is, the inputs and state of M_2 hold whenever those of M_1 hold under R (Φ_I), and the outputs of M_1 hold whenever those of M_2 hold under R (Φ_O). Concretely, the FIFO specification refines a `Counter` module (with a count signal `ctr` and an increment/decrement operation) under the relation $R \equiv (\text{ctr} = \text{wr_ptr} - \text{rd_ptr})$. This refinement proof, which PyCaliper checks automatically, ensures that any design satisfying the counter specification also satisfies the FIFO specification under the given mapping. PyCaliper currently supports lock-step (synchronous) refinement; stuttering refinement for designs with differing timing behaviors is left for future work.

9.2.6 Design Rationale: Distribution-Awareness

A deliberate design choice is Python embedding rather than a standalone language. PyCaliper draws inspiration from the ease-of-use and flexibility of existing Python-embedded hardware tools: `cocotb` [28] for test-based validation and `MyHDL` [90] for hardware design. However, where `cocotb` targets simulation-based testing and `MyHDL` targets synthesis-oriented design, PyCaliper targets *formal* specification and verification—a distinct niche that neither tool addresses. At the other end of the spectrum, `Kami` [23] provides high-level parametric hardware specification with modular verification proofs in Coq, and `CIRCT` [72] introduces verification dialects as MLIR-based intermediate representations; both are powerful but require expertise in interactive theorem provers or compiler infrastructure that limits adoption among hardware engineers. Beyond usability (IDE support, debugging, package management), Python embedding makes specifications *distribution-aware*: large language models have been trained extensively on Python code, giving them strong priors

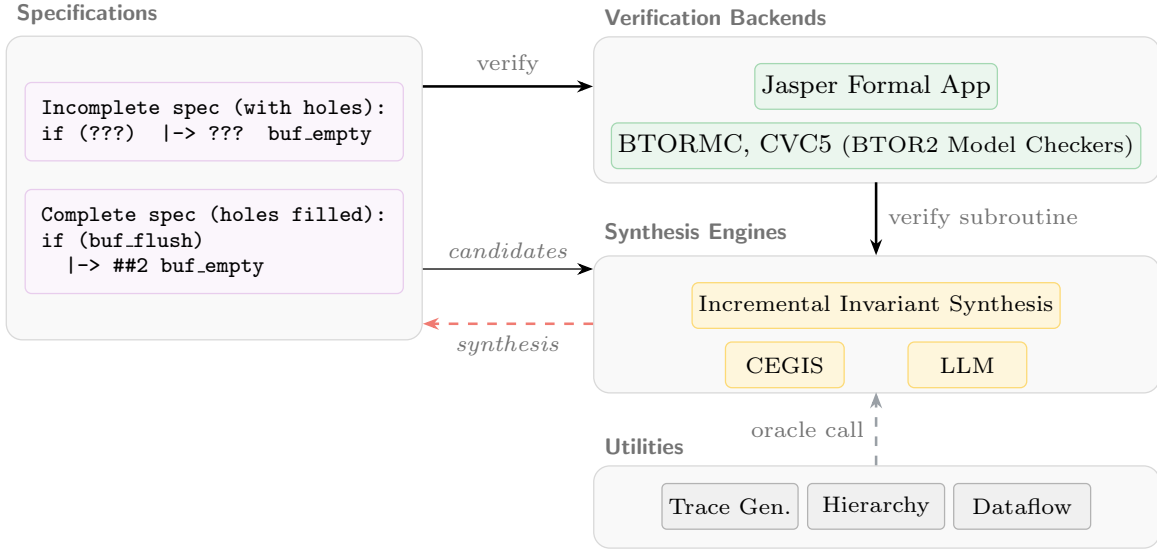


Figure 9.1: PyCaliper infrastructure: specifications (with or without synthesis holes) compile to multiple verification backends. Synthesis engines fill holes via CEGIS or incremental inductive synthesis, using verification as a subroutine. Utilities provide trace generation and design analysis as oracle calls.

for generating and completing PyCaliper specifications. This aligns with the SER vision of Chapter 10.4—LLMs as synthesizers whose outputs are formally verified.

9.3 Multi-Backend Architecture

Figure 9.1 illustrates the overall infrastructure: specifications (with or without holes) compile to multiple verification backends, synthesis engines fill holes via CEGIS or incremental inductive synthesis, and utility modules provide trace generation and design analysis. The compilation is semantics-preserving: the same specification produces equivalent verification queries across backends.

9.3.1 SVA Backend (Commercial)

The SVA backend translates PyCaliper specifications to SystemVerilog Assertions compatible with commercial formal verification tools. The translation generates:

- A proof harness with clocking and reset assumptions

- A counter FSM tracking k -inductive proof steps
- Property encodings: assumptions for cycles $< k$, assertions for cycle k

An oracle interface enables programmatic interaction with the verification engine (Cadence Jasper FPV):

- `enable_assm/disable_assm`: dynamically add or remove assumptions
- `cover`: generate a trace satisfying a condition
- `check/get_trace`: verify a property and extract counterexamples

This oracle interface is essential for synthesis: the CEGIS loop alternates between generating candidates and querying the verification engine for counterexamples.

9.3.2 BTOR2 Backend (Open-Source)

The BTOR2 backend provides a fully open-source verification path. It operates in two stages: first, Yosys [129] synthesizes the RTL design into BTOR2 format—a compact, gate-level representation of the circuit’s transition relation. Second, PyCaliper compiles its specification AST into BTOR2 constraint and property nodes that reference the synthesized circuit’s signals. The combined BTOR2 file is then verified via symbolic execution: the BTOR2 program is unrolled and the resulting SMT formula is checked by Boolector [91]. This fully open-source pipeline enables reproducible research, algorithmic experimentation, and deployment in academic settings where commercial tool licenses are unavailable.

9.3.3 Synthesis Backends

PyCaliper includes two synthesis engines that use the verification backends as oracles.

For `ctrlhole` specifications, *Counterexample-Guided Inductive Synthesis* (CEGIS) synthesizes lookup tables of the form `ite (= ctrl  $c_1$ )  $d_1$  (ite (= ctrl  $c_2$ )  $d_2$  ...  $d_s$ )`, where c_i and d_i are free variables and s is the table size. Algorithm 5 outlines the procedure: traces are extracted from RTL simulation, SMT constraints are formulated over the free variables, and the loop iterates against the verification oracle. Crucially, the algorithm starts with a small lookup table size s and increments it when the SMT query returns UNSAT, indicating that no table of the current size can satisfy the trace constraints. This size-increment strategy avoids searching an unnecessarily large space: most specifications require small tables, and the algorithm discovers the minimal sufficient size automatically. The algorithm guarantees that incorrect candidates are never regenerated, since each counterexample permanently eliminates the specific candidate.

For `ceqhole` specifications, *Incremental Inductive Synthesis* (IIS) implements a fuel-based variant of property-directed invariant generation [12, 32, 13, 97] (Algorithm 6).

Algorithm 5 CEGIS for `ctrlhole` synthesis in PyCaliper**Require:** Spec with holes $h_1, \dots, h_m$, verification oracle $\mathcal{V}$, RTL design D , max size $s_{\max}$ **Ensure:** Hole completions $c_1, \dots, c_m$ such that completed spec is valid

```

1:  $\mathcal{C} \leftarrow \emptyset$ ;  $s \leftarrow 1$  ▷ Constraint set; lookup table size
2: Extract traces  $\mathcal{T}$  from simulation of  $D$ 
3: loop
4:    $(c_1, \dots, c_m) \leftarrow \text{SMT}(\mathcal{C} \cup \text{TraceConstraints}(\mathcal{T}), \text{size} = s)$ 
5:   if UNSAT then
6:      $s \leftarrow s + 1$  ▷ Increment lookup table size
7:     if  $s > s_{\max}$  then return FAILURE
8:     end if
9:     continue
10:  end if
11:   $\text{result} \leftarrow \mathcal{V}.\text{check}(\text{spec}[h_i \mapsto c_i])$ 
12:  if  $\text{result} = \text{PASS}$  then return  $(c_1, \dots, c_m)$ 
13:  end if
14:   $\mathcal{C} \leftarrow \mathcal{C} \cup \{\text{BlockCandidate}(c_1, \dots, c_m)\}$ 
15:   $\mathcal{T} \leftarrow \mathcal{T} \cup \{\text{result.counterexample}\}$ 
16: end loop

```

The central notion is *relative inductiveness*: we write $A \vdash p$ (“ A proves p ”) if p is inductive relative to the conjunction of assumptions in A , that is, if $A(S) \wedge p(S) \wedge T_{\text{RTL}}(S, S') \implies p(S')$. The algorithm maintains sets of assumptions A (candidate invariants currently assumed to hold) and assertions P (invariants proven inductive relative to A), and applies four inference rules: *Dive* adds a candidate invariant to A if fuel remains; *Saturate* proves all assertions p satisfying $A \vdash p$; *Backtrack* removes a failed assumption and explores alternatives; and *Done* terminates when $A \subseteq P$, certifying that the accumulated invariant $\bigwedge P$ is (absolutely) inductive.

The fuel parameter bounds the search depth, trading coverage against conservativeness. Crucially, *Saturate* *restores* fuel: each newly proven assertion increments f by one (line 4), reflecting the intuition that a proven invariant reduces the speculative debt incurred by earlier *Dive* steps. Moreover, the *strategy function* π —which determines which candidate to try next—is pluggable: PyCaliper supports random selection (baseline), heuristic strategies, and neural (LLM-guided) strategies that exploit learned priors over specification structure.

9.4 Illustrative Example: FIFO No-Stale-Data Property

Before turning to the TAGE case study, we illustrate PyCaliper’s basic constructs on a FIFO queue specification. The FIFO is a natural first example: it is small enough to read in full,

Algorithm 6 Incremental Inductive Synthesis (IIS) for `ceqhole`**Require:** Candidate invariants $\mathcal{I}$, property φ , oracle $\mathcal{V}$, fuel f , strategy π **Ensure:** Inductive invariant I such that $I \Rightarrow \varphi$

```

1:  $A \leftarrow \emptyset$ ;  $P \leftarrow \emptyset$  ▷ Assumptions and proven assertions
2: while  $A \not\subseteq P$  do
3:   SATURATE: for  $p \in \mathcal{I} \setminus P$  do
4:     if  $\mathcal{V}.\text{check}(A \Rightarrow p') = \text{PASS}$  then  $P \leftarrow P \cup \{p\}$ ;  $f \leftarrow f + 1$  ▷ Restore fuel
5:   end if
6:   if  $A \subseteq P$  then return  $\bigwedge P$  ▷ DONE
7:   end if
8:   if  $f > 0$  then
9:      $c \leftarrow \pi.\text{select}(\mathcal{I} \setminus (A \cup P))$  ▷ Strategy picks next candidate
10:     $A \leftarrow A \cup \{c\}$ ;  $f \leftarrow f - 1$  ▷ DIVE
11:   else
12:     BACKTRACK: remove most recent dive from  $A$ ; try alternative
13:   end if
14: end while

```

yet exercises signal declarations, inductive specifications, and the three-component structure (input/state/output) described above.

```

1 class FifoSpec(SpecModule):
2     def __init__(self, config):
3         super().__init__()
4         depth = config["DEPTH"]
5         width = config["WIDTH"]
6         ptr_w = (depth - 1).bit_length()
7         self.rd_ptr = Logic(ptr_w)
8         self.wr_ptr = Logic(ptr_w)
9         self.data = LogicArray([depth], width)
10
11     def input(self):
12         pass # No global assumptions
13
14     def state(self):
15         for i in range(self.data.dims[0]):
16             # Entries outside [rd_ptr, wr_ptr) hold default
17             self.pycassume(
18                 ~self.in_range(i) >> (self.data[i] == 0))
19
20     def output(self):
21         # Read output is default when FIFO is empty
22         self.pycassert(

```

```

23         (self.rd_ptr == self.wr_ptr) >>
24         (self.data[self.rd_ptr] == 0))

```

Listing 9.1: FIFO no-stale-data property in PyCaliper. The specification asserts that all entries outside the valid range `[rd_ptr, wr_ptr)` contain a default value, preventing stale data leakage.

The specification in Listing 9.1 declares signal types that mirror the RTL module: `Logic` for the read and write pointers (with width computed from the FIFO depth), and `LogicArray` for the data buffer. The `state` method iterates over buffer entries using a Python loop, generating one inductive assumption per entry during elaboration. The `output` method asserts the no-stale-data property: when the FIFO is empty, the read port returns the default value. This property is critical for verifying data-at-rest protections against attacks that exploit stale microarchitectural state [47].

Because the specification is an ordinary Python class, it benefits from standard language features: the pointer width is computed arithmetically, the buffer depth parameterizes the loop bound, and the entire class can be instantiated with different configurations (e.g., varying depth and width) without modification.

9.5 Case Study: TAGE Branch Target Injection

We demonstrate PyCaliper on the TAGE Branch Target Injection vulnerability (Running Example 2 from Section 3.2.2). This case study addresses predictor isolation: ensuring that predictions for privileged-mode branches are not influenced by unprivileged-mode training. PyCaliper expresses the isolation property in Python and compiles it to both verification (SVA/BTOR2) and synthesis (CEGIS) backends.

9.5.1 Specification

The security property requires that predictions for privileged-mode branches are not influenced by unprivileged-mode training. In PyCaliper:

```

1  class TAGEBtiSpec(SpecModule):
2      BDRY = 0x8000  # valid segment boundary
3      PRIV = 1      # privileged domain
4
5      def input(self):
6          # Privileged targets are in valid segment
7          self.pycassume(
8              (self.mode == self.PRIV) >> (self.targ_i < self.BDRY))
9
10     def state(self):
11         # All priv-domain entries have valid targets

```

Table 9.1: TAGE BTI verification: SVA (Jasper) vs. BTOR2 (Boolector), 2-induction. TT = total time, ST = solving time, PT = parsing time. The Boolector backend’s total time is dominated by parsing the flattened BTOR2 representation (PT column), not SAT solving (ST column).

BHT Width	Jasper		Boolector				
	TT (s)	ST (s)	TT (s)	ST (s)	PT (s)	Stmts	Elms
4	4.0	2.6	3.6	2.6	1	5,465	598
5	3.5	2.5	6.2	2.2	4	9,029	1,046
6	3.9	3.0	16.1	2.1	14	16,142	1,942
7	5.1	4.1	55.8	3.8	52	30,376	3,734
8	7.8	6.9	196.1	6.1	190	58,915	7,318
9	12.7	11.4	753.9	11.9	742	116,186	14,486

```

12     for i in range(self.BHT_WIDTH):
13         self.pycassume(
14             (self.table[i].domain == self.PRIV) >>
15             (self.table[i].targ < self.BDRY))
16
17     def output(self):
18         # Privileged predictions stay in valid segment
19         self.pycassert(
20             (self.prev_mode == self.PRIV) >> (self.targ_o < self.
BDRY))

```

Listing 9.2: TAGE BTI isolation property in PyCaliper

The `state` method iterates over predictor table entries using a Python loop, asserting that all entries tagged with the privileged domain contain targets within the valid segment. This is an inductive invariant: if it holds before a step and the transition preserves domain tagging, then it holds after. The Python loop is unrolled during elaboration, generating one assumption per table entry.

9.5.2 Verification

The specification compiles to both SVA (for Jasper FPV) and BTOR2 (for Boolector). Table 9.1 shows verification performance across predictor table widths.

The ST columns reveal that solving times are comparable between the commercial (Jasper) and open-source (Boolector) backends across all widths: at BHT width 8, Jasper solves in 6.9s and Boolector in 6.1s. The divergence in total time (TT) is entirely due to parsing: Boolector’s PT grows from 1s at width 4 to 742s at width 9, reflecting the com-

Table 9.2: Invariant synthesis for DAWG: random vs. neural strategy (20-run average).

Invariants (k)	Random VCs	Random Nodes	Neural VCs	Neural Nodes
2	16.7 ± 2.3	5.5 ± 0.6	14.3 ± 2.6	4.9 ± 0.9
3	101.6 ± 47.1	13.3 ± 5.5	54.0 ± 21.8	8.5 ± 3.9
4	785.7 ± 91.2	54.3 ± 4.8	163.0 ± 74.2	11.6 ± 8.6

binational blowup from Yosys’s RTL-to-BTOR2 flattening (the Stmts and Elms columns roughly double with each width increment). This decomposition confirms that the open-source path is viable for research-scale designs and that front-end optimizations—not solver improvements—would yield the largest gains.

9.5.3 Invariant Synthesis

We also demonstrate invariant synthesis for the DAWG cache partitioning problem from Section 5. The task: synthesize k out of 8 cache way conditional equality invariants establishing non-interference for dynamically partitioned caches. The problem’s difficulty grows with k because the *invariant-width*—the number of invariants that must be conjoined to form a self-inductive set—increases, creating a combinatorial search space: at $k = 4$, four mutually dependent invariants must be discovered simultaneously.

Table 9.2 compares two IIS strategies: random candidate selection and neural (LLM-guided) selection using GPT-4o. The neural integration required approximately 80 lines of Python: a prompt template describing the specification structure and candidate invariant space, a function calling the OpenAI API with the current assumption/assertion sets as context, and a parser extracting the LLM’s suggested next candidate from its response. The LLM receives the list of signals, current assumptions, and previously failed candidates, and returns a ranked list of candidate invariants to try next.

The neural strategy achieves 2–5 $\times$ reduction in verification calls compared to random selection, with the advantage growing as the number of invariants increases. The reported values are means $\pm$ standard deviation over 20 independent runs. For $k = 2$, the variance overlaps, reflecting the small search space where even random selection converges quickly; for $k \geq 3$, the neural strategy’s advantage is consistent across runs. This demonstrates that PyCaliper’s pluggable strategy interface enables rapid integration of AI-guided search, with the verification backend providing the soundness guarantee.

9.6 Results and Impact

9.6.1 Key Insights

Three aspects of these results merit discussion. First, distribution-aware specification design is a force multiplier for AI-based synthesis: PyCaliper’s Python embedding means that LLMs can generate specification candidates using their strong priors on Python code, and the neural IIS strategy—approximately 80 lines of Python plus 30 minutes of human effort—outperformed the random baseline by 2–5 $\times$, providing early evidence that specification language design directly impacts the effectiveness of AI-assisted formal methods. Second, the multi-backend architecture enables research-to-practice transfer: the same PyCaliper specification, written once, verifies on both Cadence Jasper (an industry-standard commercial tool) and Boolector (an open-source SMT solver), so techniques developed on open-source tools can be deployed industrially without rewriting specifications. The TAGE BTI case study demonstrates this concretely: the specification was developed and debugged using Boolector, then deployed on Jasper for production verification at Intel Labs with no modifications. Third, industry adoption validates that usability drives uptake: PyCaliper has been applied at Intel Labs for secure processor verification, driven not by raw verification power (Jasper already provides that) but by the specification language’s usability and synthesis capabilities—suggesting that the bottleneck in formal methods adoption is infrastructural rather than algorithmic.

9.6.2 Broader Impact

PyCaliper serves as infrastructure for multiple thesis contributions. For micro-update models (Chapter 4), it can express MUM specifications and guards, enabling synthesis of MUM components through the CEGIS backend. For conditional equality invariants (Chapter 5), its `self.when` operator and `ceqhole` holes directly support DAWG-style invariant synthesis. For neural synthesis (Chapter 10.4), its distribution-aware design positions it as the specification language for LLM-based verification workflows.

Example 9.1 (Unified Verification across Property Classes). A realistic processor verification effort requires multiple property classes: speculative non-interference for microarchitectural buffers (Chapter 4), predictor isolation for shared prediction tables (this chapter), and memory consistency for store ordering (Chapter 7). Without PyCaliper, each property would require a different specification language and verification backend. PyCaliper’s multi-backend architecture handles all three in Python: MUM guard synthesis via CEGIS, TAGE isolation via SVA/BTOR2, and consistency checking via operational model compilation.

9.7 Discussion

PyCaliper demonstrates that language design is not peripheral to formal methods but central to their practical adoption. Three observations from this work have broader implications.

The specification language was co-designed with its verification and synthesis backends. Features like synthesis holes and the oracle interface exist because the language was conceived as infrastructure for SER, not just for verification; this co-design yields a system where specifications are both human-readable and machine-consumable.

Verification and synthesis are dual tasks: verification asks “does this specification hold?” while synthesis asks “what specification holds?” PyCaliper unifies them through a common AST in which the same specification fragment can serve as an assertion (verification) or a template with holes (synthesis). This duality is fundamental to SER—every verification problem is reducible to a synthesis problem, and every synthesis result must be verified.

Finally, infrastructure outlives algorithms. Verification algorithms improve incrementally, but specification languages and tool interfaces persist for decades (SVA was standardized in 2005 and remains the industry standard). By providing a well-designed, multi-backend specification language, PyCaliper creates durable infrastructure that future algorithms—including the AI-based approaches explored in [Chapter 10.4](#)—can plug into.

Part V

Conclusion and Outlook

Chapter 10

Conclusion and Outlook

10.1 Summary of Contributions

This thesis advanced a central claim: *important barriers to formal reasoning about hardware-software systems are not primarily algorithmic limitations but infrastructure gaps that can be closed through automated synthesis of verification artifacts*. We introduced synthesis-enabled reasoning (SER) as a methodology—identify the gap, characterize the artifact space using domain structure, synthesize candidates, and verify them—and applied it to bridge three gaps at the hardware-software interface.

Chapters 2 and 3 laid the formal groundwork, introducing the synthesizer-verifier pattern (Table 2.1) and formalizing the verification objectives—Speculative Non-Interference, Conditional Equality, and Memory Model Compliance—that the technical chapters address.

Chapter 4 addressed the *implementation-to-model gap* through micro-update models [48], which synthesize compact abstractions from RTL that preserve security-relevant behavior, achieving up to $8\times$ verification speedup on the Sodor [20] and CVA6 [136] processors. A functional-temporal decomposition separates micro-update bodies (user-specified, requiring local understanding) from guards (synthesized via SyGuS, requiring global understanding), and hierarchical lifting exploits module boundaries to scale synthesis from hours to minutes.

Chapter 5 addressed a complementary facet of the implementation-to-model gap through an abstract resource partitioning framework [49] with conditional equality invariants. By modeling partitioned resources (caches, page-colored memory) through an abstract index-set/allocation-map/policy formalism, and developing invariants that condition policy-state equality on runtime allocation, we enabled unbounded verification of hardware isolation mechanisms such as DAWG [67] cache partitioning, achieving proofs $3000\text{--}4000\times$ faster than bounded model checking (depending on replacement policy).

Chapter 6 addressed the *cross-layer gap*. SemPat [47] synthesizes k -complete attack patterns from hyperproperty specifications, bridging microarchitectural semantics to software-level analysis. A two-phase algorithm—template generation via taint-guided DFS followed by counterfactual-based constraint specialization—produces pattern sets that are provably

complete up to skeleton depth k . SemPat discovered two previously unknown Spectre-class patterns involving computation reuse, absent from all prior manual enumerations, and enables 20–120 $\times$ faster verification through pattern matching.

Chapter 7 addressed the *representation-to-tool gap* through Ax2Op [46], which converts axiomatic memory consistency models to operational automata. The construction proceeds through axiom automata—finite-state machines whose language is the set of valid event orderings under a given context—composed via intersection and bounded by the number of active instructions. This unlocks IC3 and PDR for happens-before specifications, and discovered a bug in an out-of-order processor’s register alias table.

Chapter 8 completed the representation-to-tool pipeline through PipeSynth [94], which synthesizes μ spec axioms pre-RTL from litmus tests and execution traces. A novel conjunct-based SyGuS encoding exploits the algebraic structure of μ spec axioms (conjunctions of predicates from small, well-structured groups), reducing synthesis from infeasible (20+ hours for a naive encoding) to tractable (under 2 hours per axiom), recovering 42 of 46 axioms across four microarchitectures.

Chapter 9 provided the unifying infrastructure. PyCaliper [50] is a Python-embedded DSL that compiles to both verification (SVA, BTOR2) and synthesis (CEGIS, incremental inductive synthesis) backends, enabling a single specification to serve both purposes. Its distribution-aware design—Python embedding aligned with LLM training distributions—enables AI-assisted specification synthesis, with neural strategies achieving 2–5 $\times$ speedup over random search for non-trivial invariant counts.

Example 10.1 (Contributions in Context). Consider the task of verifying a RISC-V processor for both security and memory consistency. MUMs abstract speculative behavior of microarchitectural buffers, enabling scalable security analysis (implementation-to-model, Chapter 4). Conditional equality invariants prove that partitioned caches and predictors enforce domain isolation (Chapter 5). SemPat synthesizes k -complete attack patterns from non-interference specifications, discovering new Spectre-class patterns including those exploiting cache and computation-reuse side channels (cross-layer, Chapter 6). Ax2Op and PipeSynth ensure that the processor’s memory ordering behavior matches its axiomatic specification, verified via IC3/PDR (representation-to-tool, Chapters 7 and 8). PyCaliper expresses all three property classes in a single Python framework. No single contribution solves the full problem; together, they address the major facets of processor verification.

Figure 1.1 (Chapter 1) provides a visual summary of how the contributions relate.

10.2 Impact

The contributions advance the state of practice in four ways.

MUM abstractions reduce verification time by up to 8 $\times$, making security analysis of application-class processors practical within existing computational budgets. Hierarchical synthesis further extends reach by enabling synthesis for designs where monolithic approaches

time out. The functional-temporal decomposition—user provides the “what” (micro-update bodies), synthesis determines the “when” (guards)—distributes the modeling burden in a way that aligns with how hardware engineers understand their designs.

SemPat’s automated pattern generation discovered two Spectre-class attack patterns absent from all prior manual enumerations [19, 34], confirming that synthesis can complement—and in some cases surpass—manual expert analysis. The k -completeness guarantee provides a formal coverage bound that manual enumeration cannot offer: if no pattern in a k -complete set matches a program, that program satisfies non-interference for all violation skeletons up to depth k .

Conditional equality invariants achieve unbounded proofs three orders of magnitude faster than bounded model checking, a qualitative improvement from bounded to unbounded guarantees that transforms overnight computations into sub-second results. The abstract resource model generalizes across partitioning schemes—hardware way-partitioning and OS-level page coloring verified in the same framework—demonstrating that the formalism captures the essential structure shared by diverse resource partitioning mechanisms.

PyCaliper’s Python embedding and multi-backend architecture lower the barrier to formal methods adoption. Intel Labs’ adoption of PyCaliper [50] was driven by the specification language’s usability and synthesis capabilities rather than raw verification power, suggesting that the bottleneck in formal methods adoption is infrastructural rather than algorithmic. The distribution-aware design positions PyCaliper as infrastructure not just for classical verification but for the emerging paradigm of AI-assisted formal methods.

10.3 Limitations

Each SER contribution has scope limitations that merit honest assessment.

Micro-update models require manual identification of signals of interest and a user-specified micro-update library. While the library need not be exact—irrelevant micro-updates are filtered, and incomplete libraries produce diagnostic failures—automating signal selection, potentially via LLM-guided analysis of RTL documentation, remains an open challenge. The bounded model checking used for equivalence verification provides only a bounded guarantee; extending to unbounded equivalence proofs would strengthen the formal assurance.

SemPat generates patterns that are k -complete for a fixed depth k , so attacks requiring longer instruction sequences may be missed. The grammar determines pattern precision: coarser architectural grammars produce portable patterns with more false positives, while finer microarchitectural grammars reduce false positives at the cost of portability. Generation time scales exponentially with grammar depth beyond the minimum required, and solver scalability limits word widths on the more complex platforms (store-to-load forwarding timed out at 24-bit word width).

Conditional equality invariants require the user to specify the filtering function that determines which policy state elements must agree—a step that involves domain expertise for

each new partitioning scheme. Synthesizing the filter function automatically is an attractive direction that would further reduce the manual burden.

Of PipeSynth’s 4 unsynthesized axioms (out of 46), 3 require combining multiple templates and 1 requires 4+ quantifier alternations; both limitations are addressable through template extensions. The conjunct-based encoding assumes that axiom antecedents and consequents are conjunctions of predicates, which covers the vast majority of practical axioms but may not extend to more exotic specification patterns.

Ax2Op assumes that traces are t -reordering bounded and extensible, conditions that correspond to realistic hardware (finite reorder buffers, incremental execution) but restrict applicability to well-behaved designs. The infeasibility theorem (Theorem 7.1) confirms that these restrictions are necessary, not merely convenient.

PyCaliper does not yet support unbounded-past temporal operators, and the IIS synthesis algorithm’s effectiveness depends on the fuel parameter and strategy function. The BTOR2 backend’s parsing time grows rapidly with design size due to Yosys’s combinational flattening, limiting the open-source path for large industrial designs.

More broadly, SER is effective when domain structure constrains the synthesis search space. For domains where the artifact space lacks exploitable structure—where the “right” model or invariant has no compact characterization—SER may not outperform exhaustive search.

10.4 Outlook: Synthesis-Enabled Reasoning in the Age of AI

The preceding chapters established synthesis-enabled reasoning as a methodology: synthesize verification artifacts—abstractions, patterns, operational models, invariants—to bridge gaps between systems and formal tools. Each contribution used *classical* synthesis (CEGIS, SyGuS, template-based search) with domain-specific structure constraining the search space. This section looks forward, exploring how large language models (LLMs) transform the SER landscape—both as synthesizers within the framework and as consumers of SER-generated specifications—and argues that specification synthesis is well-positioned to become an important capability for trustworthy AI-assisted software development.

10.4.1 LLMs as Specification Synthesizers

The synthesizer-verifier pattern introduced in Chapter 2 decomposes synthesis into candidate generation and verification. This decomposition is naturally suited to LLMs: the model generates candidates (exploiting learned priors over code and specifications), and formal verification validates them (exploiting soundness guarantees). Incorrect candidates are rejected with counterexamples, which can be fed back to the LLM for refinement—the *verify-after-generate* paradigm.

Classical synthesis (CEGIS, SyGuS) searches over a grammar-defined space using constraint solvers; LLMs search over a distribution-defined space using learned priors. The key advantage is that LLMs exploit learned statistical regularities in code and specifications to propose candidates that are likely correct, rather than exhaustively enumerating a grammar. When the LLM’s training distribution aligns with the target domain, synthesis converges faster than constraint-based methods.

Chapter 9 introduced PyCaliper’s Python embedding as a concrete instance of *distribution-aware* specification design: because LLMs have been trained extensively on Python code, they have strong priors for generating PyCaliper specifications. The neural IIS strategy (Table 9.2) demonstrated this concretely: an LLM-guided search achieved 2–5× fewer verification calls than random search for invariant synthesis on DAWG cache partitioning. The specification language itself becomes a design parameter influencing synthesis effectiveness.

This verify-after-generate paradigm is gaining traction across the formal methods community. Clover [115] applies closed-loop LLM generation with Dafny verification to produce provably correct code, and Baldur [40] demonstrates that LLMs can generate and repair entire proofs in Isabelle/HOL, achieving substantial automation of what was previously manual theorem proving. These efforts, like PyCaliper’s neural IIS strategy, share a common architecture: the LLM proposes, the formal tool disposes.

More broadly, the verify-after-generate paradigm offers a fundamental architectural advantage: LLMs contribute *speed* (good candidates quickly) while formal methods contribute *soundness* (only correct candidates accepted). Neither alone suffices; together they enable practical formal verification at scales that neither classical synthesis nor unverified LLM generation can achieve independently.

10.4.2 Specifications as Guides for Code Generation

The SER methodology also applies in the reverse direction: rather than using LLMs to synthesize specifications, specifications can guide LLM-generated code. Syzygy [109], presented at the LLM4Code Workshop at ICSE 2025, explores this direction in the context of C-to-Rust translation, where the gap between C’s permissive memory model and Rust’s ownership discipline creates systematic translation challenges.

When an LLM translates C code to Rust, it must infer programmer intent from syntactic patterns: which pointers can alias, what are the allocation bounds, which fields are mutably accessed. These are precisely the questions that formal specifications answer. Without specifications, the LLM guesses—sometimes correctly, sometimes not.

Syzygy addresses this through *dynamic specification synthesis*: rather than asking users to write specifications manually, it extracts lightweight behavioral contracts—allocation bounds, pointer mutability constraints, data structure invariants—from C program executions. These contracts are not full functional correctness proofs but rather specifications that capture enough of the program’s intent for the LLM to produce a correct Rust translation. The architecture has four components: SpecMiner extracts contracts from dynamic analysis, CodeGenerator uses an LLM to produce Rust translations guided by the contracts,

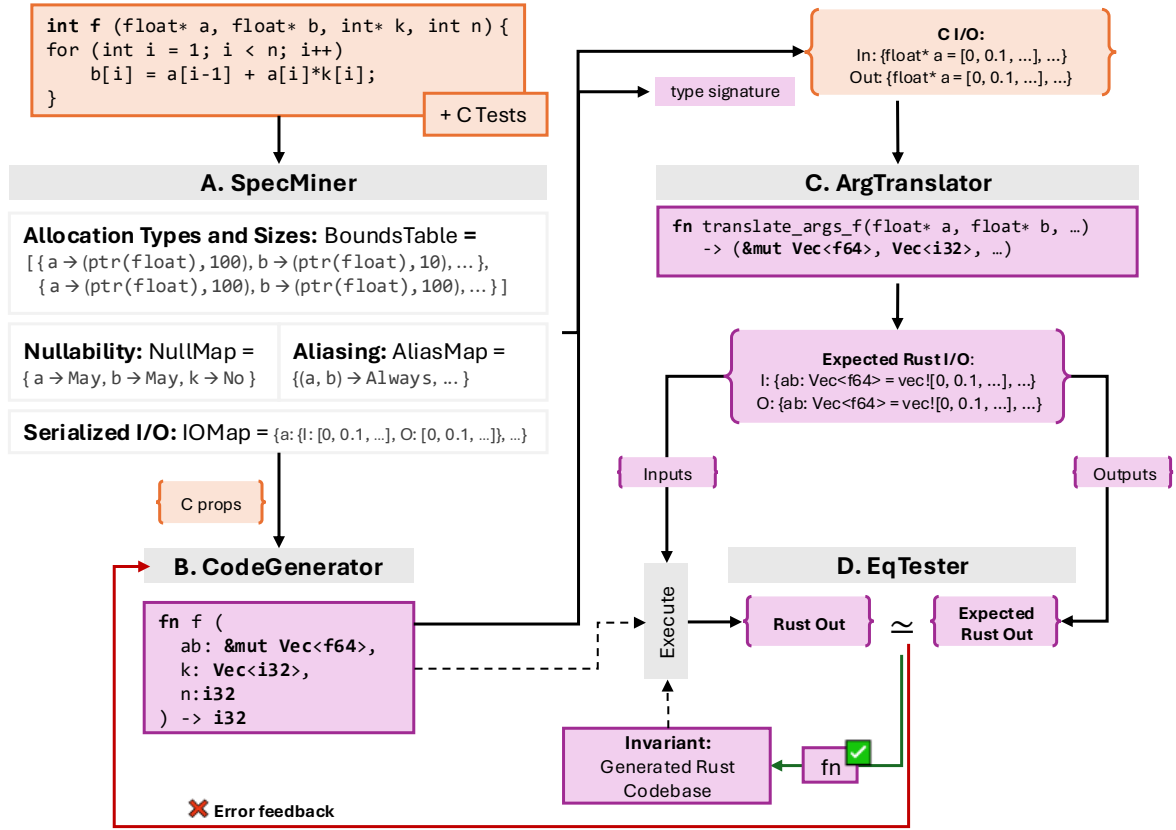


Figure 10.1: Syzygy architecture. **A. SpecMiner** extracts behavioral contracts (allocation bounds, aliasing, nullability) from C program executions. **B. CodeGenerator** uses an LLM to produce Rust translations guided by the contracts. **C. ArgTranslator** converts C-to-Rust type signatures and I/O. **D. EqTester** validates behavioral equivalence; failures feed back to the code generator for iterative refinement.

ArgTranslator converts C-to-Rust type signatures, and EqTester validates behavioral equivalence with failures feeding back for iterative refinement (Figure 10.1).

Syzygy was evaluated on the Zopfli compression library (3,000+ lines of C, 98 functions), achieving 75% compilation and execution pass rates with 88% line coverage—among the largest test-validated automated C-to-safe-Rust translations reported as of early 2025. As a secondary case study, it solved the UrlParser benchmark (400 lines of C), previously unsolved by prior C-to-Rust tools. Existing approaches to this translation problem span a spectrum: Emre et al. [33] apply static analysis to lift unsafe Rust output from C2Rust into safer patterns, Zhang et al. [138] use ownership analysis to guide the translation, and VERT [132] employs LLMs as few-shot transpilers with equivalence checking. These tools operate at the function or statement level; Syzygy’s distinguishing contribution is repository-scale transla-

tion guided by dynamically synthesized specifications, which provides the LLM with richer semantic context than static type information alone.

Syzygy instantiates the SER pattern in a new domain: the “gap” is between C semantics and Rust’s ownership model, the “synthesis” produces behavioral contracts from dynamic analysis, and the “reasoning” uses these contracts to guide correct translation. The formal artifacts (specifications) bridge a semantic gap, just as attack patterns bridge the hyper-property gap (Chapter 6) and operational models bridge the paradigm gap (Chapter 7).

10.4.3 The Case for Formal Compositional Reasoning in AI-Assisted Development

The experiences with PyCaliper and Syzygy point toward a broader thesis: as LLMs become integral to software development—generating code, translating between languages, refactoring legacy systems—specification synthesis is poised to play an important role in ensuring that AI-generated artifacts are correct.

Current evaluation of LLM-generated code relies heavily on test suites and heuristic metrics (BLEU scores, exact match). These methods assess surface plausibility but provide limited guarantees of correctness, failing to detect subtle errors in boundary conditions, security-sensitive code paths, or cross-language semantic mismatches. In high-assurance domains, this gap poses serious risks: a plausible but incorrect translation of a cryptographic library can silently compromise security.

A more principled approach treats correctness as a *spectrum of layered obligations*. At the top level, functional equivalence requires that source and target produce identical outputs for all inputs. Internally, function contracts and loop invariants must be preserved, enabling functions to be verified independently and composed. Beyond functional equivalence, translations must preserve non-functional properties: ABI conventions, resource bounds, memory safety, and timing guarantees such as constant-time execution.

Specification synthesis plays a dual role in this vision. First, specifications serve as *validity definitions*: they make explicit what “correct” means for a given translation, replacing implicit assumptions with formal obligations. Second, specifications serve as *feedback signals*: verification outcomes (pass/fail, counterexamples) guide the LLM toward correct candidates, either through online constrained decoding or offline iterative refinement.

The key enabling technology is *compositional* specification and verification. The assume-guarantee paradigm [98, 123]—where each module is verified under assumptions about its environment that other modules’ guarantees discharge—is well-established in hardware verification (Chapter 9) and has deep roots in software verification through Hoare-style function contracts. Monolithic formal verification does not scale to realistic codebases; breaking problems into smaller components with explicit contracts enables proof assembly. LLMs can both consume specifications (using contracts as translation guides, as in Syzygy) and produce them (synthesizing invariants and contracts from code, documentation, and dynamic analysis).

This positions the specification synthesis capabilities developed in this thesis—PyCaliper’s distribution-aware design, the synthesizer-verifier pattern, the verify-after-generate paradigm—as potentially applicable beyond hardware verification, toward the broader challenge of trustworthy AI-assisted software development.

10.5 Open Problems and Future Directions

Several research directions emerge from the intersection of SER, formal methods, and AI.

Automating intent discovery. Current SER workflows require human-specified inputs: signals of interest (Chapter 4), litmus tests (Chapter 8), predicate grammars (Chapter 6). These represent design intent at an appropriate level of abstraction. LLMs may be able to infer intent from documentation, code comments, commit histories, and design specifications, further automating the SER pipeline.

Composing SER contributions into a fully automated pipeline. Each technical chapter addresses a single gap in isolation, with human expertise providing the interfaces between steps. A fully automated pipeline would compose these steps end-to-end—discover signals of interest, synthesize abstractions, generate attack patterns, verify against patterns—with LLMs serving as the “glue” connecting steps by interpreting intermediate results and formulating inputs to subsequent tools.

SER beyond hardware. The three-gap framework developed in this thesis is not specific to hardware. Software verification faces analogous gaps: program analysis models abstract away runtime behavior such as garbage collection and thread scheduling (a implementation-to-model gap); security properties span application code and operating system layers (a cross-layer gap); and specifications written in one formalism are often incompatible with tools designed for another (a representation-to-tool gap). The SER methodology—synthesize domain-informed verification artifacts to bridge these gaps—applies wherever these structural mismatches arise. Early evidence from Syzygy [109], which applies specification synthesis to C-to-Rust translation, suggests that the methodology transfers effectively to software domains.

Scaling formal methods via AI. If LLMs can generate specifications, select abstractions, and guide synthesis with minimal human effort, formal verification may become accessible to a broader class of engineering projects. PyCaliper’s distribution-aware design is a step toward this vision, but significant challenges remain in specification quality, domain adaptation, and trust calibration.

Hybrid classical-neural synthesis. The optimal division of labor between classical and neural synthesis remains an open question. LLMs generate initial candidates exploiting semantic understanding, while CEGIS/SyGuS refines them exploiting formal guarantees. PyCaliper’s pluggable strategy interface supports this hybrid approach, and the neural IIS results suggest that even simple LLM integration yields measurable gains—but the full potential of tightly coupled classical-neural synthesis is unexplored.

The SER–AI virtuous cycle. More speculatively, SER and AI may form a virtuous cycle. AI advances make synthesis faster and more applicable—LLMs generate specification candidates that classical methods would take hours to enumerate. But SER also benefits AI: formal specifications generated by SER provide training data, evaluation benchmarks, and correctness guarantees for AI-generated code. As AI systems become more capable, the need for formal guarantees grows; as formal methods become more automated, the cost of providing those guarantees falls. The specification synthesis capabilities developed in this thesis—PyCaliper’s distribution-aware design, the synthesizer-verifier pattern, the verify-after-generate paradigm—are infrastructure not just for hardware verification but for this broader vision of trustworthy AI-assisted development.

The thesis opened with the observation that hardware-software systems rest on a tower of abstractions, and that the gaps between those abstractions rarely receive systematic formal scrutiny. The work presented here demonstrates that these gaps can be identified, characterized, and closed through synthesis—producing verification artifacts that are sound by construction and practical to deploy. The methodology generalizes across the hardware-software interface domains studied here; the infrastructure is in place; and the integration with AI is underway. What remains is to extend the reach of synthesis-enabled reasoning to the full breadth of systems that depend on abstractions for their correctness and security.

Appendix A

Proofs for Ax2Op Operationalization

This appendix provides full proofs for the theorems and lemmas stated in Chapter 7. All proofs are from the extended version of our FMCAD 2022 paper [46].

A.1 Axiom-Automata Lemma (Lemma 7.3)

Proof. We show that if a word $w \in E(AX)^{\text{perm}}$ and $s(w) \models \phi(i_1, \dots, i_k)[s]$ for some assignment s that agrees with context cxt , then $s'(w) \models \phi(i_1, \dots, i_k)[s']$ holds for all assignments s' that agree with cxt .

We claim that $s(w) \models \psi[s] \iff s'(w) \models \psi[s']$ whenever s, s' have the same context w.r.t. ϕ for all sub-formulae ψ of ϕ . We show this by structural induction over ϕ . The base case follows since (1) for non-hb atoms, s, s' having the same context means that their interpretation is identical and (2) for hb atoms, we have $s(i_a).\text{st}_a \xrightarrow{\text{hb}} s(i_b).\text{st}_b$ in $s(w)$ whenever we have $s'(i_a).\text{st}_a \xrightarrow{\text{hb}} s'(i_b).\text{st}_b$ in $s'(w)$. The inductive case follows trivially.

Hence, we simply need to determine words from $E(AX)^{\text{perm}}$ such that $s(w) \models \phi[s]$ for some particular s which agrees with cxt . But this follows since $|E(AX)|$ is finite, and we can check these words one at a time. $\square$

A.2 Checking Extensibility (Lemma 7.2)

We prove an intermediate lemma that implies Lemma 7.2. Given a universal axiom $AX \equiv \forall i_1 \dots \forall i_k, \phi(i_1, \dots, i_k)$, and a context cxt for AX , we say that a subset $I' \subseteq I(AX)$ is *prefix closed* if whenever $i_j \in I'$ and atom $i_i <_r i_j$ in AX with $\text{cxt}(\text{atom}) = \text{true}$, then $i_i \in I'$.

Lemma A.1. *The following are equivalent for a universal axiomatic semantics $\mathcal{A}$:*

- $\mathcal{A}$ satisfies extensibility.

- For every axiom AX in $\mathcal{A}$, and consistent context cxt , for all prefix closed $\emptyset \subset I' \subset I(AX)$, the language $\mathcal{L}$ of $\text{aa}(AX, \text{cxt})$ satisfies

$$\{\text{i.st} \mid \text{i} \in I', \text{st} \in \text{Stages}\}^{\text{perm}} \cdot \{\text{i.st} \mid \text{i} \in I(AX) \setminus I', \text{st} \in \text{Stages}\}^{\text{perm}} \subseteq \mathcal{L}$$

It is easy to see how this proves Lemma 7.2: since the set of contexts and prefix-closed subsets is finite, one can check for inclusion by enumeration.

Proof. $\implies$ Assume that the universal axiomatic semantics $\mathcal{A}$ is extensible. Consider an axiom $AX \in \mathcal{A}$ of arity k with $I(AX) = \{i_1, \dots, i_k\}$. Consider any consistent context cxt . Since the context is consistent, there exists an assignment $s : I(AX) \rightarrow \mathbb{I}$ that agrees with cxt . Consider the program $\mathcal{P}$ consisting of instruction streams from $\{s(i) \mid i \in I(AX)\}$ (preserving the $<_r$ order for instructions on the same core). This program has $|\text{instrsOf}(\mathcal{P})| = k$ instructions.

Now consider any nonempty proper subset $\emptyset \subset I' \subset I(AX)$ that is prefix closed w.r.t. cxt and consider the instructions from $\mathcal{P}$ corresponding to I' under s . Since I' is prefix closed, the program corresponding to these instructions is a prefix program $\mathcal{P}'$ of $\mathcal{P}$ with $0 < |\text{instrsOf}(\mathcal{P}')| < k$. Consequently the residual program $\mathcal{P}'' = \mathcal{P} \oslash \mathcal{P}'$ also satisfies $0 < |\text{instrsOf}(\mathcal{P}'')| < k$. By the semantics of quantification (we need distinct assignments), any executions for $\mathcal{P}'$ and $\mathcal{P}''$ are valid w.r.t. AX and by refinability, any linearized executions are also valid. This implies that all words in $\{\text{i.st} \mid \text{i} \in I', \text{st} \in \text{Stages}\}^{\text{perm}} \cdot \{\text{i.st} \mid \text{i} \in I(AX) \setminus I', \text{st} \in \text{Stages}\}^{\text{perm}}$ represent valid executions and must be contained in $\mathcal{L}$.

$\Leftarrow$ Now assume that the second condition holds. Consider a program $\mathcal{P}$ and a prefix/residual: $\mathcal{P}', \mathcal{P}'' = \mathcal{P} \oslash \mathcal{P}'$ respectively. Consider valid executions G', G'' of $\mathcal{P}', \mathcal{P}''$ w.r.t. AX . Consider the graph $G = G' \triangleright G''$ and some linearized execution σ of G . Suppose that G is not valid w.r.t. AX and instructions $i_1, i_2, \dots, i_k$ form a violating assignment to ϕ . If all of $I = \{i_1, \dots, i_k\}$ are from $\mathcal{P}'$ or $\mathcal{P}''$, this contradicts either $G' \models_{\mathcal{P}'} AX$ or $G'' \models_{\mathcal{P}''} AX$ respectively. Otherwise, there is a proper subset $\emptyset \subset I' \subset I$ such that instructions from I' are from $\mathcal{P}'$ and those in $I'' = I \setminus I'$ are from $\mathcal{P}''$. Consider context cxt corresponding to the assignment $s = \{i_i \mapsto i_i\}$. This is a consistent context and hence by assumption,

$$\{\text{i.st} \mid \text{i} \in I', \text{st} \in \text{Stages}\}^{\text{perm}} \cdot \{\text{i.st} \mid \text{i} \in I(AX) \setminus I', \text{st} \in \text{Stages}\}^{\text{perm}} \subseteq \mathcal{L}$$

for the language $\mathcal{L}$ of $\text{aa}(AX, \text{cxt})$. The linearized trace σ projected to events in $\{\text{i.st} \mid \text{i} \in I, \text{st} \in \text{Stages}\}$ belongs to the corresponding concretized set. This follows since all events of I' precede those of I'' in σ . Hence the concrete automaton $\text{aa}(AX, s)$ on the linearized trace σ accepts it, contradicting the assumption that instructions from I constituted a violation of AX . $\square$

A.3 Bounding In-Progress Instructions (Lemma 7.4)

We define auxiliary notation. Fix a trace σ and consider a trace index $0 \leq j \leq |\sigma|$. Let $\text{CM}(j)$ and $\text{NF}(j)$ be the set of instructions which have completed execution of all events and

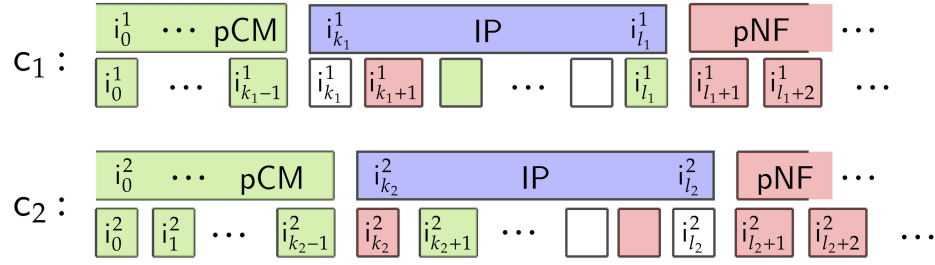


Figure A.1: Completed prefix (pCM), in-progress (IP), and not-fetched postfix (pNF) during execution. The commit-point $\text{cp}(j, c)$ and fetch-point $\text{fp}(j, c)$ delimit the in-progress range for each core. The second row for each core shows completed (CM, green) and not-fetched (NF, red) instructions; pCM and pNF are their maximal prefix/postfix-closed subsets.

not executed any events at step j respectively. The (maximal) prefix-closed subset of $\text{CM}(j)$ is $\text{pCM}(j)$ and the (maximal) postfix-closed subset of $\text{NF}(j)$ is $\text{pNF}(j)$:

$$\begin{aligned} \text{pCM}(j) &= \{i \mid \forall i'. i' \leq_r i \implies i' \in \text{CM}(j)\} \\ \text{pNF}(j) &= \{i \mid \forall i'. i \leq_r i' \implies i' \in \text{NF}(j)\} \\ \text{IP}(j) &= \text{instrsOf}(\mathcal{P}) \setminus (\text{pCM}(j) \cup \text{pNF}(j)) \end{aligned}$$

We define the commit-point and fetch-point of the trace at step j for core c :

$$\begin{aligned} \text{cp}(j, c) &= \min\{\lambda(i) \mid c(i) = c \wedge i \notin \text{CM}(j)\} \\ \text{fp}(j, c) &= \max\{\lambda(i) \mid c(i) = c \wedge i \notin \text{NF}(j)\} \end{aligned}$$

Figure A.1 illustrates this notation.

Proof of Lemma 7.4. Consider a t -reordering bounded trace σ . First note that $|\text{IP}(j)| = \sum_{c \in \text{Cores}} (\text{fp}(j, c) - \text{cp}(j, c) + 1)$. We show that $(\text{fp}(j, c) - \text{cp}(j, c) + 1) \leq t$ for all cores c .

Suppose this quantity exceeded t for some core c . Let the instructions at labels $\text{cp}(j, c)$ and $\text{fp}(j, c)$ be i_1 and i_2 respectively. By definition of $\text{cp}(\cdot)$ and $\text{fp}(\cdot)$ we know that i_1 has not yet executed some event at j , say $i_1.\text{st}_1$, and i_2 has executed at least one event by j , say $i_2.\text{st}_2$. However, $\text{diff}_r(i_1, i_2) = \text{fp}(j, c) - \text{cp}(j, c) \geq t$, contradicting the t -reordering boundedness assumption. $\square$

A.4 Bounding Active Instructions (Lemma 7.5)

We first prove a generalization of the coupling condition in the t -reordering bound definition. We say that instructions form a *coupling chain* $i \bowtie i_1 \bowtie \dots \bowtie i_k \bowtie i'$ if $\text{coup}(i_i, i_{i+1})$ for each consecutive pair. Two instructions are k -coupled if they are endpoints of a coupling chain of length k .

Lemma A.2. *For any $k \in \mathbb{N}$ there exists a bound $t_k \in \mathbb{N}$ (independent of program $\mathcal{P}$ and axiomatic model) such that: for any t -reordering bounded trace σ of program $\mathcal{P}$, and for any same-core instructions i, i' , if there exists a coupling chain $i \bowtie i_1 \bowtie \dots \bowtie i_k \bowtie i'$, then $|\text{diff}_r(i, i')| < t_k$.*

Proof. Consider a t -reordering bounded trace σ for program $\mathcal{P}$, and two same-core instructions i, i' on core c with coupling chain $i \bowtie i_1 \bowtie \dots \bowtie i_k \bowtie i'$. WLOG let $i <_r i'$.

Case 1. Some event of i' executes before an event of i : then $\text{diff}_r(i, i') < t$ by the first condition of t -reordering boundedness.

Case 2. All events of i' execute strictly after those of i : let $i_1^c, \dots, i_l^c$ be the instructions from c such that some event of each i_i^c has executed between $\text{end}(i)$ and $\text{start}(i')$. For each i_i^c there exists an instruction $i_j \in \{i_1, \dots, i_k\}$ such that $\text{coup}(i_i^c, i_j)$, since the interval $[\text{start}(i_i^c), \text{pfxend}(i_i^c)]$ must overlap with one of the coupling chain intervals. By the second condition of t -reordering boundedness, at most t instructions on core c can be coupled with any of the i_j 's. Consequently $l < k \cdot t$. Using the first condition, at most $t - 1$ instructions after i (in $<_r$ order) can execute before i , and similarly at most $t - 1$ before i' can execute after i' . Hence $\text{diff}_r(i, i') \leq l + 2(t - 1) < t(k + 2) - 2$.

Since Case 1's bound is smaller, $t_k = t(k + 2) - 2$ suffices. $\square$

Proof of Lemma 7.5. If i_1 and i_2 are k -coupled and i_2 and i_3 are k -coupled, then i_1 and i_3 are $(2k - 1)$ -coupled.

Consider an instruction $i \in \text{IP}(j)$. On any core c , there can exist at most t_{2k-3} instructions with which i is k -coupled. Suppose to the contrary there were more. Then there are two instructions i_1, i_2 on c with $|\text{diff}_r(i_1, i_2)| \leq t_{2k-3}$ which are both k -coupled with i . By the earlier observation, i_1 and i_2 are $(2k - 1)$ -coupled, which by Lemma A.2 implies $|\text{diff}_r(i_1, i_2)| < t_{2k-3}$, a contradiction.

Across all cores and all instructions in $\text{IP}(j)$, there are at most $|\text{IP}(j)| \cdot |\text{Cores}| \cdot t_{2k-3}$ instructions in $\text{pCM}(j) \cup \text{IP}(j)$ that are k -coupled with some instruction in $\text{IP}(j)$. Hence the lemma holds for $b_k = |\text{IP}(j)| \cdot |\text{Cores}| \cdot t_{2k-3}$, which is independent of the program and axiom. $\square$

A.5 Active Automata and the Feasibility Theorem (Theorem 7.2)

For a K -ary axiom AX , a concrete automaton $\text{aa}(\text{AX}, s)$ is *active* at trace index j if there is a path from its current state to a non-accepting state on some word from the remaining events. Intuitively, an active automaton can still reject the trace.

Lemma A.3. *Consider a trace σ and index $0 \leq j \leq |\sigma|$. For an injective assignment $s : \text{I}(\text{AX}) \rightarrow \text{pCM}(j) \cup \text{IP}(j)$, the automaton $\text{aa}(\text{AX}, s)$ is active only if each instruction in $I = s(\text{I}(\text{AX}))$ is K -coupled by I to some instruction in $\text{IP}(j) \cap I$.*

Proof. Assume that some instruction $i \in I$ is not K -coupled by I to any instruction in $\text{IP}(j) \cap I$. Then there exists a partitioning $I = I_a \uplus I_b$ such that $i \in I_a$, $\text{IP}(j) \cap I \subseteq I_b$, and there do not exist $i' \in I_a$ and $i'' \in I_b$ such that $\text{coup}(i', i'')$. This holds since $|I| = K$ and we can define I_b to be the **coup**-closure starting from $\text{IP}(j) \cap I$.

By the definition of **coup**, I_a is a prefix-closed subset of I . Hence by the equivalent characterization of extensibility in Lemma A.1, there is no permutation over the events of I_b such that the automaton does not accept. Consequently, the automaton is not active. $\square$

Corollary A.1. *For each universal axiom AX, there is a bound $d \in \mathbb{N}$ such that for any program $\mathcal{P}$ and t -reordering bounded trace σ of $\mathcal{P}$, $|\text{active}_{\text{AX}, \mathcal{P}}(j)| \leq d$ for all $0 \leq j \leq |\sigma|$.*

Proof. Each active automaton must have all its corresponding instructions be K -coupled to some instruction in $\text{IP}(j)$. By Lemma 7.5, the number of instructions K -coupled to some instruction in $\text{IP}(j)$ is bounded above by b_K . Consequently, the number of active automata is bounded (by $\binom{b_K}{K}$). $\square$

This corollary, combined with the axiom-automata construction (Lemma 7.3) and the deployment procedure described in Section 7.4, yields Theorem 7.2: synthesis of finite-state, bounded-history operational models satisfying soundness and t -completeness is feasible for refinable, extensible universal axiomatic semantics.

Appendix B

Hardware Mapping for DAWG and COLORIS

This appendix describes how the DAWG and COLORIS hardware designs map to the abstract resource model from Chapter 5.

B.1 DAWG Cache Partitioning

DAWG [67] partitions the associative ways of a set-associative cache. The mapping to the abstract resource model is as follows.

Resource. The *resource* consists of the associative ways of a single cache set. Each way is a cell in the abstract model, indexed by $\mathbb{I} = \{0, \dots, W - 1\}$ where W is the associativity.

Domains. Each domain is identified by a unique 8-bit `domain_id`. DAWG maintains per-domain `hit_way` and `fill_way` bitmasks that record which ways the domain can read from and write to, respectively. In the abstract model, these bitmasks define the allocation map $a : \mathbb{I} \rightarrow \mathbb{D}$.

Policies. The replacement metadata records the order of cache way accesses and is itself partitioned: updates to the metadata for one domain must not influence another domain's replacement decisions. DAWG introduces a `policy_mask`, computed from `hit_way` and `fill_way`, that constrains metadata updates to respect partition boundaries. In the abstract model, the metadata and policy mask together constitute the policy state p , with the policy elements E corresponding to individual tree pointer bits (PLRU) or access bits (NRU).

Access semantics. A cache access in DAWG proceeds as in a traditional set-associative cache: all ways are checked for a hit. On a hit, the metadata is updated according to the

`policy_mask`, ensuring that only the accessing domain’s portion of the metadata changes. On a miss, DAWG selects a victim way using the partition-constrained metadata, updates the way contents, and modifies the metadata accordingly. This maps directly to the access semantics in Listing 5.1.

B.2 COLORIS Page Coloring

COLORIS [133] assigns page colors to memory pages based on address bits, ensuring that processes with different colors use different cache sets.

Resource. The *resource* consists of the cache ways within a given set. The page color determines which cache set a memory address maps to.

Domains. Each domain corresponds to an operating system process, identified by its process ID. In our secure modification (Section 5.5), each process is assigned disjoint page colors, ensuring that processes cannot share cache sets.

Policies. In the COLORIS model, the cache behaves as a standard cache within each color partition. We use an uninterpreted eviction function to model the replacement policy abstractly, which strengthens the verification by proving security for all possible eviction policies rather than a specific one.

Access semantics. Cache access follows standard semantics: on a hit, the replacement state is updated; on a miss, a victim is selected and replaced. The key security property is that the page color assignment prevents cross-domain interference at the cache-set level, which the conditional equality invariants verify.

Bibliography

- [1] Sarita V. Adve and Kourosh Gharachorloo. “Shared Memory Consistency Models: A Tutorial”. In: *IEEE Computer* 29.12 (1996), pp. 66–76.
- [2] Jade Alglave, Luc Maranget, and Michael Tautschnig. “Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory”. In: *ACM Transactions on Programming Languages and Systems* 36.2 (2014).
- [3] Rajeev Alur et al. “Syntax-Guided Synthesis”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. 2013, pp. 1–8. DOI: [10.1109/FMCAD.2013.6679385](https://doi.org/10.1109/FMCAD.2013.6679385).
- [4] Alasdair Armstrong et al. “ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019).
- [5] Haniel Barbosa et al. “cvc5: A Versatile and Industrial-Strength SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442. DOI: [10.1007/978-3-030-99524-9_24](https://doi.org/10.1007/978-3-030-99524-9_24).
- [6] Gilles Barthe, Juan Manuel Crespo, and César Kunz. “Relational Verification Using Product Programs”. In: *International Symposium on Formal Methods (FM)*. 2011.
- [7] Mark Batty, Alastair F. Donaldson, and John Wickerson. “Overhauling SC atomics in C11 and OpenCL”. In: *POPL*. 2016, pp. 634–648.
- [8] Mark Batty et al. “Mathematizing C++ Concurrency”. In: *POPL*. 2011.
- [9] Armin Biere et al. “Symbolic Model Checking without BDDs”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 1579. Lecture Notes in Computer Science. Springer, 1999, pp. 193–207. DOI: [10.1007/3-540-49059-0_14](https://doi.org/10.1007/3-540-49059-0_14).
- [10] Nathan Binkert et al. “The gem5 Simulator”. In: *ACM SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7.
- [11] James Bornholt and Emina Torlak. “Synthesizing Memory Models from Framework Sketches and Litmus Tests”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2017, pp. 467–481.

- [12] Aaron R. Bradley. “SAT-Based Model Checking without Unrolling”. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Vol. 6538. Lecture Notes in Computer Science. Springer, 2011, pp. 70–87. DOI: [10.1007/978-3-642-18275-4_7](https://doi.org/10.1007/978-3-642-18275-4_7).
- [13] Aaron R. Bradley and Zohar Manna. “Property-Directed Incremental Invariant Generation”. In: *Formal Aspects of Computing* 20.4–5 (2008), pp. 379–405.
- [14] Robert Brayton and Alan Mishchenko. “ABC: An Academic Industrial-Strength Verification Tool”. In: *International Conference on Computer Aided Verification (CAV)*. 2010.
- [15] J. Richard Büchi. “On a Decision Method in Restricted Second-Order Arithmetic”. In: *The Collected Works of J. Richard Büchi* (1990).
- [16] Pablo Buiras et al. “Validation of Side-Channel Models via Observation Refinement”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2021.
- [17] Jerry R. Burch and David L. Dill. “Automatic Verification of Pipelined Microprocessor Control”. In: *International Conference on Computer Aided Verification (CAV)*. 1994, pp. 68–80.
- [18] Sunjay Cauligi et al. “Constant-Time Foundations for the New Spectre Era”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2020, pp. 913–926.
- [19] Sunjay Cauligi et al. “SoK: Practical Foundations for Software Spectre Defenses”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 666–680. DOI: [10.1109/sp46214.2022.9833707](https://doi.org/10.1109/sp46214.2022.9833707).
- [20] Christopher Celio. *Sodor Processor Collection*. RISC-V Education. <https://github.com/ucb-bar/riscv-sodor>. 2015.
- [21] Kevin Cheang et al. “A Formal Approach to Secure Speculation”. In: *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 2019, pp. 288–303. DOI: [10.1109/CSF.2019.00027](https://doi.org/10.1109/CSF.2019.00027).
- [22] Mark Chen et al. “Evaluating Large Language Models Trained on Code”. In: *arXiv preprint arXiv:2107.03374* (2021).
- [23] Joonwon Choi et al. “Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification”. In: *Proceedings of the ACM on Programming Languages* 1.ICFP (2017).
- [24] Edmund Clarke et al. “Counterexample-Guided Abstraction Refinement”. In: *International Conference on Computer Aided Verification (CAV)*. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 154–169. DOI: [10.1007/10722167_15](https://doi.org/10.1007/10722167_15).
- [25] Edmund M. Clarke and E. Allen Emerson. “Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic”. In: *Logics of Programs*. Vol. 131. Lecture Notes in Computer Science. Springer, 1982, pp. 52–71. DOI: [10.1007/BFb0025774](https://doi.org/10.1007/BFb0025774).

- [26] Edmund M. Clarke, Orna Grumberg, and David E. Long. “Model Checking and Abstraction”. In: *ACM Transactions on Programming Languages and Systems* 16.5 (1994), pp. 1512–1542. DOI: [10.1145/186025.186051](https://doi.org/10.1145/186025.186051).
- [27] Michael R. Clarkson and Fred B. Schneider. “Hyperproperties”. In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210. DOI: [10.3233/JCS-2009-0393](https://doi.org/10.3233/JCS-2009-0393).
- [28] cocotb contributors. *cocotb: Coroutine-based Co-simulation Testbench*. 2024.
- [29] *CVA6 RTL Bug Issues*. <https://github.com/openhwgroup/cva6/issues?q=is%3Aissue+label%3AType%3ABug+label%3AComponent%3ARTL+is%3Aopen>. Accessed: 2024.
- [30] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. “Hunting the Haunter — Efficient Relational Symbolic Execution for Spectre with Haunted RelSE”. In: *Network and Distributed System Security Symposium (NDSS)*. 2021.
- [31] diffblue. *EBMC: Enhanced Bounded Model Checker*. 2024.
- [32] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. “Efficient Implementation of Property Directed Reachability”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. 2011, pp. 125–134.
- [33] Mehmet Emre et al. “Translating C to Safer Rust”. In: *Proceedings of the ACM on Programming Languages* 5.OOPSLA (2021), pp. 1–29.
- [34] Xaver Fabian, Marco Guarnieri, and Marco Patrignani. “Automatic Detection of Speculative Execution Combinations”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2022.
- [35] Mohammad Rahmani Fadiheh et al. “A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors”. In: *57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: [10.1109/DAC18072.2020.9218572](https://doi.org/10.1109/DAC18072.2020.9218572).
- [36] Mohammad Rahmani Fadiheh et al. “Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking”. In: *DATE*. 2019, pp. 994–999. DOI: [10.23919/DATE.2019.8715004](https://doi.org/10.23919/DATE.2019.8715004).
- [37] Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. “Relational Symbolic Execution”. In: *PPDP*. 2019.
- [38] Grigory Fedyukovich et al. “Quantified Invariants via Syntax-Guided Synthesis”. In: *International Conference on Computer Aided Verification (CAV)*. 2019.
- [39] Bernd Finkbeiner and Sven Schewe. “Bounded Synthesis”. In: *International Journal on Software Tools for Technology Transfer (STTT)* 15 (2013), pp. 519–539.
- [40] Emily First et al. “Baldur: Whole-Proof Generation and Repair with Large Language Models”. In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2023, pp. 1229–1241.

- [41] Harry D. Foster. “Trends in Functional Verification: A 2014 Industry Study”. In: *52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 2015, pp. 1–6.
- [42] Antoine Geimer et al. “A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2023.
- [43] GitHub. *The State of Open Source and Rise of AI in 2023*. <https://github.blog/news-insights/research/the-state-of-open-source-and-ai/>. GitHub Octoverse Report. 2023.
- [44] Klaus v. Gleissenthall et al. “Solver-Aided Constant-Time Hardware Verification”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2021, pp. 429–444. DOI: [10.1145/3460120.3484810](https://doi.org/10.1145/3460120.3484810).
- [45] Klaus von Gleissenthall et al. “IODINE: Verifying Constant-Time Execution of Hardware”. In: *USENIX Security*. 2019.
- [46] Adwait Godbole, Yatin A. Manerkar, and Sanjit A. Seshia. “Automated Conversion of Axiomatic to Operational Models: Theory and Practice”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. 2022, pp. 331–340.
- [47] Adwait Godbole, Yatin A. Manerkar, and Sanjit A. Seshia. “SemPat: Using Hyperproperty-based Semantic Analysis to Generate Microarchitectural Attack Patterns”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2024, pp. 2756–2770. DOI: [10.1145/3658644.3690214](https://doi.org/10.1145/3658644.3690214).
- [48] Adwait Godbole et al. “Lifting Micro-Update Models from RTL for Formal Security Analysis”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2024, pp. 631–648. DOI: [10.1145/3620665.3640418](https://doi.org/10.1145/3620665.3640418).
- [49] Adwait Godbole et al. “Modelling and Verification of Security-Oriented Resource Partitioning Schemes”. In: *Formal Methods in Computer-Aided Design (FMCAD)*. 2023, pp. 268–273. DOI: [10.34727/2023/isbn.978-3-85448-060-0_35](https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_35).
- [50] Adwait Godbole et al. “PyCaliper: Python-embedded Infrastructure for RTL Verification and Specification Synthesis”. In: *International Conference on Computer Aided Verification (CAV)*. 2025.
- [51] Aman Goel and Karem A. Sakallah. “AVR: Abstractly Verifying Reachability”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 12078. Lecture Notes in Computer Science. Springer, 2020, pp. 413–422.
- [52] Joseph A. Goguen and José Meseguer. “Security Policies and Security Models”. In: *1982 IEEE Symposium on Security and Privacy*. IEEE, 1982, pp. 11–20. DOI: [10.1109/SP.1982.10014](https://doi.org/10.1109/SP.1982.10014).
- [53] Joseph A. Goguen and José Meseguer. “Unwinding and Inference Control”. In: *IEEE Symposium on Security and Privacy*. 1984, pp. 75–86. DOI: [10.1109/SP.1984.10019](https://doi.org/10.1109/SP.1984.10019).

- [54] Kathryn E. Gray et al. “An Integrated Concurrency and Core-ISA Architectural Envelope Definition, and Test Oracle, for IBM POWER Multiprocessors”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015, pp. 635–646.
- [55] Sergey Grebenshchikov et al. “Synthesizing Software Verifiers from Proof Rules”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2012. DOI: [10.1145/2254064.2254112](https://doi.org/10.1145/2254064.2254112).
- [56] Roberto Guanciale, Musard Balliu, and Mads Dam. “InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2020, pp. 1853–1869.
- [57] Marco Guarnieri et al. “Hardware-Software Contracts for Secure Speculation”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1868–1883. DOI: [10.1109/SP40001.2021.00036](https://doi.org/10.1109/SP40001.2021.00036).
- [58] Marco Guarnieri et al. “Spectector: Principled Detection of Speculative Information Flows”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1–19. DOI: [10.1109/sp40000.2020.00011](https://doi.org/10.1109/sp40000.2020.00011).
- [59] Shengjian Guo et al. “SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection”. In: *ICSE*. 2020.
- [60] Klaus Havelund and Grigore Roşu. “Synthesizing Monitors for Safety Properties”. In: *TACAS*. 2002.
- [61] Stafford Horne. *SDRAM Controller*. <https://github.com/stffrdhrn/sdram-controller>. Accessed 2021-11-05.
- [62] Yao Hsiao et al. “Synthesizing Formal Models of Hardware from RTL for Efficient Verification of Memory Model Implementations”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 2021, pp. 679–694.
- [63] Bo-Yuan Huang et al. “ILang: A Modeling and Verification Platform for SoCs Using Instruction-Level Abstractions”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2019, pp. 351–357.
- [64] Susmit Jha et al. “Oracle-Guided Component-Based Program Synthesis”. In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*. Vol. 1. 2010, pp. 215–224. DOI: [10.1145/1806799.1806833](https://doi.org/10.1145/1806799.1806833).
- [65] Shoaib Kamil et al. “Verified Lifting of Stencil Computations”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2016.
- [66] Rahul Kande et al. “TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities”. In: *USENIX Security Symposium*. 2022.
- [67] Vladimir Kiriansky et al. “DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 974–987. DOI: [10.1109/MICRO.2018.00083](https://doi.org/10.1109/MICRO.2018.00083).

- [68] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19. DOI: [10.1109/SP.2019.00002](https://doi.org/10.1109/SP.2019.00002).
- [69] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. “Taming Release-Acquire Consistency”. In: *POPL*. 2016.
- [70] Ori Lahav et al. “Repairing Sequential Consistency in C/C++11”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2017.
- [71] Leslie Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Transactions on Computers* C-28.9 (1979), pp. 690–691.
- [72] Lattner, Chris and others. *CIRCT: Circuit IR Compilers and Tools*. <https://circuit.llvm.org/>. 2025.
- [73] Hernán Ponce de León and Johannes Kinder. “Cats vs. Spectre: An Axiomatic Approach to Modeling Speculative Execution Attacks”. In: *IEEE Symposium on Security and Privacy*. 2022, pp. 235–248.
- [74] Kevin M. Lepak, Gordon B. Bell, and Mikko H. Lipasti. “Silent Stores and Store Value Locality”. In: *IEEE Transactions on Computers* 50.11 (2001), pp. 1174–1190.
- [75] Kevin M. Lepak and Mikko H. Lipasti. “Silent Stores for Free”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2000, pp. 22–31. DOI: [10.1145/360128.360133](https://doi.org/10.1145/360128.360133).
- [76] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium*. USENIX Association, 2018, pp. 973–990.
- [77] Fangfei Liu et al. “Last-Level Cache Side-Channel Attacks are Practical”. In: *IEEE Symposium on Security and Privacy*. 2015, pp. 605–622.
- [78] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. “PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2014, pp. 635–646. DOI: [10.1109/MICRO.2014.38](https://doi.org/10.1109/MICRO.2014.38).
- [79] Daniel Lustig et al. “Automated Synthesis of Comprehensive Memory Model Litmus Test Suites”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2017.
- [80] Daniel Lustig et al. “COATCheck: Verifying Memory Ordering at the Hardware-OS Interface”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2016.
- [81] Yatin A. Manerkar. *Multi-V-Scale: A Multi-Core Extension of the V-Scale RISC-V Processor*. Tech. rep. Available at https://github.com/ymanerka/multi_vscale. 2019.

- [82] Yatin A. Manerkar et al. “CCICheck: Using μ hb Graphs to Verify the Coherence-Consistency Interface”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015, pp. 26–37.
- [83] Yatin A. Manerkar et al. “PipeProof: Automated Memory Consistency Proofs for Microarchitectural Specifications”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Press, 2018, pp. 788–801.
- [84] Yatin A. Manerkar et al. “RTLCheck: Verifying the Memory Consistency of RTL Designs”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 2017, pp. 463–476. DOI: [10.1145/3123939.3124536](https://doi.org/10.1145/3123939.3124536).
- [85] Makai Mann et al. “Pono: A Flexible and Extensible SMT-Based Model Checker”. In: *International Conference on Computer Aided Verification (CAV)*. Vol. 12760. Lecture Notes in Computer Science. Springer, 2021, pp. 461–474.
- [86] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.
- [87] Zohar Manna and Richard Waldinger. “A Deductive Approach to Program Synthesis”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2.1 (1980), pp. 90–121.
- [88] Nicholas Mosier et al. “Axiomatic Hardware-Software Contracts for Security”. In: *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2022. DOI: [10.1145/3470496.3527412](https://doi.org/10.1145/3470496.3527412).
- [89] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: [10.1007/978-3-540-78800-3_24](https://doi.org/10.1007/978-3-540-78800-3_24).
- [90] MyHDL contributors. *MyHDL: From Python to Silicon*. 2024.
- [91] Aina Niemetz, Mathias Preiner, and Armin Biere. *Boolector at the SMT Competition 2018*. Tech. rep. FMV Reports Series, Technical Report 18/1; SMT-COMP 2018. Johannes Kepler University Linz, 2018.
- [92] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. “An Operational Semantics for C/C++11 Concurrency”. In: *OOPSLA*. 2016, pp. 111–128.
- [93] Rishiyur S. Nikhil. “Bluespec System Verilog: Efficient, Correct RTL from High Level Specifications”. In: *Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. 2004, pp. 69–70.
- [94] Chase Norman, Adwait Godbole, and Yatin A. Manerkar. “PipeSynth: Automated Synthesis of Microarchitectural Axioms for Memory Consistency”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2023, pp. 513–527. DOI: [10.1145/3582016.3582056](https://doi.org/10.1145/3582016.3582056).

- [95] Oleksii Oleksenko et al. “Revizor: Testing Black-Box CPUs against Speculation Contracts”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2022.
- [96] Scott Owens, Susmit Sarkar, and Peter Sewell. “A Better x86 Memory Model: x86-TSO”. In: *TPHOLs*. 2009.
- [97] Oded Padon et al. “Induction Duality: Primal-Dual Search for Invariants”. In: *Proceedings of the ACM on Programming Languages* 6.POPL (2022).
- [98] Amir Pnueli. “In Transition from Global to Modular Temporal Reasoning about Programs”. In: *Logics and Models of Concurrent Systems*. Vol. 13. NATO ASI Series. Springer, 1984, pp. 123–144.
- [99] Amir Pnueli. “The Temporal Logic of Programs”. In: *18th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 1977, pp. 46–57. DOI: [10.1109/SFCS.1977.32](https://doi.org/10.1109/SFCS.1977.32).
- [100] Elizabeth Polgreen et al. “UCLID5: Multi-modal Formal Modeling, Verification, and Synthesis”. In: *International Conference on Computer Aided Verification (CAV)*. Vol. 13371. Lecture Notes in Computer Science. Springer, 2022, pp. 538–551. DOI: [10.1007/978-3-031-13185-1_27](https://doi.org/10.1007/978-3-031-13185-1_27).
- [101] Christopher Pulte et al. “Simplifying ARM Concurrency: Multicopy-Atomic Axiomatic and Operational Models for ARMv8”. In: *Proceedings of the ACM on Programming Languages* 2.POPL (2018), pp. 1–29.
- [102] Jose Rodrigo Sanchez Vicarte et al. “Opening Pandora’s Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data”. In: *International Symposium on Computer Architecture (ISCA)*. 2021, pp. 347–360.
- [103] Sanjit A. Seshia. “Combining Induction, Deduction, and Structure for Verification and Synthesis”. In: *Proceedings of the IEEE* 103.11 (2015), pp. 2036–2051. DOI: [10.1109/JPROC.2015.2471838](https://doi.org/10.1109/JPROC.2015.2471838).
- [104] Sanjit A. Seshia. “Sciduction: Combining Induction, Deduction, and Structure for Verification and Synthesis”. In: *Design Automation Conference (DAC)*. 2012. DOI: [10.1145/2228360.2228425](https://doi.org/10.1145/2228360.2228425).
- [105] Sanjit A. Seshia and Pramod Subramanyan. “UCLID5: Integrating Modeling, Verification, Synthesis, and Learning”. In: *Proceedings of the 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 2018. DOI: [10.1109/MEMCOD.2018.8556946](https://doi.org/10.1109/MEMCOD.2018.8556946).
- [106] Peter Sewell et al. “x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors”. In: *Communications of the ACM* (2010).
- [107] André Seznec. “A 64-Kbytes ITTAGE Indirect Branch Predictor”. In: *Journal of Instruction-Level Parallelism: Championship Branch Prediction (JWAC-2)*. 2011.

- [108] André Seznec. “The L-TAGE Branch Predictor”. In: *Journal of Instruction-Level Parallelism* 9 (2007).
- [109] Manish Shetty et al. “Syzygy: Dual Code-Test C to (Safe) Rust Translation using LLMs and Dynamic Analysis”. In: *LLM4Code Workshop at the 47th International Conference on Software Engineering (ICSE)*. 2025.
- [110] Anirudh Sivaraman et al. “Packet Transactions: High-Level Programming for Line-Rate Switches”. In: *SIGCOMM*. 2016.
- [111] Jens U. Skakkebak, Robert B. Jones, and David L. Dill. “Formal Verification of Out-of-Order Execution Using Incremental Flushing”. In: *International Conference on Computer Aided Verification (CAV)*. 1998, pp. 98–109.
- [112] Wilson Snyder. *Verilator*. 2022.
- [113] Avinash Sodani and Gurindar S. Sohi. “Dynamic Instruction Reuse”. In: *Conference Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*. 1997, pp. 194–205.
- [114] Armando Solar-Lezama et al. “Combinatorial Sketching for Finite Programs”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2006, pp. 404–415. DOI: [10.1145/1168857.1168907](https://doi.org/10.1145/1168857.1168907).
- [115] Chuyue Sun et al. “Clover: Closed-Loop Verifiable Code Generation”. In: *SAIV*. Vol. 14846. Lecture Notes in Computer Science. Springer, 2024, pp. 134–155.
- [116] Tachio Terauchi and Alex Aiken. “Secure Information Flow as a Safety Problem”. In: *12th International Conference on Static Analysis (SAS)*. Vol. 3672. Lecture Notes in Computer Science. Springer, 2005, pp. 352–367. DOI: [10.1007/11547662_24](https://doi.org/10.1007/11547662_24).
- [117] Robert M. Tomasulo. “An Efficient Algorithm for Exploiting Multiple Arithmetic Units”. In: *IBM Journal of Research and Development* 11.1 (1967), pp. 25–33.
- [118] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. “Security Verification via Automatic Hardware-Aware Exploit Synthesis: The CheckMate Approach”. In: *IEEE Micro* 39.3 (2019), pp. 84–93. DOI: [10.1109/MM.2019.2910010](https://doi.org/10.1109/MM.2019.2910010).
- [119] Caroline Trippel et al. “TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA”. In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2017.
- [120] Viktor Vafeiadis et al. “Common Compiler Optimisations are Invalid in the C11 Memory Model and what we can do about it”. In: *POPL*. 2015.
- [121] Marco Vassena et al. “Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade”. In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021). DOI: [10.1145/3434330](https://doi.org/10.1145/3434330).

- [122] Miroslav N. Velev and Randal E. Bryant. “Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic”. In: *CHARME*. 1999.
- [123] Mahesh Viswanathan and Ramesh Viswanathan. “Foundations for Circular Compositional Reasoning”. In: *28th International Colloquium on Automata, Languages and Programming (ICALP)*. Springer, 2001, pp. 835–847.
- [124] Guanhua Wang et al. “KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2020).
- [125] Zilong Wang et al. “Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2023.
- [126] Andrew Waterman et al. *The RISC-V Instruction Set Manual*. Tech. rep. UCB/EECS-2014-54. UC Berkeley, 2014.
- [127] Conrad Watt et al. “Repairing and Mechanising the JavaScript Relaxed Memory Model”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2020, pp. 346–361.
- [128] Mark Weiser. “Program Slicing”. In: *IEEE Transactions on Software Engineering* (1984).
- [129] Claire Wolf and Johann Glaser. *Yosys – A Free Verilog Synthesis Suite*. <https://yosyshq.net/yosys/>. 2019.
- [130] Pierre Wolper, Moshe Y. Vardi, and A. Prasad Sistla. “Reasoning About Infinite Computation Paths”. In: *FOCS*. 1983, pp. 185–194.
- [131] Wenjie Xiong and Jakub Szefer. “Leaking Information Through Cache LRU States”. In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020, pp. 139–152.
- [132] Aidan Z. H. Yang et al. “VERT: Verified Equivalent Rust Transpilation with Large Language Models as Few-Shot Learners”. In: *arXiv preprint arXiv:2404.18852* (2024).
- [133] Ying Ye et al. “COLORIS: A Dynamic Cache Partitioning System Using Page Coloring”. In: *2014 23rd International Conference on Parallel Architecture and Compilation (PACT)*. 2014, pp. 381–392.
- [134] YosysHQ. *SymbiYosys: Front-end for Yosys-based Formal Verification Flows*. <https://github.com/YosysHQ/sby>. 2023.
- [135] Jiyong Yu et al. “Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 2019, pp. 954–968.

- [136] Florian Zaruba and Luca 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”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27.11 (2019), pp. 2629–2640.
- [137] Yu Zeng, Aarti Gupta, and Sharad Malik. “Automatic Generation of Architecture-Level Models from RTL Designs for Processors and Accelerators”. In: *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe (DATE)*. 2022, pp. 460–465.
- [138] Hanliang Zhang et al. “Ownership Guided C to Rust Translation”. In: *International Conference on Computer Aided Verification (CAV)*. 2023.
- [139] Hongce Zhang, Aarti Gupta, and Sharad Malik. “Syntax-Guided Synthesis for Lemma Generation in Hardware Model Checking”. In: *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2021.
- [140] Hongce Zhang et al. “Synthesizing Environment Invariants for Modular Hardware Verification”. In: *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2020, pp. 202–225.