Scalable Verification with Applications to Hardware Security

Sushant Dinesh

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-133
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-133.html

May 27, 2025

Scalable Verification with Applications to Hardware Security

By

Sushant Dinesh

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Christopher Fletcher, Chair
Professor Sanjit A. Seshia
Professor Raluca Ada Popa
Dr. Patrice Godefroid

Spring 2025

Scalable Verification with Applications to Hardware Security

Abstract

Scalable Verification with Applications to Hardware Security

by

Sushant Dinesh

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Christopher Fletcher, Chair

As Moore's Law slows, microarchitects are turning to clever and exotic microarchitectural optimizations to accelerate workloads. However, these optimizations are often data-dependent, inadvertently creating side channels that leak sensitive information. Disabling them for security is impractical. Meanwhile, security-critical software—such as cryptographic code—lacks visibility into how its execution might trigger such leaks. As these optimizations grow more complex, writing microarchitecturally safe software will only become more difficult. Defending against side-channel attacks therefore demands a holistic, cross-layer approach that bridges the hardware–software divide.

This dissertation provides a methodology, and accompanying formal analyses, to tackle the microarchitectural side channel problem. Our approach is two-pronged. First, we verify the microarchitecture, e.g., the RTL, for security. Once verified, we obtain software-facing artifacts: security-centric microarchitectural specifications. Then, we develop compiler-like frameworks that take *leaky* code and derived security specifications as inputs to automatically produce microarchitecture-specific code that conforms to the security specification, guaranteeing that no secrets can leak through side-channels.

We achieve this vision through three key technical contributions.

First, drawing inspiration from a variety of side-channels and defenses, we formulate an instruction set-centric definition to microarchitectural security. We articulate this as a formal property: the Safe Instruction Set Property, SISP, which guarantees that unbounded executions of a compositions of instructions do not leak secrets on the microarchitecture. Verifying if a set of instructions satisfies SISP on a microarchitecture gives us a convenient software-facing abstraction: the set of instructions that are safe to allow compute on secret data.

However, state-of-the-art verification tools do not scale to verify SISP on large hardware designs. To overcome this verification bottleneck, we develop H-Houdini, a new scalable

invariant learning algorithm capable of proving properties on large hardware designs. We implement H-Houdini in tool called VeloCT: a (mostly) push-button tool to verify SISP on hardware designs. VeloCT, for the first time, is able to scale security verification to BOOM, a large open-source Out-of-Order (OoO) core, in timescales ranging from 6m to 3.3h from the smallest to the largest parameterization of BOOM. More importantly, the set of safe instructions verified by VeloCT can now be used as a software-facing abstraction to harden code.

Lastly, we develop SynthCT, a program synthesis based framework that uses the safe set specification to automatically harden security-critical code against side-channels on a specific microarchitecture. Notably, SynthCT is a robust, scalable framework that handles modern, complex ISAs like x86-64 with 1000s of instructions, and is capable of rewriting even the most complex instructions, like division (`DIVL`), using a set of simple safe instructions.

We believe that the combination of techniques and tools developed in this thesis can serve as a first step towards holistic, scalable, automated, principled defenses against microarchitectural side-channel attacks.

To my wife, Lavanya, and my family.

Contents

List of Figures

List of Tables

Acknowledgments

I'm deeply grateful to my advisor, Chris, for being an incredible mentor throughout my PhD. I feel genuinely lucky to have had someone with such wide-ranging curiosity and infectious enthusiasm for research. More than just a technical guide, you supported me both professionally and personally, and—most importantly—trusted me to dive into research areas that neither of us knew much about at the start. That trust and freedom to explore played a huge role in shaping the researcher I've become. I've learned so much from you—especially how to think clearly and ask the right questions—so thank you, sincerely, for everything.

I also want to thank my committee members—Sanjit, Raluca, and Patrice—for generously giving their time and thoughtful feedback. A special thank you to Patrice: I deeply appreciated the chance to intern with you early in my PhD. Even though we worked remotely through COVID, the experience was incredibly formative and stayed with me throughout the rest of my research. I truly hope we get to meet in person someday soon!

I'm incredibly grateful to all my lab mates and collaborators who made this journey so memorable—Riccardo, Jiyong, Kartik, Moe, Rutvik, Jose, Nandeeka, Yongye, Boru—thank you for sitting through years of my talks (yes, even the 100th round of H-Houdini) and always giving thoughtful feedback. Special shoutout to Riccardo: I miss our late-night conversations about the future of CS and the world. Moe, I had a blast working with you on OLA in the early days. Rutvik, I really enjoyed our read-gadget sessions and your deep physics and math analogies—though I'll admit, I was probably lost half the time. More recently, Adwait, brainstorming ideas with you has been a ton of fun—let's keep that going. And Alex, digging into tough technical problems with you has been a highlight—I'm looking forward to more of that. This journey wouldn't have been the same without you all.

Last—and most importantly—I want to thank my family for their boundless love and support. To my wife, Lavanya, you are my rock. You stood by me through some of the most difficult decisions of my PhD, and your courage, faith, and steady encouragement gave me the strength to keep moving forward. You believed in my potential even when I doubted it myself, and that trust carried me through more than you know. You inspired me to reach higher—and made the journey full of joy, laughter, and love. To my sister, Dyuthi, and to my parents, Suma and Dinesh—thank you for always being my foundation. You instilled in me the clarity to recognize what truly matters, and the courage to chase after it. I could never have even imagined taking on this journey without your constant belief in me. I also want to thank my in-laws, Uma and Ramkumar, and my brother-in-law, Rahul, for all their warmth and support.

Chapter 1

Introduction

Hardware design is currently at a pivotal juncture. With Moore's law reaching its limits and generative AI (GAI) workloads demanding unprecedented amounts of resources [162], hardware architects increasingly rely on sophisticated microarchitectural optimizations to extract maximal performance from available resources.

However, history shows that these advanced optimizations often inadvertently introduce new microarchitectural vulnerabilities [125, 137, 210, 203, 217, 43, 215, 216], leaking secrets, e.g., cryptographic keys, model weights, passwords, pixels and more, through novel side-channels. Such microarchitectural side-channel attacks are particularly problematic since they break all software-level isolation mechanisms, like kernel-mode isolation [125, 137, 177, 36] and process isolation [180, 177, 123, 36], and cannot be easily *patched*. To make things worse, we are increasingly giving systems access to our private data. Case in point: GAI is being deeply integrated across the software stack—emails and apps, to file and operating systems—granting it access to unprecedented amounts of private data, in exchange for seamless experience and better results. Potentially, *all* of this data is at risk of leaking through side-channels [2, 86]. Thus, to fully enjoy the benefits of emerging hardware/software technologies requires fundamental methods and tools integrated in the hardware-software design lifecycle to provide robust security guarantees.

Addressing microarchitectural side-channel vulnerabilities is inherently challenging. To protect secrets from leakage, the research community has developed many software-only mitigations [170, 239, 164, 37, 207, 44, 38, 212, 171, 14] or advise to carefully craft code following the constant-time programming (CT) discipline [22, 24]. Since software has no visibility into *opaque* world of microarchitectural optimizations, these mitigations need to make conservative assumptions about the microarchitecture in order to be sound. This leads to a high performance overhead. Further, these techniques are brittle: any mismatch between the *assumed* vs. actual leakage, e.g., due to a novel optimization, can leave existing code vulnerable subtly [128].

A parallel line of research explores hardware-only mitigations [235, 45, 234] or suggests secure-by-construction clean-slate hardware design methodologies [238, 236]. Despite being provably safe against *all* side-channels, the former is not implemented in modern processors

as they do not meet the strict power, performance, area (PPA) requirements in the industry, while the latter requires redesigning the processor from scratch—an infeasible task.

We claim that developing holistic, high-performance mitigations necessarily needs to span both hardware and software domains. However, developing such mitigations demands coordination among several key practitioners. To fully appreciate the complexities involved in this process, we will now view the challenges involved in developing a hardware-software solution from the perspectives of three practitioners who currently lack the essential knowledge, techniques, tools, resources, and common vocabulary to systematically confront these security challenges.

Hardware Architects. Microarchitects are responsible implementing new optimizations to achieve higher performance while satisfying strict power and area constraints. However, they typically focus on performance and resource-efficiency, often lacking the expertise and *the feedback needed to assess the security implications* of their design decisions. Consequently, they may inadvertently introduce microarchitectural timing vulnerabilities as they do not have a clear understanding of what makes a microarchitecture *unsafe*.

Verification Engineers. Verification engineers collaborate closely with microarchitects, ensuring the functional correctness of hardware designs. Verification is a *significant bottleneck* in the hardware development cycle, as teams must meet tight deadlines to guarantee timely market releases. *This challenge continues to grow for two primary reasons.* First, designs have become increasingly complex, while the *scalability of verification tools has not kept pace.* As a result, verification engineers frequently need to manually decompose large properties into smaller, tractable intermediate properties. Second, verification now additionally involves evaluating designs for microarchitectural side-channel vulnerabilities. Meanwhile, although verification engineers are skilled in formal methods, *security testing remains largely ad-hoc due to the absence of standardized security specifications or properties.* Current verification tools also struggle to scale when proving end-to-end security properties on complex hardware designs [57, 198, 218].

Software Developers. On the software side, developers working on cryptographic libraries, such as OpenSSL, use constant-time (CT) programming paradigms to prevent cryptographic key leakage via microarchitectural side-channels. The standard guideline is straightforward: *avoid passing secret data to unsafe instructions.* However, developers typically lack precise information about *which instructions are unsafe* for a given microarchitecture. Because this set of unsafe instructions varies between microarchitectures—with each CPU generation potentially being a different microarchitecture—developers conservatively assume that simple instructions are *unlikely* to be optimized. But this assumption is perilous. Recent research demonstrates that *even these simple instructions can become sources of leakage* due to novel microarchitectural optimizations [209, 210, 55, 64].

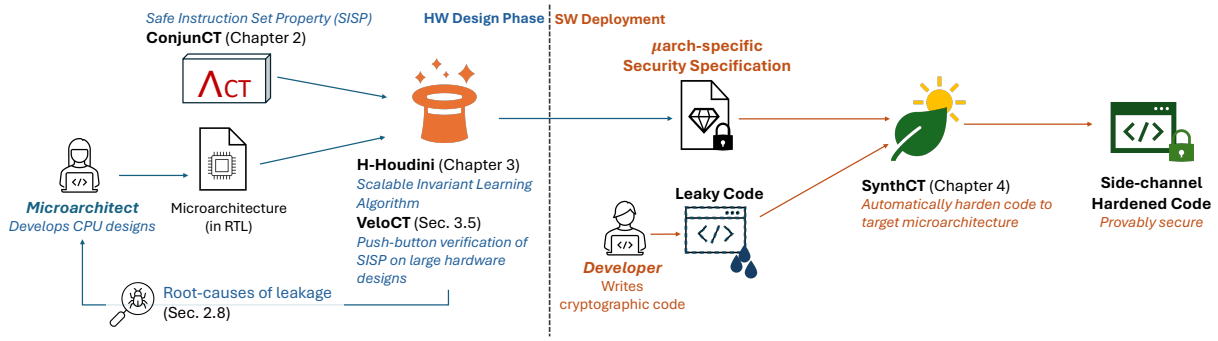


Figure 1.1: Overview of the work presented in this dissertation.

Summary of Challenges

To summarize the above perspectives, there are several key challenges we need to solve across the hardware-software boundary to enable *holistic* defenses against microarchitectural side-channel attacks:

(C1) What does it mean for a microarchitecture to be safe? How do we derive a definition general enough to capture a plethora of side-channels?

(C2) Even if we define what safe means, can we scale verification to automatically prove security on large hardware designs?

(C3) After a microarchitecture is verified for security, what is the correct software-facing microarchitectural security specification we can ship to developers?

(C4) Once we have a microarchitectural security specification, how do we enable developers to easily develop high-performance, microarchitecturally-safe (cryptographic) code with minimal effort?

(C5) Lastly, how can we develop push-button tools that provide microarchitects with actionable feedback on the security of their optimizations during hardware development phase?

This Dissertation. This dissertation provides the first-steps towards a comprehensive solution that addresses the verification and security challenges faced by practitioners across the hardware-software stack. Concisely, this dissertation advocates for the following vision:

> *"We can holistically defend against microarchitectural side-channels by formally proving security properties on hardware implementations, deriving software-facing security specifications from these proofs, and enforcing conformance at the binary level using compiler-like toolchains, thereby ensuring no secrets leak through side-channels."*

The complete realization of this dissertations's vision is depicted in Figure 1.1. On the hardware side, we start with the definition of microarchitecture security. In ConjunCT, published in [61], we develop an instruction-centric property, called the Safe Instruction Set Property, or SISP for short, which formally defines the safety for a set of instructions computing on secret data (C1). Notably, verifying the SISP directly gives us a convenient software-facing security abstraction: the set of instructions that are guaranteed to safely compute to secret data (C3). ConjunCT also implements a tool, using existing verification techniques, to verify SISP on hardware designs (RTL) by learning an inductive invariant. The inductive invariants obtained as proof artifacts enable microarchitects to understand the security implications of their optimization decisions (C5).

However, ConjunCT faces severe scalability issues, unable to prove SISP on larger, Out-of-Order (OoO) cores, e.g., BOOM. To achieve our above vision, we need to be able to *scale beyond state-of-the-art verification tools*. The *key technical advancement* this dissertation contributes is a new, *scalable* invariant learning technique, named H-Houdini (C2), published in [62]. We instantiate H-Houdini in a tool named VeloCT to prove SISP. Using VeloCT, we're able to scale verification, for the first time, to large OoO cores, in timescales ranging from 13s to 3.3h from smallest to the largest design. As artifacts from SISP verification, we obtain μarch-specific security specifications.

Crossing over to the software side, we develop SynthCT [60], an automated program synthesis based framework to harden security-critical code, e.g., cryptography, against side-channels (C4). Just as compilers enable development of architecture agnostic code, SynthCT enables developers to focus on functionality, while automatically handling the security-related microarchitectural idiosyncrasies. The end-result is high-performance, provably secure code.

Tying it all together, this dissertation holistically addresses the challenges in developing secure software on modern leaky hardware. Viewing it through the lens of practitioners:

- VeloCT can be integrated as a part of CI/CD pipeline in an agile development methodology giving microarchitects regular feedback and insights.
- H-Houdini (and VeloCT) improve scalability helping verification engineers by drastically reducing the amount of time and effort needed to verify complex designs. Generalizing H-Houdini to prove properties beyond security is an interesting line of future work we're actively pursuing.
- With the security definition and tooling developed in this dissertation, hardware vendors can now ship security specifications as a part of their hardware release cycles. This enables principled software defenses against side-channels.
- Lastly, developers can focus on writing optimized functional code and can use SynthCT with the security specifications to generate high-performance, secure code specific to a microarchitecture.

1.1 Thesis Contributions and Organization

I have led the development of the results presented in this dissertation along with my co-authors: Grant Garrett-Grossman, Yongye Zhu, Madhusudan Parthasarathy, Christopher W. Fletcher. We will now present a brief overview of this dissertation and the salient contributions of each piece of work:

Chapter 2: ConjunCT. ConjunCT formulates the safe instruction set problem (SISP in Section 2.3): the formal foundation to microarchitectural security and a software-facing abstraction. The *key insight* is that the root-cause of many microarchitectural timing side-channels can be attributed to instruction operand-dependent optimizations. Therefore, proving that *all possible programs*, composed from a set of instructions, execute in time independent of operand data will give us the set of instructions that can safely compute on secrets. Chapter 2 is organized as follows:

- In Section 2.3 we develop SISP, the foundational property to describe and verify microarchitectures in this dissertation,
- In Section 2.5-Section 2.6, we develop the methodology to verify SISP, using existing verification techniques, to verify SISP. Using the methodology and tooling we developed, we automatically proved SISP, for the first time, on Rocketchip: a popular in-order core,
- ConjunCT is enabled by two technical nuggets: (i) Section 2.5.1 introduces a two-phased analysis to more accurately root-cause unsafe instructions, and (ii) Section 2.6.3 describes an invariant mining procedure that dynamically instantiates the necessary predicates during the verification phase,
- Beyond proving SISP, Section 2.8.4 shows how the synthesized inductive invariant helps us root-causes of unsafety in the design, and Section 2.8.5 shows how ConjunCT can inform microarchitects about the security of their proposed optimizations,
- We also provide proof-sketches regarding the soundness and precision of the derived invariants in Section 2.7.

Chapter 3: H-Houdini. H-Houdini is a new scalable algorithm for learning inductive invariants on large hardware designs. The *key insight* in H-Houdini is to use the hierarchical nature in hardware designs to incrementally learn an inductive invariant. This allows us to overcome the scalability limitations of existing Machine Learning-based Invariant Synthesis (MLIS) techniques. As a successor to ConjunCT, we implement H-Houdini, in a tool called VeloCT, to prove SISP. VeloCT, for the first time, scales up verification to large OoO cores like BOOM. In Chapter 3, we highlight the following:

- In Section 3.4 we describe the main algorithm along with the salient features that make it scalable,

- Section 3.6 dives into the implementation of VeloCT to prove SISP. Notably, Section 3.6.2 talks about practical issues when learning invariants with *noisy* traces,
- Supporting the description of H-Houdini (and VeloCT), Section 3.12 describes a running example of proving a property on a toy design,
- Section 3.7 evaluates VeloCT across a variety of open-source designs and shows H-Houdini's scalability characteristics wrt. design size and available parallelism,
- Lastly, proofs of soundness and completeness of H-Houdini is presented in Section 3.10 for interested readers.

Chapter 4: SynthCT. SynthCT develops a framework to automatically ensure that a binary conforms to the security specification. In short, SynthCT uses program synthesis to automatically translate any unsafe instruction in the binary into safe instructions for a particular microarchitecture. The *key insight* enabling SynthCT to scale to large complex ISAs, e.g., x86-64, is its ability to exploit the structural representation of instruction semantics to guide the synthesis search space. Using the techniques we develop in SynthCT, we're able to automatically synthesize translations for even the most complex instructions, like division, using simple safe instructions. At a high-level, Chapter 4 is organized as follows:

- Section 4.6 highlights the key challenges in scaling program synthesis to large complex ISA's like x86-64,
- In Section 4.7 we first develop the core technique, component selection (Section 4.7.1), which uses machine learning embeddings to estimate structural similarity between instructions,
- Then we develop supplementary techniques: (i) instruction factorization in Section 4.7.2, a divide-and-conquer that synthesizes instruction semantics in parts and stitches the solutions back together, and (ii) node splitting in Section 4.7.3 to expose structural similarity between instructions to aid component selection,
- We then evaluate SynthCT in Section 4.8. In particular we develop a safe-set agnostic evaluation methodology and show how SynthCT can support most (75%) of the ISA, using a small set (25%) of instructions. This small set of instructions match what experts believe are safe on today's microarchitectures,
- Finally, in Section 4.8.4 we show how the combination of techniques in SynthCT enable scalability to even the most complex instructions: in specific, we're able to automatically synthesize the divide instruction, known to be unsafe on today's microarchitectures, using only safe instructions.

Chapter 5: Conclusion & Future Work. Lastly, we will conclude by summarizing this dissertation and provide our thoughts on future directions.

All chapters are self-contained. As this dissertation straddles the areas of hardware security, formal methods, and program synthesis, the background and related work most relevant to each work is introduced in their respective chapters.

Chapter 2

ConjunCT:

Learning Inductive Invariants to Prove Unbounded Instruction Safety Against Microarchitectural Timing Attacks

This chapter addresses the foundational question: what does it mean for a microarchitecture to be safe? To answer this, we develop a formal, instruction set–centric property called the Safe Instruction Set Property (SISP). Once SISP is verified, it yields a set of instructions that can safely operate on secrets for unbounded durations. We also present the first automated analysis capable of proving SISP on hardware designs written in RTL. This chapter addresses Challenges: (C1), (C3), and (C5). The next chapter, Chapter 3, builds on these definitions and properties to scale verification to larger Out-of-Order designs, while Chapter 4 leverages the verified safe instruction set to harden software.

The past decade has seen a deluge of microarchitectural side channels stemming from a variety of hardware structures (the cache, branch predictor, execution ports, the TLB, speculation, etc). These attacks stem from software that passes sensitive data to so-called unsafe or transmitter instructions, i.e., those whose execution time depends on their operands' values. Correspondingly, there has been a large number of defenses (spanning hardware and software) that attempt to enforce the policy: sensitive data $\nrightarrow$ unsafe instruction operand. Implementing this policy assumes that one can identify unsafe instructions for a given microarchitecture. But this is quite difficult—requiring the designer to analyze potentially unbounded compositions of dynamic instructions to tease out subtle interactions they may have with one another.

This chapter addresses the above challenge by proposing ConjunCT. Given RTL: ConjunCT proves, for all possible executions, whether each ISA instruction is either i) safe for an unbounded number of cycles or ii) unsafe. This is done using a combination of symbolic analysis (to generate examples) and inductive invariant learning (bootstrapped by said

examples), and enabled by a novel conditional information flow predicate that we show is useful for analyzing information flows in processor pipelines.

We demonstrate our analysis on several RISC-V microarchitectures of varying complexity, and use it to extract the safe/unsafe sets for each. Through a judicious use of program synthesis, we are able to automate the analysis (almost entirely) from end to end, e.g., requiring only 8 designer annotations to fully analyze the RISC-V RocketChip core. Lastly, we show through several case studies how ConjunCT can be used by microarchitects to understand the security implications of an advanced optimization, and how the invariants generated by ConjunCT can be used to localize where in the microarchitecture unsafety occurs.

2.1 Introduction

It is well known that microarchitectural optimizations—such as the cache and branch predictor—leak program privacy through timing (or microarchitectural) side channels [90]. To comprehensively mitigate these attacks, a multitude of software and hardware defenses (some commercialized [17, 54, 92]) strive to enforce/support enforcing the following policy for sensitive programs: *that no so-called unsafe (or transmit) instruction in the program should compute on secret data (i.e., receive secret data as its operand) [126, 234, 235, 37, 84, 60, 164, 239, 153].* Here, an unsafe instruction is one whose execution creates observable timing differences as a function of its operand values. For example, in the widely deployed constant-time programming paradigm, high-value programs are carefully written by the programmer or compiler to enforce this policy [126, 23, 39, 234, 37].

Today, a major assumption made by all of the above defenses is that the set of unsafe instructions is known and correct. This assumption is perilous. Commercial-scale microarchitecture is incredibly complex and the hardware optimizations that lead to instruction unsafety leave only subtle footprints on the design (e.g., change only design timing, not functionality). Making matters worse, there is mounting evidence to suggest that future microarchitectures will include increasingly exotic optimizations, impacting instructions once thought to be safe [209, 210, 55, 64].

In light of the above, this paper develops ConjunCT: an automated analysis framework that, given a microarchitecture's RTL, determines which subset of instructions are safe (non-transmitters) for the given RTL.

The notion of when instructions are unsafe is hard to define and analyze. Instructions that are individually safe/secure may cease to be safe when composed, even with themselves, as leakage of data through side channels (including timing) can occur in any cycle, and through a sequence of state changes cause information flow. Many unsafe/insecure instructions can be detected using symbolic execution of bounded instruction compositions followed by logic satisfiability engines. However, proving that a set of instructions is safe under *unbounded composition* requires establishing an *inductive invariant* over microarchitectural states that proves that unbounded executions of these instructions keep secrets

unrevealed. These invariants are *relational invariants* that relate the states of any program composing these instructions executed on two different inputs that differ only on secret data. Such relational invariants are difficult to express (let alone discover) in the context of modern microarchitectures.

ConjunCT addresses these challenges and automatically constructs relational invariants using a combination of symbolic analysis and invariant learning, inspired from techniques in program synthesis [10, 9, 202, 157, 87]. This invariant is used to derive the set of safe and unsafe instructions in the ISA, and prove that the instructions in the safe set will never cause a safety violation *for an unbounded number of cycles*. We also show how the invariant can be useful for subsequent analyses, e.g., for pinpointing the 'root cause' of an instruction's unsafety.

We formulate the above as an invariant learning problem in the theoretical framework Sorcar [157] (which enhances the famous Houdini algorithm [82]), a methodology for learning conjunctive invariants from examples.

At the top level, the ConjunCT analysis proceeds in two phases. In Phase 1, we design a symbolic execution-based analysis (similar to 'bug finding' in other domains) that starting from *all possible* microarchitectural states, either: (i) finds that an instruction is unsafe, or (ii) proves that an instruction is safe for a bounded number of cycles. These results from Phase 1 are used to bootstrap inductive invariant learning in Phase 2.

In Phase 2, our first main contribution is to develop an invariant language that is rich enough to express relational invariants for real-world microarchitecures while at the same time being amenable to learning and minimizing designer intervention/annotation. Our key insight is that unsafety in designs can be expressed as conjunctions of *conditional* information flow rules, where each rule states that secret data should be allowed to flow into some state element s only if the value in some other state element(s) s' satisfies certain conditions. An example of such a rule might be: secrets should not be allowed to flow into the input latch of a variable-time multiplier (s) if another state element s' (that stores control values for that pipeline stage) indicates that the opcode is *multiply*.

Automatically synthesizing an invariant expressed as conjunctions of the above conditional information flow rules is non-trivial. That is, for a given s, there is an exponential number of collections of state elements s' (and functions over those state elements) that could be considered as predicates in the final invariant. Using information embedded in the results from Phase 1, we develop an automated procedure for synthesizing the overall invariant from the above rules, working in tandem with a verification engine.

Putting everything together, we implement our analysis and apply it to several microarchitectures (the RISC-V V-Scale, Ibex, and RocketChip cores). Our analysis is able to, relatively quickly (on the timescale of hours), analyze and derive invariants—along with unsafe/safe sets—for all three with 3, 7 and 8 expert annotations each, respectively.

Beyond safe set and invariant synthesis, we demonstrate usage scenarios and framework capabilities for ConjunCT in two case studies. First, we show how information contained in invariants synthesized by ConjunCT can be used to *localize* where (e.g., in what pipeline stage/state element) unsafety for a particular instruction originates in the de-

sign. Second, we show how ConjunCT enables microarchitects to reason about the safety of proposed microarchitectural optimizations. Specifically, we implement two computation reuse schemes [192], an exemplar advanced microarchitectural optimization, on top of the Ibex core. We use ConjunCT to confirm a hypothesis made in Vicarte et. al. [209]: while one of the two optimization variants creates new instruction unsafety, *the other does not*. This shows how ConjunCT can be used to assist in the design of *safe and performant* microarchitecture.

2.2 Contributions

In summary, this chapter makes the following contributions:

1. We propose ConjunCT, the first unbounded analysis for determining which instructions are safe for a given microarchitecture.

2. We propose an invariant language that is sufficiently rich to derive inductive invariants for real-world microarchitectures, along with automated techniques for selecting invariants from this language for a particular microarchitecture.

3. We implement and demonstrate how our analysis, based on symbolic execution as well as invariant synthesis, is able to deduce the safe set for three RISC-V microarchitectures of varying complexity (V-Scale, Ibex, RocketChip). Analysis for each took from minutes to a day, and required no more than 8 designer annotations per design.

4. We perform two case studies to demonstrate ConjunCT's capabilities. First, we show how ConjunCT can be used to localize where unsafety originates from in a design. Second, we show how ConjunCT can be used by microarchitects to evaluate the security properties of proposed microarchitectural optimizations.

2.3 Background and Motivation

There is a rich literature on how programs interacting with hardware resources (e.g., the cache [232, 161, 231], TLB [103], branch predictor [70, 71], functional unit execution ports [7, 26, 101], complex arithmetic instructions [14, 104, 49], speculative execution [125] and others [69]) can create side channels and leak program privacy.

Despite the apparent complexity in this space, however, the root cause of the above attacks can be attributed to a relatively small number of *unsafe instructions* whose execution timing is a function of their operand values. For example, the root cause of conflict/alias-based attacks in the cache, TLB, page walker, etc., is that a secret was passed to the address operand of a memory instruction [241, 150, 161, 103, 102]. Beyond memory instructions, several other instruction types (namely branch instructions [152, 126] and specific complex arithmetic operations [14]) are well-known transmitters. This has led to a line of hardware

and software defenses [126, 234, 235, 37, 207, 24, 152, 170, 81, 242] (and many more) that aim to prevent the flow of secrets to the operands of unsafe instructions. Notably, this defense policy is capable of mitigating not only 'classical' timing channels [126] but also the more recent speculative side channels [125, 235, 84, 207]. For example, Spectre attacks [125] are due to secret data being passed to the operands of unsafe instructions that are executing *speculatively*; this understanding is used to cast defenses in terms of well-known paradigms like constant-time programming [207, 239, 164, 44].

Thus far, a saving grace for the above defenses has been that (even across microarchitectures), the set of unsafe instructions has remained mostly unchanged. That is, branches are *inherently* unsafe because they influence the number of instructions executed by the program which in all practical scenarios influences the program's timing. Likewise, memory instructions are unsafe whenever the system supports a cache (which is to say, nearly always). This simplifies the above defenses: without a careful analysis of each target microarchitecture, they can disallow secret-dependent flows to a fixed and known set of instructions.

This paper's premise is that determining each microarchitecture's set of unsafe instructions will become a more difficult problem as we continue to develop microarchitecture in the post-Moore era. Specifically, as scaling slows, one avenue to continue improving performance is to implement *software-invisible optimizations* (or fast paths) to different instructions [209, 55] to optimize their common case behavior. Vicarte and Deng et al. [209, 55] describe several families of optimizations that fit this mold:

- Computation simplification / elimination optimizations (e.g., [172, 233, 19]) have been proposed for many arithmetic operations to take advantage of operation-specific identity and absorption properties. For example, that $x \,\&\, 1 = x$.
- Computation reuse optimizations (e.g., [192, 191, 151]) memoize computation when the same instruction(s) are executed twice with the same operands.
- Value prediction (e.g., [149, 175, 185]) saves cycles when an instruction returns a predictable result.
- Significance compression (e.g., [30, 34, 213]) impacts performance depending on the position of the high-order on bit in each program word.
- Silent stores (e.g., [133, 122, 64]) impact whether stores need to be performed by inspecting the contents of memory at the store address.

These optimizations significantly complicate security audits on processor pipelines. For example, Vicarte et al. [209] describes how:

- The above optimizations are seldom implemented in the processor's 'Execute' stage / ALUs. For example, even computation simplification [172], which is typically associated with 'Execute', is often implemented in an earlier stage (e.g., register file read) to increase its performance benefit.

- The above optimizations may only activate based on the combined behavior of *multiple* in-flight instructions. For example, operand packing [30] only activates when two arithmetic instructions co-located to the same execution port both have 'narrow' operands, i.e., operands whose most-significant 1 bit is in a low bit index.
- The above optimizations may leave microarchitectural traces that modulate channels long-after the offending instruction retires. For example, silent stores [133] may not effectuate a performance improvement until the store in question is at the head of the store queue (i.e., after the store officially retires).

Putting the above together, auditing a pipeline to determine which instructions are unsafe may soon become highly non-trivial: requiring analyses a) over multiple pipeline stages and interactions across stages (as opposed to 'just', say, auditing the Execute stage logic in isolation), b) that explore how different combinations of instructions interact with each other (as opposed to analyzing each in isolation), and c) that analyze pipeline state for an unknown number of cycles after instructions finish their execution/retire.

Summary of our analysis. Section 2.4-Section 2.6 proposes a framework and automated analysis that discovers, given a processor's RTL as input, which instructions are unsafe on that RTL. Our analysis considers arbitrary compositions of instructions and their executions over an unbounded number of cycles over the entire pipeline, and hence is able to cope with the nuances of the hardware optimizations described above.

Our technical contribution is broken into three sections. First, Section 2.4 defines the problem. As summarized in Section 2.1, the analysis itself is broken into two phases. First, we perform a bounded analysis over a fixed number of cycles (Section 2.5) which generates examples, along with a preliminary set of unsafe and potentially-safe instructions. Second, we use the examples/preliminary unsafe set to bootstrap invariant learning (Section 2.6) and prove safety of instructions for an unbounded number of cycles.

2.4 Preliminaries and Problem Definition

Let us fix a design-under-test $\mathcal{D}$ with a finite set of state variables $\mathbb{V}$ in $\mathcal{D}$. The set $\mathbb{Z}_{bv}$ is a domain of n-length bit vectors, for some n, as in the width of elements in $\mathbb{V}$.

Definition 2.4.1. State (s): A state is a mapping $\mathbb{V} \rightarrow \mathbb{Z}_{bv}$.

Let $\mathbb{S}$ denote the set of all states. Let us fix a set of opcodes $Opcodes$. For each opcode, we fix a set of parameter variables $\overrightarrow{pv}$, i.e., operand labels. An instruction is a tuple $(opcode, \overrightarrow{pv})$, where $opcode \in Opcodes$.

Definition 2.4.2. State Machine (C): A finite state machine C is a tuple $(\mathbb{S}, s_{init}, R, \Sigma, O, \rightsquigarrow)$. Here $s_{init} \in \mathbb{S}$ is the special initial state, e.g., the reset state of the machine. $R \subseteq \mathbb{V}$ act as sources of secret data. Σ is a set of input symbols. Each input symbol is an instruction

of the form $(opcode, \vec{pv})$ or is the special symbol ϵ (no instruction).[1] $O \subseteq \mathbb{V}$ is a set of output variables or attacker-observable variables/the attacker's view. The partial function $\rightsquigarrow: \mathbb{S} \times \Sigma \to \mathbb{S}$ is the state transition function that maps a state and input symbol to the next state.

Let $s \rightsquigarrow^a s'$ denote that $\rightsquigarrow (s, a)$ is defined and is equal to s' where $a \in \Sigma$.

Definition 2.4.3. Trace (π): Let w represent a sequence of instructions $a_0, a_1, \ldots, a_n$. A trace of C over w starting at s_0 is a finite sequence of states of the state machine C $s_0 \rightsquigarrow^{b_0} s_1, \rightsquigarrow^{b1} \ldots \rightsquigarrow^{b_m} s_m$ where each $b_i \in \Sigma$ (an instruction or ϵ) and such that $w = b_1 \ldots b_m$.

Note that in the above, the sequence $b_1, \ldots, b_m$ may include ϵ, and the condition $w = b_1 \ldots b_m$ says that the concatenation of the b_i's (where ϵ vanishes) is equal to w.

Let $s \downharpoonright O$ denote the projection of the state s onto O.

Definition 2.4.4. Trace Distinguishability: Two traces π, π' over a sequence w (with different start states) are trace distinguishable if they are of different lengths, or they are of the same length with $\pi = s_0, s_1, \ldots, s_n$ and $\pi' = s'_0, s'_1, \ldots, s'_n$ such that for some $j \in [0, n]$, $s_j \downharpoonright O \neq s'_j \downharpoonright O$.

Definition 2.4.5. Equal-modulo-secret ($\approx^{sec}$): Let $\approx^{sec}$ be the relation over $\mathbb{S} \times \mathbb{S}$ such that, $\approx^{sec}$ relates two states s, s' if $\forall_{v \in \mathbb{V} \setminus R}\ s[v] = s'[v]$, where $\mathbb{V} \setminus R$ denotes set difference and $s[v]$ is the value of v on s.

In other words, two states are equal-modulo-secret if the values of non-secret variables are the same.

Definition 2.4.6. Safe Instruction Set Problem: Find a *maximal* $\Sigma^+ \subseteq \Sigma$ (the set of *safe instructions* on $\mathcal{D}$) such that: for every sequence of instructions x over Σ^+, and for every $(s^{\mathsf{L}}, s^{\mathsf{R}})$ where $s^{\mathsf{L}} \approx^{sec} s^{\mathsf{R}}$, the pair of traces $(\pi^{\mathsf{L}}, \pi^{\mathsf{R}})$ of C over x starting from states $(s^{\mathsf{L}}, s^{\mathsf{R}})$, respectively, are *not* trace distinguishable.

We instantiate the above framework for microarchitectural designs-under-test $\mathcal{D}$, where the state machine C captures the execution semantics of $\mathcal{D}$ written in Verilog and $\mathbb{V}$ is the set of state elements (or registers) in $\mathcal{D}$. The safety of instructions needs to hold for traces of unbounded length.

The next two sections develop the symbolic execution-based bounded analysis (Section 2.5) to determine the set of *potentially safe* instructions, i.e., that are candidates for inclusion in Σ^+. More precisely, we discard instructions (opcodes with parameters) that clearly leak secrets. Later, in Section 2.6, we learn an invariant that will *prove* the safety of instructions and their composition, thereby deriving the safe set of instructions Σ^+ as defined above.

Remarks regarding Def. 2.4.6. We make two remarks regarding the definition.

First, we ask for *any maximal* safe set as a unique maximum safe set may not exist and the maximal safe sets may not be comparable. Ideally, we would like an analysis to be

[1] Like a processor, the machine may not take a new instruction in each step, if a step is a cycle.

Name	Symbol	Description
Design	$\mathcal{D}$	Microarchitecture design-under-test.
Concrete State	s	A state with concrete assignment to state elements, usually generated by a counterexample.
State	$\mathbf{S}$	A mapping of all microarchitectural state elements, e.g., wires and registers, to both concrete and symbolic values.
Input Vocabulary	Σ	Set of all instructions plus the special symbol ϵ.
State Transition	$\rightsquigarrow$	A partial function mapping $\mathbb{S} \times \Sigma \to \mathbb{S}$
Trace	π	A sequence of states $\mathbf{S}_0 \rightsquigarrow^{a_0} \mathbf{S}_1 \rightsquigarrow^{a_1} \ldots \rightsquigarrow^{a_{n-1}} \mathbf{S}_n$.
Instruction	a	An instruction from the ISA.
Instruction Under Test (IUT)	a_{IUT}	Instruction whose execution we are analyzing for safety.
Program	P	A static sequence of instructions represented as a word w over Σ.
Attacker View	$\mathbf{S} \downharpoonright O$	A projection of state $\mathbf{S}$ to attacker observation variables (O).
Secret Sources and Data	R, D_{secret}	$R \subseteq \mathbb{V}$ annotated to be sources of secret data (D_{secret}).
Safety	$Safe(\mathbf{S})$	A predicate that evaluates to `true` if $\mathbf{S}$ is *safe*, and `false` otherwise.
Unsafe Set	$\mathbb{U}$	Set of unsafe instructions output by the bounded analysis.

Table 2.1: A summary of all definitions used in ConjunCT.

able to list out all such maximal safe sets. However, we note that in our evaluation, all the maximal safe sets were unique and also the maximum. Hence, this alternative formulation of the problem does not yield any additional safe sets.

Second, certain usage scenarios (e.g., constant-time programming) assume that unsafe instructions can be executed in composition with safe instructions, subject to the constraint that unsafe instruction operands only see non-secret data. Our definition only concerns compositions of safe instructions. We note that the invariants generated in Section 2.6 do not preclude injecting unsafe instructions, and will soundly detect when doing so can violate security. However, the current analysis does not provide guarantees on precision in the regime where unsafe/safe instructions are composed. That is, it will not necessarily recognize that a given *safe composition* of safe/unsafe instructions is indeed safe. We leave addressing this issue to future work.

2.5 Phase 1: Bounded Analysis

In this section, we build on the terminology defined in Section 2.4 to develop our symbolic execution-based analysis on hardware designs described, e.g., in Verilog. For convenience, terms defined are summarized in Table 2.1.

We build on previous definitions and define a Program (P) as a sequence of instructions $a_0, a_1, \ldots, a_n$ where $a_j \in \Sigma$. The transition function $\rightsquigarrow^a$ is derived from the execution

semantics of $\mathcal{D}$, e.g., written in Verilog.

ConjunCT analyzes the safety of each instruction in $a \in \Sigma$ using symbolic execution. We denote the current instruction-under-test (IUT) as a_{IUT}. As we want to capture all possible interactions of a_{IUT} with other instructions in the pipeline, we consider the analysis of a_{IUT} as a part of a larger program $P = a_{IUT} \,\|\, P_s$, where P_s is any possible suffix program.[2]

We augment the definition of state s from Def. 2.4.1 by allowing assignment of *symbolic values* to variables. To disambiguate from here on, an uppercase boldface $\mathbf{S}$ refers to states that may have a symbolic or concrete assignment to each variable, while the lowercase s refers to states with concrete assignments only.

Constructing all pairs of $(s^{\mathsf{L}}, s^{\mathsf{R}})$. As analyzing the execution of a_{IUT} from every possible pair of concrete states is not possible, we analyze the execution from a fully symbolic start state ($\mathbf{S}$) instead.[3] We first duplicate a fully symbolic state $\mathbf{S}$ to obtain $(\mathbf{S}^{\mathsf{L}}, \mathbf{S}^{\mathsf{R}})$. We extend the definition of $\approx^{sec}$ on concrete states to operate on symbolic states. We say two symbolic states $\mathbf{S}, \mathbf{S}'$ are $\mathbf{S} \approx^{sec} \mathbf{S}'$ when $\forall_{v \in \mathbb{V} \setminus R} \mathbf{S}[v] \equiv \mathbf{S}'[v]$, where $\equiv$ is symbolic equivalence between the expressions which could be implemented, e.g., using an SMT solver. Note that $\mathbf{S}^{\mathsf{L}} \approx^{sec} \mathbf{S}^{\mathsf{R}}$ trivially as all state elements are equal. Next, $\forall_r r \in R$ we set $\mathbf{S}^{\mathsf{L}}$ and $\mathbf{S}^{\mathsf{R}}$ to hold different (secret) symbolic values. We say a variable $v \in \mathbb{V}$ is *symbolic constrained* if it is symbolic and is equal in L and R ($\mathbf{S}^{\mathsf{L}}[v] \equiv \mathbf{S}^{\mathsf{R}}[v]$) and *symbolic unconstrained* if v is symbolic and *need not be* equal in L and R. Now, the pair $(\mathbf{S}^{\mathsf{L}}, \mathbf{S}^{\mathsf{R}})$ are still $\mathbf{S}^{\mathsf{L}} \approx^{sec} \mathbf{S}^{\mathsf{R}}$ and represent all possible pairs of $(s^{\mathsf{L}}, s^{\mathsf{R}})$. Note that this may also include states that are unreachable in the microarchitecture.

Product Construction for Two-Safety. For convenience, we can construct a product state $\mathbf{S} = (\mathbf{S}^{\mathsf{L}} \cdot \mathbf{S}^{\mathsf{R}})$, where $\cdot$ represents concatenation. This is the well-known construct of a product program [8, 21] used for checking two-safety properties such as non-interference and is equivalent to a miter circuit used in the architecture community [74]. From here on, $\mathbf{S}$ will refer to a product state that holds variables for both L and R executions of the microarchitecture.

Safe$_O(\mathbf{S})$**.** We define a predicate *Safe* that evaluates to `true` on a state $\mathbf{S}$ if $\mathbf{S}$ is safe: Formally, let $(\mathbf{S}^{\mathsf{L}} \cdot \mathbf{S}^{\mathsf{R}}) = \mathbf{S}$. $\mathbf{S}$ is *Safe* wrt. O if $\mathbf{S}^{\mathsf{L}} \downharpoonright O \equiv \mathbf{S}^{\mathsf{R}} \downharpoonright O$. Note that we will omit the subscript and say *Safe*($\mathbf{S}$) (or *Safe*) when the observer model (and state $\mathbf{S}$) is clear from the context.

Defining Secret Data (R). We require that the designer annotate the secret sources R. In the safe instruction set problem, R is set to the architectural register file. (This special case of Def. 2.4.6 is formalized in Section 2.7.1.) We will refer to the (symbolic unconstrained) secret data released by the register file as D_{secret}.

[2] Although we write P as having a suffix but not prefix program, the application of P in Section 2.5.1 will be equivalent to considering P with an arbitrary prefix program.

[3] Such a fully symbolic state $\mathbf{S}$ can represent all possible states $s_i \in \mathbb{S}$.

Controlling the Release of Secret Data. Since the analysis is over a single instance of a_{IUT} (but in the context of an infinitely long P), we further need to limit the release of D_{secret} so that only a_{IUT} receives it. Thus, our symbolic interpreter treats $\vec{pv}_{IUT}$, of a_{IUT}, differently from those of other instructions and only releases D_{secret} from the register file when it is being accessed by a_{IUT}. This does not prevent a_{IUT} from forwarding a function of D_{secret} to later instructions. We address this in Section 2.5.1.1.

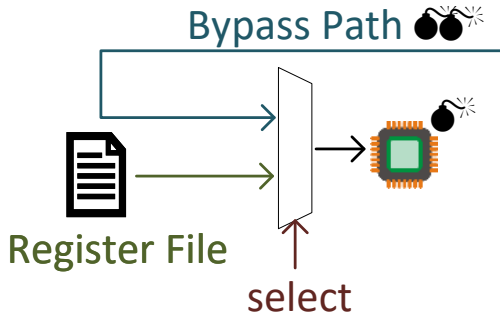


Figure 2.1: Multiple sources for operand values. Operand values for instructions may either arrive from the register file or along the bypass path and is selected by the control variable to the MUX, `select`. Each 💣 represents a potential source of unsafety.

Soundness of analysis in the presence of corner cases. In the above, we only annotate the architectural register file as R and only release secrets from R in a single cycle (when they are read for each a_{IUT}). Can this miss cases where the instruction computes on data from a bypass path (see (ii) in Figure 2.1)? Can this miss cases where *multiple* a_{IUT} need to receive secret data and interact for unsafety to manifest? We prove in Section 2.7.1 that our complete analysis is sound and handles these cases, while requiring only the above specified annotation burden.

2.5.1 Symbolic Execution

Using the ideas from the previous subsection as building blocks, we now describe the bounded analysis (the first phase of ConjunCT). The goal is to discover a preliminary set of unsafe instructions and "potentially safe" instructions (i.e., those where no security violation was found for a bounded number of cycles), along with their execution traces. These will be used to bootstrap the invariant learning stage (Section 2.6).

This process is, itself, two parts. The high-level algorithm for both parts is shown in algorithm 1 and takes the following as inputs: O: the attacker observation variables, K: the max number of cycles to run the analysis for, $\mathbb{I}_{IUT}$: the $\mathbb{I}_{IUT} \in \Sigma$ representing the instruction-under-test (IUT), and P_s: the suffix program to execute after the IUT, and returns either Safe indicating that $\mathbb{I}_{IUT}$ is safe up to the bound K or Unsafe with a counterexample that violates the safety property. The two parts of the bounded analysis involve calling the above algorithm twice, with different arguments passed to the P_s parameter each time.

The analysis starts from a `symbolic-start-state` which initializes a blank state where all microarchitectural variables in **S** are *symbolic and constrained to be equal* on the left and

Algorithm 1: Symbolic Execution-based analysis.

Data: O, K, $\mathbb{I}_{IUT}$, P_s
Result: Safe or Unsafe(cex)

```
1  S = symbolic-start-state();
2  P' = a_IUT || P_s;
3  for i ∈ (0...K) do
4  |   a = pop(P');
5  |   S' = ⇝(S, a);
6  |   cex = check-safety(S', O);
7  |   if cex then
8  |   |   return Unsafe(cex);
9  |   end
10 |   S = S'
11 end
12 return Safe;
```

right executions of the design. See Section 2.5 "Constructing all pairs..." for details. Then, on line 2 the algorithm constructs P', a concatenation of the IUT a_{IUT} and the suffix program P_s. Now, in each step, the next instruction a is popped from P' and used to generate the new state $\mathbf{S}'$ by evaluating $\rightsquigarrow (\mathbf{S}, a)$. Next, the `check-safety` function uses O to check if the state $\mathbf{S}'$ is *Safe* by querying an off-the-shelf SMT solver, e.g., Z3 or CVC5. If the $Safe_O(\mathbf{S}')$ evaluates to `true`, then the state passes the safety check. Otherwise, `check-safety` returns Unsafe and a counterexample to the safety property, i.e., an assignment to variables in $\mathbf{S}'$ that leads to the safety violation. A counterexample indicates that one or more of the assertions are violated, in which case the design is unsafe with respect to O and $\mathbb{I}_{IUT}$. Therefore, the overall analysis terminates with Unsafe and returns the counterexample. The loop is repeated for a maximum of K iterations by popping the next instruction a from P_s in each iteration, and returns Safe if safety is not violated for K steps.

2.5.1.1 Attributing Blame for Unsafety

As we alluded to earlier, younger instructions in the pipeline (part of P_s) may eventually interact with secret data from $\mathbb{I}_{IUT}$ through architectural data dependencies with $\mathbb{I}_{IUT}$, microarchitectural state set by $\mathbb{I}_{IUT}$, etc. This creates a blame attribution problem: if a safety violation occurs due to the execution of $\mathbb{I}_{IUT}$ followed by some instruction a', should $\mathbb{I}_{IUT}$ be deemed unsafe, or should a'? Suppose a' *is actually* unsafe. In that case, we risk incorrectly blaming $\mathbb{I}_{IUT}$, which could lead to false positives in the overall analysis.

To solve the above problem, we perform the above symbolic analysis in two parts while varying the instructions allowed in the suffix program P_s. In part (i), we try to find instructions whose executions are unsafe independently, i.e., we set P_s to only contain `nop`'s. Therefore, any safety violation we find in this phase can be attributed to the instruction-

under-test $\mathbb{I}_{IUT}$ (as the only source of unconstrained data is from the register read on behalf of $\mathbb{I}_{IUT}$). From this part (i) we get a list of unsafe instructions, $\mathbb{U}$, and a list of *potentially safe* instructions $\widehat{\Sigma^+_{(i)}}$. Here, the subscript (i) refers to the safe set after phase (i) of the bounded analysis. Note that by only allowing `nop`s in this phase, we have bypassed the issue from the previous paragraph.

Next, in part (ii), we re-analyze the instructions $\widehat{\Sigma^+_{(i)}}$ for safety while constraining the suffix program P_s to only contain instructions from $\widehat{\Sigma^+_{(i)}}$. Any safety violation now is due to interactions between $\mathbb{I}_{IUT}$ and one or more instructions $\mathbb{I}'$ in P_s. In this case, we make a conservative assumption and treat both $\mathbb{I}_{IUT}$ and $\mathbb{I}'$ (i.e., all instructions a' in the suffix) as unsafe. This could be optimized for improved precision in a variety of ways. For example, with more expressive specifications we believe we could more accurately capture that the *interaction* of $\mathbb{I}_{IUT}$ and a *specific* $\mathbb{I}'$ is unsafe, but leave this for future work. The implications of this in the precision of the overall analysis is discussed in Section 2.7.2.

At the end of this two-phase symbolic analysis we have a list of instructions known to be safe for K cycles in arbitrary composition with other potentially safe instructions (denoted $\widehat{\Sigma^+_{(ii)}}$) and a set of unsafe instructions $\mathbb{U}$.

We remark that $\widehat{\Sigma^+_{(ii)}}$ satisfies the requirements for a safe set in Def. 2.4.6, but only for the bounded K cycles.

2.5.2 Generating Examples for Learning

In addition to identifying the set of unsafe instructions $\mathbb{U}$, the bounded analysis also outputs a set of examples used in the next phase (invariant learning). In this context, each example is a microarchitectural state $\mathbf{S}$ we encounter during the bounded analysis and contains a mix of concrete and symbolic assignments to state variables. We generate two types of examples: (i) *positive examples* are states that are *Safe* and not known to lead to any unsafe states in the bounded analysis, and (ii) *negative examples* are states that are either not *Safe* or are known to lead to states that are not *Safe* in the bounded analysis.

We now discuss each of these in more detail. Consider the sequence of states in the trace generated by the symbolic execution of an IUT, a_{IUT}: $\mathbf{S}_0, \mathbf{S}_1, \ldots, \mathbf{S}_K$.

If a_{IUT} is safe, then none of the states $\mathbf{S}_0, \mathbf{S}_1, \ldots, \mathbf{S}_K$ are unsafe. In other words, all possible concrete s represented by $\mathbf{S}_i$ are safe. We will directly use each of the symbolic states $\mathbf{S}_0, \mathbf{S}_1, \ldots \mathbf{S}_K$ as separate positive examples.

On the other hand, consider if a_{IUT} was unsafe. In this case, one of the $\mathbf{S}_u$ fails the safety check and we get a counterexample (*cex*), i.e., a concrete assignment of values to $\mathbb{V}^* \subseteq \mathbb{V}$ that causes unsafety. Note that the *cex* gives us a concrete state, i.e., an s_u that is actually unsafe, while other concretizations of $\mathbf{S}_u$ may still be safe. Therefore, we use the information in the *cex* to concretize each of the symbolic states $\mathbf{S}_0, \mathbf{S}_1, \ldots$ to generate concrete states $s_0, s_1, \ldots$, i.e., the sequence of concrete states that will eventually lead to the concrete unsafe state s_u. Each of these concrete states s_i are used as separate negative examples.

2.6 Phase 2: Invariant Learning

The bounded analysis (Section 2.5) is useful to find instructions that are unsafe but, being a bounded analysis, cannot prove that an instruction that has been safe for K cycles will remain safe under unbounded composition. The goal of this section is to do exactly that: prove that a set $\widehat{\Sigma^{+}_{(ii)}} \subseteq \Sigma$ of *potentially safe* instructions—instructions that have remained safe for K cycles in the bounded analysis—remain safe forever.

To prove safety, we need to show that starting from a safe state $\mathbf{S}$ we cannot reach an *unsafe state* through one or more applications of $\rightsquigarrow^{a}$, where a is any instruction in the ISA. To do this, we will define an invariant $\mathbf{H}$ such that for a state $\mathbf{S}$ and invariant $\mathbf{H}$,

$$\mathbf{S} \models \mathbf{H} \implies Safe(\mathbf{S}) \tag{2.1}$$

For this safety check to hold for an unbounded number of cycles, we require $\mathbf{H}$ to be inductive. That is, satisfy a base case and inductive step. Let $\mathcal{P}$ be the set of positive examples discovered during the bounded analysis. For the base case, we require that $\mathbf{H}$ allow all such positive examples: $\forall p \in \mathcal{P}, p \models \mathbf{H}$. To satisfy the inductive step, we require that

$$(\mathbf{S} \models \mathbf{H}) \wedge (\mathbf{S} \rightsquigarrow^{a} \mathbf{S}') \implies \mathbf{S}' \models \mathbf{H} \quad \forall a \in \widehat{\Sigma^{+}_{(ii)}} \tag{2.2}$$

An $\mathbf{H}$-state is any state $\mathbf{S}$ that satisfies $\mathbf{H}$. Therefore, any $\mathbf{H}$-state is *Safe*. Together, Equation 2.1 and Equation 2.2 guarantee that starting from an $\mathbf{H}$-state we can never reach an unsafe or non-$\mathbf{H}$-state. Putting it all together, if $\mathbf{H}$ holds for a state corresponding to all possible executions of a potentially safe instruction, then we prove that the instruction remains safe for an unbounded number of cycles.

Both Equation 2.1 and Equation 2.2 are checked using an SMT solver (like CVC5). As both equations need to hold for *every* state allowed by $\mathbf{H}$, we perform the check on the most permissive symbolic state $\mathbf{S}$ allowed by $\mathbf{H}$. Such a state $\mathbf{S}$ is constructed by first initializing a fully symbolic state and then constraining $\mathbf{S}$ based on $\mathbf{H}$.

Approach to construct H. The principle challenge in the above is how to find an $\mathbf{H}$ for a given design $\mathcal{D}$ that is both sound and precise, i.e., does not induce false positives (preclude safe executions) or negatives (allow unsafe executions). We would also, ideally, like for our invariant to be *minimal*, i.e., contain as few predicates as possible. Such 'smaller' invariants typically lead to both improved analysis time—for both our and any subsequent analysis [157]—and as we later show in Section 2.8.4 is also useful in root cause analysis. Constructing such an $\mathbf{H}$ by hand is impractical. Instead, our approach is to *automatically synthesize* $\mathbf{H}$.

At the high level, we follow the blueprint established by the invariant learning framework called SORCAR [157]. SORCAR describes a theoretical framework for learning conjunctive invariants from examples, which is an improvement over the well-known Houdini algorithm [82], for learning invariants when the number of predicates is very large. SORCAR is a theoretical framework that works by proposing invariants in each round along with a verification engine

that produces counterexamples to incorrect invariants. SORCAR takes as input a large number of predicates, selectively chooses predicates to include in the invariant and guarantees convergence to an inductive invariant using a number of rounds that is linear in the number of predicates.

Adapting SORCAR to our setting is non-trivial as SORCAR leaves open many design decisions when it comes to solving the safe instruction set problem. First, setting up a self-product transition system to invoke SORCAR on so that it solves the safe instruction set problem precisely is nontrivial (see Section 2.7.2). We also need to specify/generate the positive/negative samples to bootstrap the learning algorithm (see Section 2.5) and we need to find mechanisms to recover from failure when an invariant is not found (see Section 2.6.5).

Beyond the above, our main conceptual contribution (also not covered in the SORCAR work as it is domain agnostic) is to define an appropriate universe of predicates through which to express inductive invariants **H**. Choosing too large a universe would make the analysis intractable or require significant designer intervention (e.g., to provide annotations/analysis constraints). Choosing too few predicates would lead to invariants not being expressible for given designs. In Section 2.6.1, we describe an invariant grammar that is sufficiently rich to enable analysis of several recent microarchitectures (e.g., the pipelined RISC-V RocketChip [18]). We then describe in Section 2.6.3 an algorithm that makes finding invariants in said grammar tractable without additional expert annotation burden. Putting it all together, our whole analysis run on the RISC-V RocketChip required only 8 annotations and was able to generate an invariant in 10 hours.

Tolerating unsafe instructions. Proving that a set of instructions is safe using an invariant **H** has further benefits. In particular, we can allow *unsafe* instructions to execute in states satisfying **H** provided that results in states that remain in **H** (we cannot allow unsafe instructions in other states, of course). Hence, finding a larger semantic invariant (i.e., a 'minimal' invariant made up of fewer predicates as discussed before) is also a useful heuristic for admitting more safe compositions of safe/unsafe instructions. We discuss how to obtain such minimal invariants in Section 2.6.4.

2.6.1 Language for H

We use the DSL shown in Figure 2.2 to learn and express the invariant. The hypothesis space of this DSL is tailored to be able to express inductive safety invariants for designs we expect to encounter in practice.

In this DSL, the $\mathsf{Eq}(v)$ predicate expresses the equality between the L and R versions of a variable v, i.e., $v_{\mathsf{L}} = v_{\mathsf{R}}$. This is similar to secrecy assumptions in prior work [94]. With this predicate, we say v is *constrained* to be equal in the L and R executions. Intuitively, this means that v can only store non-secret data, i.e., data whose value is independent of D_{secret}.

Beyond $\mathsf{Eq}(v)$, we include a higher-level predicate $\mathsf{Impl}(v, vs)$ to express more complex relationships between state variables in modern microarchitectures. $\mathsf{Impl}(v, vs)$ allows for a

$\langle H \rangle ::= \langle p \rangle \wedge \langle H \rangle$
| $\langle empty \rangle$

$\langle p \rangle ::=$ Eq ($\langle state\_element \rangle$) ; Equality constraint
| Impl ($\langle state\_element \rangle$, $\langle condition \rangle$)

$\langle state\_element \rangle ::= s$; State-element in $\mathcal{D}$

$\langle condition \rangle ::= \mathbb{C} : \mathfrak{P}(\mathbb{V}) \rightarrow \{$ `true`, `false` $\}$

Figure 2.2: DSL for synthesis of $\mathbf{H}$. $\mathbb{C}$ is a boolean conditional over a subset of $\mathbb{V}$. $\mathfrak{P}(\mathbb{V})$ is the power set of $\mathbb{V}$.

variable v to conditionally hold *unconstrained* (not equal) values, i.e., secret values, when certain conditions are true. For example, the values read by an instruction from a register file are allowed to be secret if the instruction currently executing in the pipeline is not an unsafe instruction. More concretely, the Impl(v, vs) predicate adds the constraint: $v_l \neq v_r \implies \mathbb{C}(vs)$ where, $\mathbb{C}$ is a condition on state element(s), $\mathbb{C} : \mathfrak{P}(\mathbb{V}) \rightarrow \{$ `true`, `false` $\}$, where $\mathfrak{P}(\mathbb{V})$ is the power set of $\mathbb{V}$.

Deciding what Impl predicates to include in an invariant $\mathbf{H}$ is more difficult than choosing which Eq predicates to include. Specifically, Impl implies a predicate space of $O(2^{|\mathbb{V}|})$, times the complexity of choosing an appropriate $\mathbb{C}$ which is exponential in $|vs|$. Eq implies a predicate space of $O(|\mathbb{V}|)$. To address this, Section 2.6.3 describes an algorithm for efficiently selecting a small number of useful Impl predicates to consider during invariant synthesis.

Finally, following SORCAR/Houdini, we permit conjunctions of individual predicates (Eq and Impl). Conjunctions are sufficient to represent safety invariants for a large class of problems in practice [25, 130, 146] (including the microarchitectures we studied) and also admit an efficient analysis.

2.6.2 Learning H

With the invariant DSL from the previous section, we now describe a high-level overview of our learning algorithm, shown pictorially in Figure 2.3. We define a function `check-ind-safe?` that takes a candidate invariant, $\mathbf{H}_{cand}$ and outputs either a counterexample (a negative/implication example) or successfully proves that the candidate invariant is safe/inductive and outputs $\mathbf{H}_{cand}$ as the inductive invariant $\mathbf{H}_{ind}$. Internally, `check-ind-safe?` performs a safety (Equation 2.1) and inductivity (Equation 2.2) check through an SMT query.[4]

All the examples (*positive, negative*, and *implication* examples) shown in the figure are seeded from the bounded analysis (Section 2.5). Initially, the set of implication examples is

[4]As discussed in Section 2.6, we check this property for the most generic (symbolic) state admitted by $\mathbf{H}_{cand}$. The checks are decidable as we only use quantifier-free bit-vector theories from SMT.

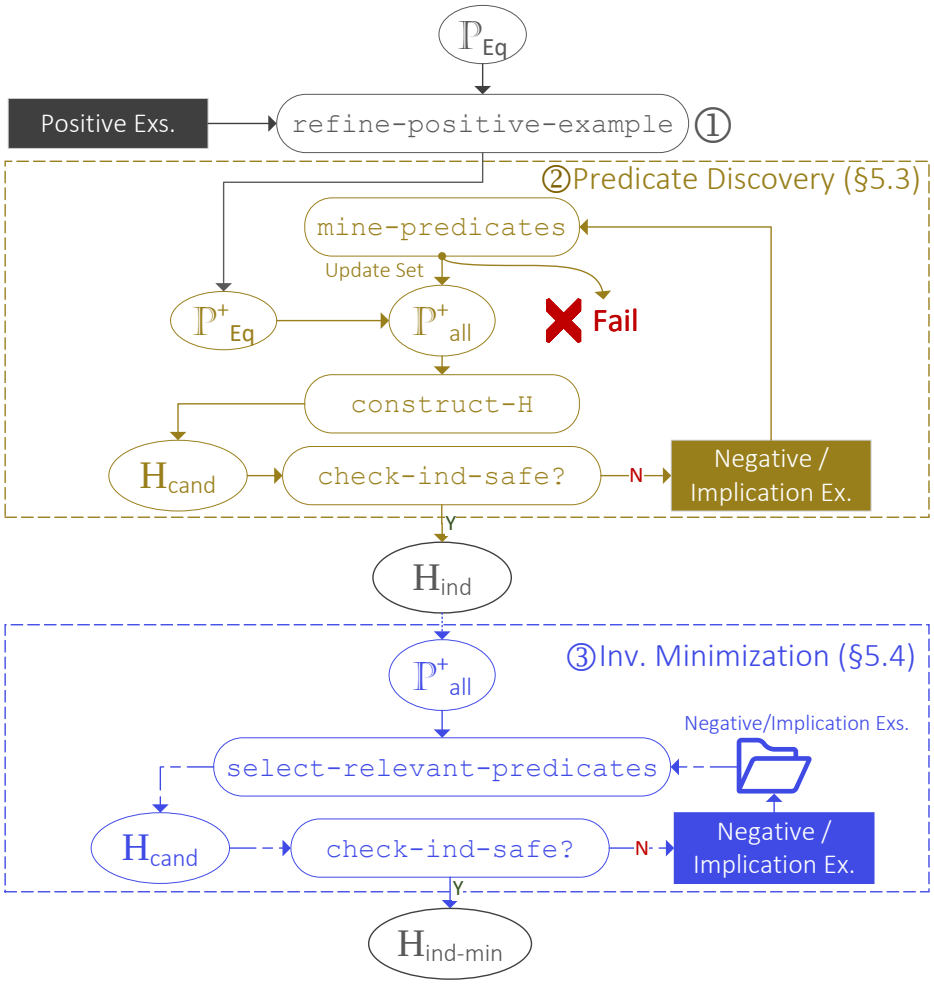


Figure 2.3: Workflow diagram showing learning of the inductive invariant **H**. `construct-H` is a procedure that generates an invariant $\mathbf{H}_{cand}$ by taking a conjunction over $\mathbb{P}^+_{all}$.

empty. The algorithm then proceeds as follows:

Step 1. To prove safety of instructions, the invariant should admit all the positive examples. Initially, we enumerate the set of predicates of type Eq, $\mathbb{P}_{Eq}$. However, we're only interested in a subset of $\mathbb{P}_{Eq}$ that actually hold on the positive examples. Therefore, using the procedure `refine-positive-example` and the set of all positive examples, we filter $\mathbb{P}_{Eq}$ to obtain the set of predicates, $\mathbb{P}^+_{Eq}$, that are satisfied by all positive examples.[5] The predicate discovery algorithm takes this set $\mathbb{P}^+_{Eq}$ as the initial set of all predicates $\widehat{\mathbb{P}^+_{all}}$.

Step 2. As the set of predicates, $\widehat{\mathbb{P}^+_{all}} = \mathbb{P}^+_{Eq}$, is usually insufficient to express an invariant

[5]In the following, we will use superscript + when discussing predicates, e.g., $\mathbb{P}^+$, to denote sets of predicates that satisfy the positives examples.

for real-world designs, we need to augment it with a set of Impl predicates. But, we cannot enumerate the set of predicates $\mathbb{P}_{Impl}$ as it is exponential in size. So, we develop a predicate discovery algorithm to find a subset of Impl predicates that are sufficient to derive an inductive invariant.

Predicate discovery takes as input an initial set of all predicates $\widehat{\mathbb{P}^{+}_{all}} = \mathbb{P}^{+}_{Eq}$ and outputs $\mathbf{H}_{ind}$ of the form $\mathbf{H}_{ind} = \bigwedge_p p \in \mathbb{P}^{+}_{all}$, where $\mathbb{P}^{+}_{all}$ is the set of all predicates needed to derive the invariant. The predicate discovery procedure invokes the `mine-predicates` sub-procedure that uses information within a negative or an implication example, ex, and either (i) outputs a set of Impl-type predicates that is consistent with the positive examples and eliminates ex, or (ii) fails if no such predicate exists. Notice that predicate discovery only explores $\mathbf{H}$ by adding predicates, and therefore cannot find an $\mathbf{H}$ that requires a certain p to be dropped. To overcome this, on failure, we first run Sorcar (which *can* drop predicates) with the set of predicates discovered so far to check if $\mathbf{H}$ exists within this set. The soundness of this step follows from Sorcar. We discuss the other causes and remediations for failure in Section 2.6.5 and present the details of predicate discovery next in Section 2.6.3.

Step 3. Lastly, we have an optional step to minimize the number of predicates in the inductive invariant $\mathbf{H}_{ind}$. The invariant minimization step takes as input the discovered set of predicates that are consistent with the positive examples $\mathbb{P}^{+}_{all}$ and constructs an invariant smaller than $\mathbf{H}_{ind}$. The procedure uses `select-relevant-predicates` that picks a subset of predicates from $\mathbb{P}^{+}_{all}$ based on the negative and implication examples seen so far and outputs a candidate invariant $\mathbf{H}_{cand}$. The candidate invariant is checked for safety and inductivity. Failing produces a new negative (safety) or implication (inductivity) counterexample, respectively. If the check passes then the candidate invariant $\mathbf{H}_{cand}$ is our new minimized invariant $\mathbf{H}_{ind-min}$. We formulate the invariant minimization problem in Section 2.6.4.

2.6.3 Predicate Discovery

In practice, it may not be possible to enumerate all predicates. Consider the predicates of type Impl where the first argument is any state element and the second argument is a condition over state element(s) in the design. As the conditional expression can be arbitrarily complex, listing out all predicates in $\mathbb{P}_{Impl}$ is intractable.

Starting from the set of all predicates containing only Eq-type predicates that hold on positive examples, i.e., $\widehat{\mathbb{P}^{+}_{all}} = \mathbb{P}^{+}_{Eq}$, the predicate discovery algorithm adds a small, but useful subset of Impl predicates to $\widehat{\mathbb{P}^{+}_{all}}$, forming $\mathbb{P}^{+}_{all}$, such that the resulting set is sufficiently expressive to learn an invariant for $\mathcal{D}$.

Predicate discovery is based on the following two key observations:

First, that the unsafety in a design is due to unsafe instructions interacting with unconstrained (secret) data when that data is stored in specific state elements. In other words: we can represent those unsafe states by formulating Impl constraints that forbid secrets from being present in specific state elements when *opcode bits* (or functions of opcode bits) of unsafe instructions are present in potentially other state elements.

Second, that both the state elements containing secret data and those encoding instruction opcode information have *signatures* that make it possible to identify secret- and opcode-holding state element candidates in our negative and implication examples. Specifically: state elements that do not satisfy Eq constraints potentially contain secret data; whereas state elements that satisfy Eq constraints potentially contain opcode-related information.

With the above in mind, we proceed as follows. In each example, we partition the state elements into two sets:

i. V_s : the set of state elements that, for the current example, hold different values on the L and R executions.

ii. V_p : the set of state elements that, for the current example, hold the same values on the L and R executions.

We build a set of Impl predicates, i.e., a subset of $\mathbb{P}_{Impl}$, by taking the cartesian product of (i) and (ii). That is, we allow (i) only when the assignments in (ii) do not equal specific values that are equal in both the L and R executions. Our thesis is that if a given state element holds the same value in both the L and R executions, it is an opcode-derived constant. Thus, this construction captures potential interactions between secret data and opcode-related data.

More detailed pseudocode for predicate discovery is given in algorithm 2. The top-level algorithm starts from the universe of predicates $\widehat{\mathbb{P}^{+}_{all}} = \mathbb{P}^{+}_{Eq}$ consistent with positive examples. In every round, new predicates consistent with positive examples are added until the predicates in what becomes $\mathbb{P}^{+}_{all}$ are sufficient to prove inductive safety. Each round considers the largest conjunctive invariant $\mathbf{H}_{cand} = \wedge\mathbb{P}^{+}_{all}$ (line 3). On a counterexample, *cex*, the procedure calls `mine-predicates` to find one or more predicates to add to $\mathbb{P}^{+}_{all}$ that can eliminate the *cex* (line 4). To generate this set of predicates, `mine-predicates` tracks variables of types (i) (line 15) and (ii) (line 17) as described above. Next, the potential set of predicates, $\mathbb{P}^{'}_{Impl}$, is constructed by taking a cartesian product of the above two cases (line 20). Lastly, we retain only predicates from $\mathbb{P}^{'}_{Impl}$ that hold on positive examples (line 22) to form $\mathbb{P}^{+}_{Impl}$: the set of predicates that hold on positive examples *and* are useful in eliminating *cex*. The final set of useful Impl predicates is the union over useful predicates discovered on each *cex*, i.e., $\bigcup_{cex} \mathbb{P}^{+}_{Impl}$. Note that the invariant formed by the conjunction over the discovered set of predicates through this procedure is maximal (i.e., contains both the initial and discovered predicates), safe, and inductive.

2.6.4 Invariant Minimization

We now formulate the problem of minimizing the invariant and develop several approaches for doing so. A smaller invariant is desirable for three reasons. First, a smaller invariant implies a larger state space allowed by the invariant. This means that the invariant allows a larger number of states, i.e., including states in which even the execution of an unsafe

Algorithm 2: Predicate Discovery.

Input : $\widehat{\mathbb{P}^{+}_{all}}$: Initial set of predicates, i.e., $\mathbb{P}^{+}_{Eq}$.
Output: $\mathbb{P}^{+}_{all}$: The augmented set of all predicates sufficient to derive an invariant for $\mathcal{D}$.

```
1  P+_all = P̂+_all;
2  while true do
3  |  H_cand = ∧_i p_i ∈ P+_all;
4  |  cex = check-ind-safe?(H_cand);
5  |  if cex then
6  |  |  P+_all = P+_all ∪ mine-predicates(cex);
7  |  else
8  |  |  return P+_all;
9  |  end
10 end
11 def mine-predicates (cex) → P+_Impl:
12 |  V_s = V_p = ∅;
   |  // ∀ state elements in cex
13 |  for s ∈ cex do
14 |  |  if cex[s_L] = cex[s_R] then
   |  |  |  // (ii) s: non-secret constant
15 |  |  |  V_p = V_p ∪ (s, cex[s_L]);
16 |  |  else
   |  |  |  // (i) s: secret
17 |  |  |  V_s = V_s ∪ s;
18 |  |  end
19 |  end
20 |  P'_Impl = {Impl(v, s ≠ c) : (v, (s, c)) ∈ (V_s × V_p)};
21 |  P+_Impl = refine-positive-example(P'_Impl);
22 |  return P+_Impl;
23 end
```

instruction may also be safe. Second, it helps experts analyze and understand the root cause of unsafety in their designs, as we show in Section 2.8.4. This is harder to do if the invariant is large (contains a large # of predicates) with many irrelevant predicates. Lastly, a smaller invariant results in faster checks during verification.

Recall that the minimization procedure starts from the set of predicates $\mathbb{P}^{+}_{all}$ that is output by the predicate discovery algorithm (Section 2.6.3). As predicate discovery only stops once it has found an inductive invariant, we know that invariant $\mathbf{H}$ exists in the set of predicates $\mathbb{P}^{+}_{all}$. Therefore, the following minimization strategy cannot fail to produce an invariant. At worst it will produce an invariant no smaller than the existing $\mathbf{H}_{ind}$ from predicate discovery.

Definition 2.6.1. Hitting-set Formulation: *Observe that (ideally) we only need one* $p \in$

$\mathbb{P}^+_{all}$ *to be consistent with a negative/implication example,* ex_i*. We formulate the problem of picking* $\mathbb{P}_{cand} \subseteq \mathbb{P}^+_{all}$ *consistent with each example* ex_i *as a minimum-hitting set problem [121]: Record for every* ex_i *the set of predicates* $\mathbb{P}_i \subseteq \mathbb{P}^+_{all}$ *that eliminates* ex_i*. Find* $\min |\mathbb{P}_{cand}|$ *s.t.* $\mathbb{P}_{cand}$ *"hits" every* $\mathbb{P}_i$*, i.e.,* $\forall_i(\mathbb{P}_{cand} \bigcap \mathbb{P}_i) \neq \emptyset$*.*

Minimization using the *greedy* algorithm. We use a well known greedy approximation [46] to find a solution to the minimum hitting set formulation from above. In short, the procedure selects $p \in \mathbb{P}^+_{all}$ greedily such that in each step the selected p eliminates the largest number of examples that are not yet eliminated until there are no more examples to eliminate. This returns an $\mathbf{H}_{cand}$ in polynomial time. Although the above described greedy algorithm runs in polynomial time, it may take an exponential number of examples to find the invariant. Therefore, we also implement a softer version, as described in Sorcar, where we force a new predicate to be added to the invariant on every example. We call the former greedy-ConjunCT and the latter greedy-frozen. We use both minimization approaches: first we try greedy-ConjunCT and fallback to greedy-frozen when the minimization does not find an invariant in a reasonable amount of time. Compared to predicate discovery, the invariant synthesized by the greedy scheme can be much smaller. For example, on Rocketchip the invariant synthesized by greedy-frozen only contained 851 predicates vs. the original invariant from predicate discovery which contained 3,232 predicates.

2.6.5 Failure and Recovery

The above described learning algorithm may terminate and fail to produce an inductive safety invariant for one of several reasons:

Poisoned positive examples. It is possible that one of the examples assumed to positive is actually a state that will eventually result in a safety violation when run for some steps $K' > K$. Therefore, by considering an unsafe intermediate state as safe, we may have inadvertently pruned out predicates from $\mathbb{P}^+_{Eq}$ that are essential in synthesizing a safe and inductive $\mathbf{H}$. To recover from this failure, one can imagine a human-in-the-loop who can analyze the failure, attribute it to a certain unsafe instruction being misclassified as a safe instruction, and re-run synthesis by moving the corresponding positive examples to the negative examples. Another simpler, less involved solution is to re-run the bounded analysis with a bound $K' > K$ to trigger the unsafe behavior in the bounded analysis phase and then synthesize the invariant using the cleaned-up set of positive and negative examples.

DSL is not expressive enough. In general, we cannot guarantee that our DSL is complete and sufficient to express invariants for designs that we have not evaluated on. Fundamentally, there is a trade-off between the expressiveness of the hypothesis space and the tractability of synthesis. That said, Impl was inspired by and captures common design patterns seen in designs today. For example, how each state element is associated with a specific in-flight instruction in a given cycle. Thus, we believe it will be useful in verifying larger designs and show in Section 2.8 that it is sufficiently powerful to express invariants for the open-source designs that we have evaluated ConjunCT on so far. We note that

our analysis may also fail if the root cause of the unsafety is due to reasons other than executing unsafe instructions, e.g., if the optimization acts on an instruction's execution in an operand-independent way.

2.7 Proof Sketches

In this section we will provide proof sketches for ConjunCT. First, we will refine Def. 2.4.6 to define sets of safe instructions useful in practice, e.g., for constant-time programs. Next, we will show that the sets of safe instructions produced by ConjunCT satisfy the constant-time safe sets definition (in Section 2.7.1). Finally, we show that the invariant $\mathbf{H}_{ind}$ synthesized by ConjunCT is precise (in Section 2.7.2).

2.7.1 ConjunCT Proof of Soundness

First, we will instantiate Def. 2.4.6 for the constant-time programming setting. Recall, Def. 2.4.6 is parameterized by: (a) R: the set of secret sources, and (b) O: the set of attacker observable variables. We refer to this definition as $\mathsf{SISP}(R, O)$. The choices of (a) and (b) influence what set of instructions are safe and, as such, are context-dependent. Not all combinations of (a) and (b) yield meaningful results.

In this work, we're interested in the set of safe instructions which can be composed together to form constant-time programs. This means R should be architectural sources of operand data—i.e., the architectural register file (ARF) and/or data memory. W.l.o.g., as we consider RISC-like ISAs where all data memory is written to the ARF before being used, we set R to be equal to the ARF. With that in mind, we can define the constant-time safe instruction set problem:

Definition 2.7.1. Constant-Time Safe Instruction Set Problem: The constant-time safe instruction set problem ($\mathsf{CT\text{-}SISP}(O)$) is an instance of SISP where the set of secret sources, R, is set of be the architectural register file. That is, $\mathsf{CT\text{-}SISP}(O) \coloneqq \mathsf{SISP}(R = \mathsf{ARF}, O)$.[6]

It should be clear that by the definition of SISP that $\mathsf{CT\text{-}SISP}(O)$ – for an appropriate choice of O – defines Σ^+ for $\mathcal{D}$ which are safe for use in constant-time programs. That is, the first access of a given secret must be from the ARF and the definition considers all compositions of instructions and starting states from the point that the secret is initially accessed. Let us call the final safe set output by ConjunCT $\widehat{\Sigma^+}$. Section 2.11 provides a proof that $\widehat{\Sigma^+}$ is valid in $\mathsf{CT\text{-}SISP}(O)$ for the specified O.

[6]Note, we continue to leave O to be parametric to be able to model different attacker capabilities, although it will generally be set to signals that correspond to an instruction's retirement time.

2.7.2 Proof-Sketch that $\mathbf{H}_{ind}$ is Precise

In this section, we show that the invariant synthesized by ConjunCT is precise. We note that invariants formed by Sorcar/Houdini are naturally precise, but precision isn't discussed in those works. Below, we give an argument for why they are precise, and reconcile differences between their analysis and ours to show that the precision arguments governing their analysis applies to ours as well.

As we're interested in a relational invariant for C (Def. 2.4.2), we define a product machine that operates over a pair of states.

Definition 2.7.2. Product machine C_p: Construct a product machine for C, named $C_p = (\mathbb{S}_p, (s_{init}, s_{init}), R, \Sigma, O, \mapsto)$, where all symbols have their usual notations, but redefined for the product setting: $\mathbb{S}_p$ is the set formed by taking a cartesian self-product $\mathbb{S} \times \mathbb{S}$, $\mapsto^a : \mathbb{S} \times \mathbb{S} \to \mathbb{S} \times \mathbb{S}$ maps the pair of states $(s_0, s_0') \mapsto^a (s_1, s_1')$ if $(s_0 \rightsquigarrow^a s_1)$ and $(s_0' \rightsquigarrow^a s_1')$.

Let $x = a_0, a_1, \ldots, a_n$ denote a sequence of inputs over Σ^+, the set of safe instructions.

Definition 2.7.3. Precision: We say $\mathbf{H}$ is precise if for each state $s_i^p = (s_i^{\mathsf{L}}, s_i^{\mathsf{R}})$ appearing in any trace generated by a x on C_p starting from states $(s^{\mathsf{L}}, s^{\mathsf{R}})$ where $s^{\mathsf{L}} \approx^{sec} s^{\mathsf{R}}$, $\mathbf{H}$ allows s_i^p, i.e., $s_i^p \models \mathbf{H}$.

We define C_p to have a non-deterministic ϵ transition from the initial state (s_{init}, s_{init}) to all states $(s^{\mathsf{L}}, s^{\mathsf{R}})$ where $s^{\mathsf{L}} \approx^{sec} s^{\mathsf{R}}$. If an inductive invariant $\mathbf{H}$ for this C_p exists within our predicate language, Sorcar will find said $\mathbf{H}$. This $\mathbf{H}$ is precise: the state (s_{init}, s_{init}) is safe and allowed by $\mathbf{H}$, and furthermore, by the definition of an inductive invariant, $\mathbf{H}$ should allow all safe states reachable through successive applications of $\mapsto^a \quad \forall a \in \Sigma^+$, as otherwise it would violate the inductive property of $\mathbf{H}$.

ConjunCT constructs a precise invariant. Recall that we start the analysis described in Section 2.5 from a symbolic start state, $\mathbf{S}_0$, that captures all states $(s^{\mathsf{L}}, s^{\mathsf{R}})$ where $s^{\mathsf{L}} \approx^{sec} s^{\mathsf{R}}$. Therefore, every positive example collected from the bounded analysis has the state $\mathbf{S}_0$ as the prefix. Allowing $\mathbf{S}_0$ into $\mathbf{H}$ is equivalent to allowing all pairs of states $(s^{\mathsf{L}}, s^{\mathsf{R}})$ where $s^{\mathsf{L}} \approx^{sec} s^{\mathsf{R}}$ into $\mathbf{H}$. As we allow all valid start states into $\mathbf{H}$, it follows from the above argument that the inductive invariant synthesized by ConjunCT is also precise. Hence, as long as the set of positive examples is complete, i.e., covers all safe instructions $a_s \in \Sigma^+$, the synthesized invariant is precise.

Caveat / Source of Imprecision. Since the bounded analysis in Section 2.5 may be imprecise when attributing blame to an instruction (Section 2.5.1.1) the resulting subset of safe instructions, $\widehat{\Sigma^+}$, may be non-maximal, i.e., $\widehat{\Sigma^+} \subseteq \Sigma^+$. As a result, the derived invariant may also be imprecise in the same way, e.g., if $\widehat{\Sigma^+}$ excluded a safe instruction $a_s \in \Sigma^+$ due to the above mentioned imprecision then $\mathbf{H}$ will not allow any states in the execution of a_s. Or in other words, the states in the execution of a_s are false positives. We leave better attribution of blame to future work.

Lastly, we note that in our evaluations, we never encounter the case where the composition of potentially safe instructions results in unsafe behavior. Therefore, the analysis does not

suffer from a loss of precision described in Section 2.5.1.1, and so $\widehat{\Sigma^+} = \Sigma^+$, and the derived invariant is indeed precise.

2.8 Evaluation

We now evaluate an implementation of ConjunCT on several RISC-V microarchitectures, reporting on analysis time, annotation effort and statistics related to the constructed invariants / per-design safe instruction sets. The end of the section provides two case studies. First, we show how the minimized invariant produced by ConjunCT can be used to localize where in a design an unsafe optimization is implemented. Second, we show how ConjunCT can be used to co-design safe but performant microarchitecture.

2.8.1 Implementation and Methodology

Framework. We implement ConjunCT in Python and Racket. ConjunCT is currently implemented in about $\sim$ 6000 lines of Python and is responsible for parsing the design in Verilog, converting Verilog to our internal DSL (to symbolically evaluate in Racket), performing optimizations, computing the product program, generating test harnesses, and instrumenting the code. Additionally, we implement all of our symbolic analysis and invariant learning in Rosette [202], a DSL to build solver-aided tools in Racket, in about $\sim$ 6000 lines of Racket. ConjunCT uses specifications from the official RISC-V repository [173] for instruction encodings. Annotations are described in a separate file.

Experimental Setup. We ran all our evaluations on a standard desktop machine equipped with 16GB of memory and an Intel i5-9500 with 6 cores running Ubuntu 18.04. We use CVC5 [20] as the SMT solver in all our experiments. We obtained the open-source designs from their official repositories and processed them through yosys [223], e.g., flattening modules, performing basic optimizations etc., before pairing them with ConjunCT.

Evaluated pipelines. We evaluate ConjunCT on three pipelines (summarized in Table 2.2) of varying complexity. We do not currently analyze the data cache, since non-memory instructions do not interact with the data cache, but this choice was not fundamental.

All three pipelines (*V-Scale* [204], *Ibex* [141], *RocketChip* [18]) are single-issue in-order cores, with 2, 3 and 5 pipeline stages, respectively. For Ibex, we evaluate the *small* configuration; for RocketChip the `DefaultRV32Config` configuration. We note, the small Ibex configuration is the default and only formally-verified configuration. We modified all three cores to only use uncompressed instructions. V-Scale and Ibex use the RV32IMC ISA subset; RocketChip uses the RV32 ISA subset.

	# Pipeline Stages	# Regs	# Wires	# Annotations O	S	R	P	A
VScale	3	1,080	11,186	1	1	1	0	0
Ibex	2	2,013	40,306	1	2	2	1	1
RocketChip	5	2,091	54,461	1	2	1	4	0

Table 2.2: Complexity of designs under analysis. The right half of the table shows the number of annotations (by type) needed to begin the ConjunCT analysis: (O) Setting the observation variable, (S) Setting the instruction source, Marking (R) the register file and (P) program counter register; finally: (A) any additional annotations. # Regs denotes the number of Verilog reg bits assigned inside clocked 'always' blocks. We note, this may *under count* the true number of flip-flops/registers, as some registers can manifest due to code outside of clocked blocks.

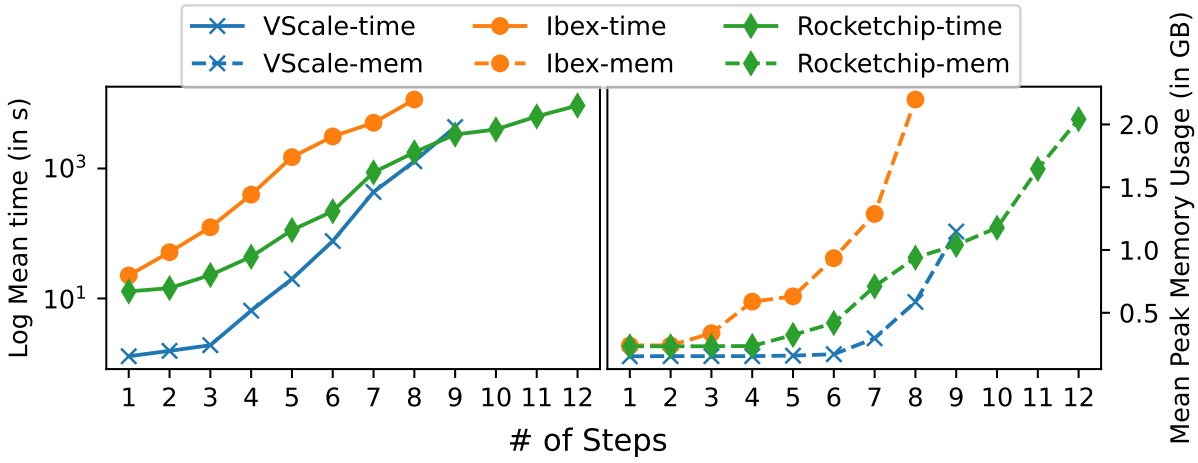


Figure 2.4: Performance of Bounded Analysis (Section 2.5). Steps correspond to clock cycles.

	Predicate Discovery Eq	Impl	Total	Time	Greedy Eq	Impl	Total	Time
VScale	161	995	1,156	10m	60	8	68	+25m
Ibex	812	0	812	4m	84	0	84	+44h
RocketChip	326	2,906	3,232	7h	235	616	851	+3h

Table 2.3: Evaluation of Invariant Learning. Eq, Impl denote the number of each type of predicate needed to construct the invariant (of size Eq + Impl). Time taken by greedy is in addition to the time taken for predicate discovery.

2.8.2 Quantitative Evaluation

Performance of ConjunCT. We now evaluate ConjunCT in terms of analysis time and memory usage, as a function of the design complexity. Figure 2.4 shows the scalability of our bounded analysis for each design. We run the bounded analysis on each of the three designs for an increasing number of cycles. Figure 2.4 (left) plots the log of mean execution time vs. a bound on analysis steps (cycles). The execution time of the analysis scales exponentially with the number of steps, with more complex designs starting off with a higher time. All executions were set to have a maximum timeout of four hours.

The number of bounded analysis steps needed to derive the invariant for the designs we evaluated was 5, 3, and 6 steps, respectively. This took on the order of minutes to run for all designs. For illustrative purposes, we show how runtime scales beyond this: The symbolic execution was able to explore up to steps 9, 8, and 12 for VScale, Ibex and RocketChip, respectively, before the timeout was reached.

It is interesting to note that by step 9, the analysis time of the simpler VScale design actually exceeds that of the more complex RocketChip. At each step of the bounded analysis, symbolic expressions are generated from values and expressions stored in the state elements from the previous step. Hence, as the number of steps of the analysis grows, these expressions also grow in complexity making them more difficult to solve. The rate of growth of this expression complexity is not just a function of the number of state elements in the design, but also of how the state elements are connected and used.

Figure 2.4 (right) shows peak memory as a function of analysis depth. The memory used by the analysis scales linearly with the number of steps on all three designs, with the more complex designs consuming more memory. In all cases, the memory usage was moderate and within what's typically available on today's desktop machines ($<$ 3GB).

Annotation Effort and Complexity. All three designs required minimal annotations ($<$ 9) for the full ConjunCT flow (both the bounded analysis and invariant learning phases). We show this annotation effort in the right half of Table 2.2. All designs require us to annotate the observation variable (O), instruction source (S), and the register file (R). In addition, Ibex and RocketChip required us to annotate the state elements holding the program counter (P) to ensure that the PC is aligned. Lastly, Ibex required us to add one more annotation to eliminate an invalid start state in the load-store unit (LSU) that led to safe instructions in the design being flagged as unsafe, but through invalid counterexamples.

For the most part, annotation effort is straightforward. The annotations for (O), (S), (R), and (P), involve identifying registers corresponding to the key structures found in all hardware designs. The only annotation that needed significant effort was the 1 (A) annotation for Ibex. The (A) annotation required debugging and understanding the false-positive counterexamples so as to identify the root cause that led to the unreachable initial state. After identifying the root cause, we had to carefully constrain the initial state to eliminate the unreachable states without removing any reachable states. This entire process took less

	and	andi	xor	xori	or	ori	sll	sra	srl	add	addi	sub	mul	mulh	mulhu	mulhu	mulhsu	div	divu	rem	remu	ecall	ebreak
VScale	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ibex	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓
RocketChip	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	lb	lh	lw	lhu	sb	sh	sw	lbu	lui	slt	sltu	slti	jal	jalr	beq	bge	bgeu	bge	bltu	blt	bne	fence	auipc
VScale	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
Ibex	✓	✗	✗	✗	✓	✗	✗	✗	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
RocketChip	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓

Table 2.4: Table showing the set of safe and unsafe instructions on three open-source designs: VScale, Ibex, and RocketChip. ✓ represents that an instruction is safe on the microarchitecture while a ✗ denotes that an instruction is unsafe.

than 1 day for a graduate student.

Learnt Invariant Statistics. ConjunCT was able to synthesize an invariant for all three designs, the statistics for which are shown in Table 2.3. For all three designs, we invoked predicate discovery (Section 2.6.3) to generate a set of Impl predicates necessary to learn an invariant. We show results for both the invariant synthesized by predicate discovery and the Greedy predicate minimization strategies (Section 2.6.4).

For all designs except Ibex, Impl predicates were necessary to synthesize an invariant. Using predicate discovery, the invariants of V-Scale, Ibex, and RocketChip have 1,156, 812, and 3,232, predicates respectively. Using the greedy-ConjunCT minimization strategy (Section 2.6.4), we were able to significantly reduce the number of predicates per invariant for V-Scale and Ibex. For RocketChip we failed to derive an invariant using the greedy-ConjunCT strategy even after 2 weeks of running. Hence, we fallback to a softer version of the greedy minimization strategy (described in SORCAR), greedy-frozen, and force a new predicate to be added to the invariant in every iteration of learning. This converges relatively quickly (in a polynomial number of examples) and generates an invariant with 851 predicates.

2.8.3 Security Properties / Security Evaluation

Next, we analyze the set of safe and unsafe instructions as identified by ConjunCT. Table 2.4 shows which instructions are safe in each design. We manually analyzed each design to understand root causes and validate that the instructions identified as safe are actually safe. We also validated our result on Ibex against a related work UPEC-DIT (Section 2.9), and found the two to be in agreement (with one exception—see below).

On V-Scale, all instructions that we tested, except for branches, are safe. Branches on V-Scale stall for a cycle if the branch is taken vs. if the branch is not taken as the next fetched instruction needs to be flushed and fetched again. As the conditional to the branch is a secret, an attacker who can observe the retirement time for the branch can learn if the secret predicate evaluated to true or false.[7]

[7]We note that aside from the above timing disturbances, branches are generally considered unsafe because

$$rs^{\mathsf{L}} \neq rs^{\mathsf{R}} \implies mem\_ctrl\_branch \neq 1$$

Figure 2.5: Example of a predicate in the RocketChip invariant. Register names have been changed for readability. The LHS specifies a register that can conditionally hold a secret and the RHS describes the condition. `rs` is the input source register to the execute stage of the pipeline. In this example, `rs` can hold a secret if the control signal `mem_ctrl_branch` is not set. This is intuitive as branches are unsafe and acted on in the execute stage.

Similarly, branches are unsafe on Ibex for the same reason. Additionally, branches are unsafe also due to misaligned targets: when the branch target is misaligned, Ibex needs to perform two fetches from memory instead of one, thereby taking an additional cycle. Most loads and stores, with the exception of `lb` and `sb`, are unsafe for the same reason: performing an unaligned load or store causes the processor to make two aligned requests instead of one, thereby influencing the load/store's retire time. As `lb` and `sb` deal with a single byte there is no misalignment. Therefore these variants of the instructions are safe.[8] Recall, we are not currently modeling cache (Section 2.8.1), so there are no cache-based attacks (Section 2.3) to render `lb`/`sb` unsafe. Lastly, Ibex implements `div`/`divu` and `rem`/`remu` in a non-constant time way as a division by zero completes in 1 cycle, while all other divisions take 37 cycles.

Lastly, on RocketChip all branch and memory instructions are unsafe for the same reason as on the other cores. The multiply and divide instructions turn out to be safe because in the default configuration they are unrolled for a minimum of 8 cycles before optimization (and since no `mul`/`div` takes > 8 cycles, these instructions for this parameterization of RocketChip are safe).

2.8.4 Case-Study: Analysis of RocketChip Invariant to Perform Root Cause Analysis

The derived invariants contain a treasure trove of information regarding the root cause of unsafety in the design. However, understanding the invariant is non-trivial as it contains a large number of predicates, e.g., the invariant for RocketChip has more than 850 predicates. This creates a needle in a haystack problem as most predicates do not point to root causes of unsafety but, rather, are required to block states that will eventually lead to unsafety. We leave a more systematic exploration of the invariant to future work, but report here encouraging best-effort results that show how the invariant can be helpful in localizing where in the microarchitecture an unsafe optimization is implemented.

they change the the number of dynamic instructions in the program's execution. Our analysis does not consider this fact, but can be easily changed to by adding the PC register to the set of observation variables.

[8] We note that UPEC-DIT does not break instructions down by data width, and thus finds that loads/-stores of all widths are unsafe.

We analyze the RocketChip invariant. To derive useful information, we focus our attention to the Impl-type predicates exclusively as they provide instruction-specific causal information in the form of: "state element X cannot hold secret values *when* state element Y is associated with a specific unsafe instruction." An example of such an Impl-type predicate found in the RocketChip invariant is shown in Figure 2.5. We started by grouping predicates based on the registers that appear on the LHS of the Impl predicates. By the semantics of Impl, these registers can conditionally hold secret data (Section 2.6.1). We found that across all (616) Impl-type predicates, there were only 8 distinct registers that appeared on the LHS:[9]

- (A, B, C) 3 registers that make up the `rs` (register source) in the Execute stage.
- (D) 1 register that stores data to be written back to memory.
- (E) 1 register that stores data to be written back to the register file.
- (F, G, H) 3 registers related to the divide unit: remainder, quotient, and a state register storing the current operation's running cycle count.

We now discuss when Impl predicates indicate that these registers are allowed to hold secret data.

(A, B, C) cannot hold secret data when an unsafe instruction (Table 2.4) is executing. More specifically, (A) cannot hold secret data when any of the control signals corresponding to an unsafe instruction-type is set (`mem_ctrl_{branch, div, fp, jalr, mem}`), and (B, C) cannot hold secret data when the executing instruction is a load or a store. This information is intuitive and useful: unsafe optimizations in RocketChip are implemented in the execute stage and the `rs` register is an input to the execute stage.

(D) cannot hold a secret when a CSR/system instruction-type is set (`mem_ctrl_csr`) or when the executing instruction is a load or a store.

(E) is not allowed to hold secret data when control signals related to CSR (control status register) writes are set. This was surprising to us as we did not consider system instructions related to the manipulation of the CSR in our analysis. That said, most system instructions that manipulate the CSR *are* unsafe. To derive an invariant, we need to eliminate all sources of unsafety, including the flow of secrets into the CSRs or any other instruction not included in the analysis. This highlights the power of predicate discovery (Section 2.6.3) in automatically finding Impl that are crucial to blocking out unsafety without the need for expert analysis or annotations.

Finally, why (F, G, H) cannot (conditionally) hold secret data requires more explanation. Recall, our earlier results reported that the division instructions in RocketChip are safe (Table 2.4). In that case, why would there be Impl constraints on (F, G, H), which are registers that are part of the division unit? We found that this is because the divide unit in RocketChip is not *truly* constant-time, but rather just constant time for the ISA subset

[9]The names of registers have been simplified/annotated to ease presentation.

(RV32) that we evaluate. ConjunCT synthesizes Impl constraints to properly initialize the divide unit and reflect that for this ISA subset, it is safe.

In more detail: By default, RocketChip's divide unit always unrolls divisions for 8 cycles. Therefore, any division that takes less than or equal to 8 cycles will *always* take 8 cycles. As we are evaluating RocketChip parameterized to operate on RV32, division on any two 32-bit values can be completed within 8 cycles, and hence are constant-time and safe. The invariant needs to capture this fact. Because the three values, count, divisor, and remainder, are unconstrained and can take any value during invariant learning, including unreachable values that imply a > 8 cycle operation, additional predicates are required on these variables to disallow variable-timing behavior of the divide unit. To prevent the secret values from affecting the retire register, the invariant disallows the above registers from holding secrets when the state of the division unit is either `s_div` (start of division) or `s_div_ready` (end of division).

In addition to all of the above predicates, there are various other predicates that prohibit exceptions under certain conditions.

2.8.5 Case-Study: Computation Reuse

Lastly, we show how ConjunCT can help microarchitects evaluate the security impacts of their proposed optimizations (echoing Section 2.3). For this case study, we implement computation reuse, an advanced microarchitectural optimization that *memoizes* the result of an expensive instruction in case it is called with the same operands twice [192]. This is an interesting optimization to study as it is typically implemented as a part of the pipeline's instruction decode (ID) logic [209, 192]. Hence, it illustrates the need to audit the entire pipeline rather than just the execution units.

Sodani et al. [192] describes two different schemes to implement computation reuse. Scheme (i) looks up the memoization (reuse) table by instruction opcode and operand *value*. Scheme (ii) looks up the reuse table by instruction opcode and operand *register id*. Both schemes update the table (add/update a table entry) when an instruction executes that is a candidate for memoization. Since it's possible for Scheme (ii) to have false hits, it needs to be flushed when an instruction writes to a register whose id is present in the table.

Interestingly, Sodani et al. [192] find that both of the above schemes improve performance. Even more interestingly, as noted by Vicarte et al. [209], they (intuitively) have different security implications: Scheme (i) can create new unsafe instructions because it skips instruction execution in an operand value-dependent way. Scheme (ii), on the other hand, cannot: it skips execution only as a function of *register id* (which is usually considered non-secret, e.g., in the constant-time programming setting).

We implemented the above two reuse schemes in Ibex [141], which consists of instruction fetch (IF) and instruction decode/execute (ID/EX) stages. On this pipeline, most instructions complete the ID/EX stage in 1 cycle. Several others require multiple cycles: (a) `mul` / `mulh` take 3/4 cycles to complete, and (b) `div` / `rem` take either 1 cycle (when there is a divide by 0) or 37 cycles to complete.

We implement both reuse schemes as a part of the ID/EX stage (conceptually as a part of the ID stage and before the EX stage), optimizing the above multi-cycle instructions. In both schemes, reuse table hits immediately return and forward the result (in a single cycle). The instruction executes normally, otherwise. For our proof-of-concept implementation, we set the reuse table to contain only a single entry, and hence, both schemes only memoize the latest `mul / div / rem` computation. We tested our implementations for both correctness (did not produce incorrect results) and "performance-functionality" (i.e., the optimization *saves* cycles as expected).

We evaluated ConjunCT on both schemes. For Scheme (i), ConjunCT correctly identifies during the bounded analysis (Section 2.5) that the `mul`-family of instructions (which were safe without the optimization added) are now unsafe. It also identifies that `div/rem` instructions are unsafe, although these instructions were already unsafe on Ibex. Finally, ConjunCT correctly identifies that Scheme (i) doesn't render any other safe instructions unsafe. For Scheme (ii), `mul` instructions continue to remain safe, `div/rem` remain unsafe, and other instructions are not affected. This also matches expectations. To complete the evaluation, we ran invariant learning (Section 2.6.2) to derive an invariant that proves unbounded safety/unsafety of all instructions on both designs. This showcases ConjunCT in action: ConjunCT is capable of advising microarchitects on when novel performance optimizations create novel security issues in a design.

2.9 Related Work

We compare to related works on three axes. **(R1)**: whether the analysis is *sound* wrt. the safety property, i.e., it does not miss any safety violations (have false negatives). **(R2)**: whether the analysis is *precise*, i.e., does not flag states/instructions that are safe to execute as unsafe (have false positives). Finally, **(R3)**: the proposal's degree of automation, i.e., whether it requires heavyweight annotations, or require a human-in-the-loop. ConjunCT achieves all three goals. Invariant learning enables **R1** (Section 2.6) and **R2** (Section 2.7.2). Predicate discovery (Section 2.6.3) and our approach using synthesis (Section 2.5-Section 2.6) empirically enables analysis with very few expert guidance/annotations (Section 2.8). However, we do acknowledge that ConjunCT may require more annotations to achieve precision for larger, more complex designs.

The closest work to ours is UPEC-DIT [58], which is a nascent proposal for identifying which instructions are safe/unsafe in a microarchitecture. UPEC-DIT can be viewed as a simplified version of our bounded analysis Section 2.5: it does not satisfy **(R1)**, as their analysis is bounded and hence not sound (although this restriction is lifted in a follow-up work that is concurrent to ours [57]). It also does not satisfy **(R3)**, as it requires a human-in-the-loop to analyze counterexamples and add annotations during each iteration of the analysis. Similarly, prior work such as Iodine [93] (and its follow-on Xenon [94]) do not satisfy **(R2)**: it is capable of discerning whether an entire design is "constant time", but not with respect to different instructions (and would therefore conclude that every instruction is

unsafe in our setting). Iodine also does not satisfy **(R3)** as it requires a human-in-the-loop to identify secrecy assumptions.

Finally, concurrent work by Wang et al. addresses the problem of verifying leakage contracts [219] using invariant synthesis. This work is closely related to ConjunCT. For example, the contracts I, B, M, O (on Ibex) corresponds to ConjunCT finding branch, memory, and mul/div/rem instructions unsafe. That said, their work is solving a different (while adjacent) problem to ours. Their work requires designers to write microarchitecture-specific leakage contracts in Verilog. By contrast, ConjunCT starts with no apriori knowledge of any contracts, i.e., what instructions might leak, and instead tries to deduce this information. In their terminology: our Phase 1 (Section 2.5) can be viewed as inferring *likely contracts*, i.e., the set of safe/unsafe instructions. Our Phase 2 (Section 2.6.2) then proves that the inferred likely contracts are actual contracts. This, when the problem at hand is determining the safe instruction set (Def. 2.4.6), requires significantly lower annotation burden. Lastly, their work does not scale to instructions that need many cycles to complete, e.g., like `div` which requires 37 cycles, while ConjunCT has no such limitations.

There is rich literature in using symbolic execution [181, 97, 143, 31], fuzzing [28, 96, 237, 95], and a combination of these techniques to find bugs in both hardware and software. While these techniques are useful in finding bugs in practice, they cannot perform verification to prove absence of bugs or derive specifications, which is the subject of this work. In theory, ConjunCT can use these advances in bug-finding techniques, with different soundness and scalability trade-offs in the bounded-analysis phase. This is an interesting future direction for research.

2.10 Conclusion

This work presented ConjunCT, a proof-of-concept framework/automated analysis that (given an input microarchitecture and low designer annotation burden) determines which instructions are safe in unbounded composition. The key finding is that with a modest family of predicates it is possible to synthesize inductive relational invariants for today's microarchitectures that are capable of discerning safe from unsafe instructions. We view ConjunCT as a starting point. Longer-term, we see ConjunCT-like analyses being used to synthesize security-centric contracts for other secure programming patterns (such as writing programs with balanced branches and spatially isolating computation).

2.11 Security Proof of ConjunCT

In this section, we will prove that the set of safe instructions, $\widehat{\Sigma^+}$, output by ConjunCT satisfies the Constant-Time Safe Instruction Set definition CT-SISP(O) (Def. 2.7.1).

For the purposes of the proof, consider ConjunCT to be composed of two black boxes, corresponding to Phase 1 (Section 2.5) and Phase 2 (Section 2.6) of the analysis. Phase 1

proposes a set of candidate safe instructions $\widehat{\Sigma^+}$, and Phase 2 learns an inductive invariant to prove that there does not exist a composition of said set of instructions that violates the ConjunCT 2-safety property. With that in mind, we can state the main theorem:

Theorem 1. *The set of safe instructions output as* $\widehat{\Sigma^+}$ *by* ConjunCT *satisfies* CT-SISP(O) *(Def. 2.7.1).*

Proof. To start, suppose $\mathbf{H}_{ind}$ is a final inductive invariant output by Phase 2 given candidate safe set $\widehat{\Sigma^+}$ (generated, perhaps, by Phase 1). By definition of an inductive safety invariant, if we start from a state $\mathbf{S} \models \mathbf{H}_{ind}$, any number of applications of $\rightsquigarrow^a, a \in \widehat{\Sigma^+}$ will only reach states that satisfy $\mathbf{H}_{ind}$ and all such states are safe.

With the above in mind, to prove the theorem, it is sufficient to show that $\mathbf{H}_{ind}$ does not contain any predicates that constrain safe instructions from reading secrets from the state elements corresponding to the architectural register file (the ARF). By definition of $\mathbf{H}_{ind}$, this is equivalent to saying that the ARF emits secret data, i.e., ARF data is not constrained between the L and R sides of the product program construction, which matches the semantics and choice of R in Def. 2.7.1.

To show that state elements in the ARF are left unconstrained in $\mathbf{H}_{ind}$, consider the following. Any invariant needs to satisfy the base case, i.e., allow all positive examples. The set of positive examples read secret data from R, i.e., R may be different between the L and R executions of said positive examples. This means:

- $\mathbf{H}_{ind}$ cannot contain Eq on state elements in R.
- $\mathbf{H}_{ind}$ also cannot contain any Impl predicates that prevent safe instructions from accessing secret data as such a predicate would not be satisfied by a positive example.

This concludes the proof: We showed that $\mathbf{H}_{ind}$ does not constrain the ARF for safe instructions, and the semantics of $\mathbf{H}_{ind}$ are that any evolution from said unconstrained-ARF data does not create a safety violation.

Note that the above argument holds for any choice of O and regardless of the microarchitectural details of the design $\mathcal{D}$. For example, whether $\mathcal{D}$ features bypass paths (Figure 2.1), out-of-order/speculative execution, etc. In all cases, the annotation burden needed to specify R to satisfy Def. 2.7.1 is just to specify the ARF.

Discussion: Non-termination. Now that we know that any safe set output by Phase 2 as $\widehat{\Sigma^+}$ always satisfies our definition, let's look at Phase 1. Phase 1 is integral to ConjunCT: it proposes different sets of candidate safe instructions which are then checked by Phase 2 (Phase 2 cannot identify a set of safe instructions by itself). If the set proposed by Phase 1 can create a composition of instructions which is unsafe, it will be rejected by Phase 2. A bad Phase 1 will cause ConjunCT to go around the Phase 1 $\leftrightarrows$ Phase 2 loop multiple times. This loop may not terminate, e.g., if Phase 1 never proposes a correct safe set.

There is no way to prove that our Phase 1 will produce a safe set. At present, our Phase 1 is a heuristic that works well in practice. For example, our current Phase 1 implementation

only releases secret data architecturally once for the instruction under test, and never again. This design helps the analysis scale, and was useful in identifying individual unsafe instructions. That said, this design will likely not be able to discover unsafeness that manifests due to interactions between multiple sources of secret data (since secrets are emitted only once in Phase 1). Such a case might result in the Phase 1 $\leftrightarrows$ 2 loop not terminating. We emphasize, however, that it will never lead to Phase 2 producing an unsafe invariant. That is, Theorem 1 holds for all Phase 1 implementations.

Chapter 3

H-Houdini:
Scalable Invariant Learning

The analysis developed in ConjunCT *(Chapter 2) does not scale to large Out-of-Order (OoO) designs: it takes 8 hours to verify Rocketchip, and worse, its runtime grows exponentially with design size. Even state-of-the-art industrial verification tools fail to scale when proving SISP on larger designs.*

This chapter introduces a new invariant learning algorithm, H-Houdini*, that enables scalable verification for large designs. We implement* H-Houdini *in a tool called* VeloCT*, a successor to* ConjunCT *for proving SISP. Concretely,* VeloCT *achieves a 2880× speedup over* ConjunCT *and, for the first time, verifies security properties on large designs like BOOM, with runtimes ranging from 6 minutes to 3.3 hours.* H-Houdini *is the* ***key technical advancement*** *that enables the central vision of this dissertation: automatically deriving security specifications for real-world processors and applying them in tools like* SynthCT *(Chapter 4). This chapter addresses Challenges: (C2).*

Formal verification is a critical task in hardware design today. Yet, while there has been significant progress in improving technique automation and efficiency, scaling to large hardware designs remains a significant challenge.

We address this challenge by proposing H-Houdini: a new algorithm for (mostly) push-button inductive invariant learning that scales to large hardware designs. H-Houdini combines the strengths of Machine Learning Inspired Synthesis (MLIS) and SAT-based Incremental Learning. The key advance is a method that replaces the monolithic SMT-style checks made by MLIS with a carefully-constructed hierarchy of smaller, incremental SMT checks that can be parallelized, memoized and reassembled into the original 'monolithic' invariant in a correct-by-construction fashion.

We instantiate H-Houdini as VeloCT, a framework that proves hardware security properties by learning relational invariants. We benchmark VeloCT on the 'safe instruction set synthesis' problem in microarchitectural security. Here, VeloCT automatically (with

no expert annotations) learns an invariant for the RISC-V Rocketchip in under 10s (2880× faster than state of the art). Further, VeloCT is the first work to scale to the RISC-V out-of-order BOOM and can (mostly-automatically) verify all BOOM variants (ranging from Small to Mega) in between 6.95 minutes to 199.1 minutes.

3.1 Introduction

Formal verification is a well-recognized bottleneck in hardware design today. Further, one can expect this bottleneck to worsen into the future as design complexity further increases, design time cycles further decrease [119], and the set of properties to be verified further increases (e.g., due to the rise of hardware security vulnerabilities [125, 137]).

At a high level, automated verification procedures consist of an interplay between two components: (i) an algorithm that proposes invariants or partial invariants to prove a property, and (ii) off-the-shelf verification engines that check the proposed invariants. While the past several decades have seen remarkable progress in this direction (e.g., [82, 157, 146, 29, 66, 116, 98]), it still faces scalability challenges when attempting to automatically learn invariants for and verify large designs.

This chapter considers scalability challenges for a popular verification paradigm called Machine Learning-based Invariant Synthesis (MLIS)[1] [40, 243, 87, 73, 82, 157, 77, 67]. MLIS approaches, such as the famous Houdini [82] algorithm and Sorcar [157], frame invariant learning as an interaction between a learner and a teacher. The learner attempts to learn an invariant using examples provided by the teacher.

MLIS algorithms come with a number of positive attributes. Broadly, they enable a "kitchen sink" approach to verification. That is, designers can freely add predicates that might be useful in finding an invariant, and unneeded predicates cannot cause a failure if an invariant exists in the current predicate abstraction. They also feature several mechanisms that can shrink the invariant search space. First, invariants that are found are restricted to the given predicate abstraction. Properly chosen, a predicate abstraction will be able to describe the invariant yet also imply a small (or at least tractable) universe of possible invariants. Second, learning is bootstrapped through the use of *positive examples* (similar to example-answer pairs in supervised learning), which further constrain the invariant search space to just those invariants consistent with at least the positive examples.

Yet, MLIS still faces significant scalability challenges because it assumes the problem under verification is a black box, i.e., does not leverage the problem's structure. Consequently, each query to the teacher to generate an example is computationally expensive, e.g., an SMT query over the set of all predicates.

By contrast, SAT-based approaches, such as IC3/PDR [29, 66], assume the system is white box and can exploit problem structure to perform much cheaper, 'relative' checks that incrementally eliminate bad states. This results in cheaper SMT queries, but without the aforementioned other benefits of MLIS algorithms.

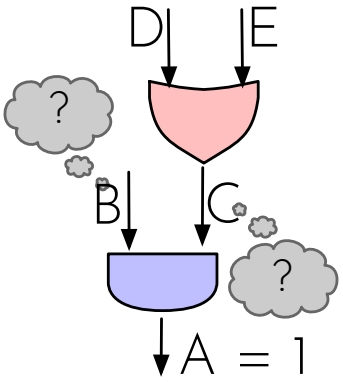
[1] aka. inductive learning

This chapter therefore asks a natural question: *Can we design an algorithm that blends the advantages of the MLIS learners with the ability to perform cheaper incremental checks from white box SAT-based approaches? Would such an approach scale significantly better than existing MLIS approaches?*

This Chapter. We answer the above question in the affirmative by presenting H-Houdini. Relative to SAT-based learners, H-Houdini uses an expert-informed predicate abstraction, a mechanism to guide predicate selection and positive examples to dramatically shrink the invariant search space. Relative to MLIS learners, H-Houdini uses incremental (white box) SMT queries to dramatically decrease the time to search through the (now shrunken) invariant search space.

At the heart of H-Houdini is a way to *soundly* decompose the process of learning an inductive invariant into smaller incremental pieces that compose together at the end to form a complete inductive invariant that proves the target property. Each of the above smaller pieces can be proven inductive using a more efficient relative inductive check. Once all of the pieces are checked, the composition guarantees a correct invariant by construction and the 'monolithic' invariant and property never needs to be checked directly.

For example, consider a two-input AND gate in the context of a larger digital system, whose inputs (B and C) and output (A) are clocked state elements. If the target property is a predicate that requires the output hold a 1, i.e., A = 1, it is sufficient to require that the final invariant include predicates that constrain the inputs to also be 1, i.e., B = 1 and C = 1. To be inductive, this may require that we recursively add predicates to state elements in the cone of influence of each input: the value of C is dependent on D and E, so what should the conditions on D and E be such that C will be 1? And so on. By verifying the inductivity of each level of predicates locally, we prove that the overall invariant (the set of all added predicates) is inductive.



Aside from featuring more efficient checks, this approach features a high degree of parallelism, opportunities for memoization and search-space pruning via positive examples. Returning to the above example, synthesizing what predicates are required during the recursion on each input state element can be done in parallel. If a predicate is inconsistent with a positive example, it need not be considered. Once a predicate is either removed or proven inductive, it need not be re-proven. That is, if two cones of influence overlap, the overlap need only be analyzed once.

Application to secure hardware verification: VeloCT. We instantiate H-Houdini to solve an important problem in hardware security called the *safe instruction set synthesis problem (SISP)* [61, 75, 57, 60, 83]. We call the resulting analysis tool VeloCT.

Verifying the SISP enables portable and secure constant-time programming on modern processors. Constant-time programming is a ubiquitous paradigm for writing code that

is safe from timing attacks [126, 23, 39, 234, 37]. The idea is to write a program using only 'safe' instructions, where a safe instruction's execution time does not depend on its operands. Then, by composition, the program's execution time is not a function of its inputs. Unfortunately, this paradigm is difficult to follow today because processors may each employ different software-invisible optimizations that turn different sets of instructions into operand timing-variable instructions. For example, a multiply instruction may be equipped with a 'skip-on-0' feature. Safe instruction synthesis addresses this issue by analyzing input RTL and determining (synthesizing) the set of safe instructions given that RTL.

Prior work [61] on automated invariant learning for the SISP is based on the MLIS algorithm Sorcar (which improves Houdini's performance by making it property directed). It thus faces the scalability issues described earlier. Specifically, the authors report that learning an inductive invariant for the open-source (in-order) RISC-V Rocketchip core took 8 hours, and were unable to scale to out-of-order cores such as RISC-V BOOM. We experimentally verified that Sorcar-style (monolithic) SMT queries did not scale to BOOM.

We use VeloCT to address these scalability issues. Using VeloCT, we are able to learn an invariant for Rocketchip with no expert annotations in under 10 seconds—a 2880× speedup over prior work. Further, with modest expert annotations, we are able to learn invariants for all variants of BOOM (from SmallBOOM to MegaBOOM) in between 6.95 minutes and 199.1 minutes. We verify that the safe sets generated by VeloCT are consistent with prior work.

3.2 Contributions

To summarize, we make the following contributions:

1. We propose H-Houdini, a scalable invariant learning algorithm that enables MLIS to use incremental, parallel SMT checks.

2. We propose VeloCT, an instance of H-Houdini that enables scalable safe instruction set synthesis given RTL as input.

3. We evaluate VeloCT on open-source processors. On the in-order Rocketchip core (10K state bits), we learn an invariant in under 10 seconds—a 2880× speedup over prior work. Our analysis is the first to be able to automatically learn invariants for the out-of-order BOOM core. Ranging from SmallBOOM (48K state bits) to MegaBOOM (133K state bits), invariant learning takes between 6.95 minutes and 199.1 minutes.

VeloCT is open source, and can be found at [63].

3.3 Background & Motivation

3.3.1 Terminology

We now formally setup the invariant synthesis problem.

Definition 3.3.1. Transition System (TS): We denote a transition system TS by a 3-tuple $(\mathbb{S}, T, s_0)$; where $\mathbb{S}$ is the set of all states, $T : \mathbb{S} \times \mathbb{S}$ is the transition relation, and $s_0 \in \mathbb{S}$ is a special *initial* state. Let each state $s \in \mathbb{S}$ be made up of state variables with identifiers in $\mathbb{V}$.

For explanatory purposes we assume that the TS is a hardware circuit. The transition relation T is equivalent to simulating the circuit for 1 cycle, and each state $s \in \mathbb{S}$ is a mapping from identifiers in $\mathbb{V}$ (e.g., registers) to concrete values.

We are interested in proving a property $\mathbf{P}$ on the transition system TS, e.g., a safety property to show TS does not reach any *Bad* states starting from the initial state s_0. One way to do this to to derive an inductive invariant $\mathbf{H}$ that proves $\mathbf{P}$ holds forever on TS. Intuitively, one can think of *any* invariant $\mathbf{I}$ as forming a set of states. We can test for set membership of a state s, denoted by $\mathbf{I}(s)$: $\mathbf{I}(s)$ is true iff $s \in \mathbf{I}$. Then, an inductive invariant $\mathbf{H}$ is a set that is closed under the transition T and ensures that for any s, if $\mathbf{H}(s)$ is true then $\mathbf{P}(s)$ is also true. Framing this intuition formally:

Definition 3.3.2. Inductive Invariant (H): $\mathbf{H}$ is an inductive invariant for a property $\mathbf{P}$ on TS if the following three conditions hold. (i) *Initiation:* $\mathbf{H}(s_0)$, (ii) *Consecution:* $\forall\ s, s' \in \mathbb{S}$: $\mathbf{H}(s) \wedge T(s, s') \implies \mathbf{H}(s')$, and finally, (iii) $\mathbf{P}$ *holds:* $\forall\ s \in \mathbb{S}$: $\mathbf{H}(s) \implies \mathbf{P}(s)$.

For brevity we will use $\mathbf{H} \implies \mathbf{H}'$ as a shorthand to denote the consecution check on $\mathbf{H}$, leaving out quantifiers and the transition relation. Similarly, we will use a shorthand of the form $\mathbf{H} \implies \mathbf{P}$ to denote $\forall\ s \in \mathbb{S}$: $\mathbf{H}(s) \implies \mathbf{P}(s)$, leaving out the quantifiers when the context is clear.

Definition 3.3.3. Invariant Synthesis Problem: Given a transition system TS and a property $\mathbf{P}$, synthesize an inductive invariant $\mathbf{H}$ to prove $\mathbf{P}$ or return None to indicate that no such $\mathbf{H}$ exists.

In the above definitions, we use the concept of *monolithic induction* to describe the consecution requirement (ii). However, induction can also be applied incrementally through *relative induction*, whose concepts are defined below.

Definition 3.3.4. Relative Inductivity: We say $\mathbf{H}$ is relatively inductive to $\mathbf{G}$ if $\mathbf{G} \wedge \mathbf{H} \implies \mathbf{H}'$. Notice that if we set $\mathbf{G}$ to *true*, then relative inductivity reduces to the well-known monolithic inductive query. These relative inductive queries are a crucial part in making SAT-based invariant learning algorithms like IC3/AVR incremental.

Definition 3.3.5. Abduct (A): Consider a formula of the form $\mathbf{H} \implies \mathbf{H}'$ that does not hold initially. We define an abduct $\mathbf{A}$, the result of an *abductive query* [166], as a formula that "fixes" the above implication, i.e., $\mathbf{A} \wedge \mathbf{H} \implies \mathbf{H}'$ holds and $\mathbf{A}$ is *non-contradictory* with $\mathbf{H}$.

Remark. $\mathbf{A} \wedge \mathbf{H}$ *is non-contradictory if* $\exists x, x \models \mathbf{A} \wedge \mathbf{H}$. *This is to ensure that the implication is not vacuously true.*

3.3.2 Machine Learning-based Invariant Synthesis

Machine Learning-based Invariant Synthesis (MLIS) frames the problem of learning an invariant as an interaction between a teacher and a learner, much like a machine learning problem. The learner (who does not see TS) has to learn an invariant based on the examples returned by the teacher (who does see TS). An example can be of three types: (i) A *positive example* that the proposed invariant needs to allow, (ii) A *negative example* that shows that the invariant allows a *Bad* state, or (iii) An *implication example* that shows why the invariant is not inductive. The algorithm proceeds in rounds where the learner uses the counterexamples to refine and propose better invariants in subsequent rounds.

3.3.2.1 HOUDINI overview

Next, we review the most popular MLIS algorithm: HOUDINI [82]. HOUDINI takes as inputs the set of all *predicates* $\mathbb{P}$, a set of positive examples $\mathcal{E}$, and a property of interest $\mathbf{P}$ and outputs either an invariant $\mathbf{H}$ if successful, or None if it fails. A predicate p is a formula in first-order logic over a subset of variables $\mathbb{V}_p \subseteq \mathbb{V}$. Given a state s, p will evaluate either evaluate to `True`, denoted by $s \models p$ or `False` denoted by $s \not\models p$.

HOUDINI first sifts the predicate set $\mathbb{P}$ through all the positive examples to eliminate all predicates that do not hold. That is, it performs

$$\mathbb{P}^* = \{p \mid p \in \mathbb{P} \text{ and } \forall_{e \in \mathcal{E}}\ e \models p\}.$$

This step greatly shrinks the invariant search space. At each point hereafter, the current candidate invariant is formed by conjuncting over predicates in the current $\mathbb{P}^*$, i.e., $\mathbf{H}_{candidate} = \bigwedge_p \mathbb{P}^*_p$. Next, the algorithm loops until the invariant is inductive. For each counterexample to inductivity *cex* found, the algorithm filters the set of remaining predicates as

$$\mathbb{P}^* = \{p \mid p \in \mathbb{P}^* \text{ and } cex \models p\}.$$

Each inductivity check is a query to a theorem prover. We assume Satisfiability Modulo Theory (SMT) solvers are used. HOUDINI guarantees that if an invariant exists as a conjunction of a subset of $\mathbb{P}$, it will be found. This encourages a "kitchen sink" approach to automated invariant learning — experts throw in all predicates that *might* be useful and HOUDINI searches for an inductive invariant.

Note, we can augment the original HOUDINI algorithm to take $\mathbf{P}$ as an input and check to see if the inductive invariant satisfies $\mathbf{P}$ (returning None if not).[2] SORCAR improves HOUDINI by making it *property directed* where the property is checked throughout invariant synthesis and used to further prune the search space.

[2]The original algorithm assumes no annotations, e.g., assertions, and tries to find *any* inductive invariant.

3.3.2.2 Limitations of HOUDINI

Each query made by HOUDINI is monolithic over the set of remaining predicates $\mathbb{P}^*$, which is $O(|\mathbb{P}|)$. Following the kitchen sink philosophy, this means that queries tend to be over a large number of predicates (even predicates that turn out not to be useful in constructing the final invariant). Further, these expensive monolithic checks are made in the 'inner-most loop' of the algorithm—for each inductivity check.

These characteristics prevent HOUDINI from scaling to our problems of interest. Specifically, in our experience and as supported by prior work [57], even a single SMT query over predicates spanning the BOOM microarchitecture is prohibitively expensive and beyond the capabilities of current automated verification tools.

3.4 H-Houdini

We now describe our invariant learning algorithm, H-HOUDINI[3]. In a nutshell, H-HOUDINI replaces HOUDINI's sequential monolithic inductivity checks with a "wave" of property-directed, incremental, memoizable and parallelizable inductivity checks. These attributes result in better efficiency and scalability.

We start by explaining the ideas conceptually. For simplicity, let the property $\mathbf{P}$ be a single predicate $p_{target} \in \mathbb{P}$. (More generally, $\mathbf{P}$ can be a conjunct over a subset of $\mathbb{P}$. In that case, the below explanation is repeated for each predicate in $\mathbf{P}$. Memoization ensures that efficiency is unaffected.)

The insight is that to show p_{target} holds, it is sufficient to require that a subset of the predicates directly influencing p_{target} be included in the final invariant. That is, it is sufficient to find an abduct $\mathbf{A}$ for p_{target} that only contains predicates whose state elements can influence p_{target} in the next step of the transition system.

For example, consider an AND gate whose inputs and output are clocked state elements. If p_{target} says that the output state element must always hold a 1, it is sufficient to require that the final invariant include predicates that force the input state elements to hold 1s.

The above creates new proof obligations. Namely, we must now show for each predicate $p \in \mathbf{A}$ that including p in the invariant leads to an inductive invariant. For this, we recursively repeat the above procedure for each p.

The recursion creates a wavefront of proof obligations that resembles a depth-first search (DFS) over the design's state elements (predicates). If an abduct for some p forces us to include a predicate p' that makes the invariant not inductive, we backtrack and ask for a different abduct for p (that does not include p'). If one does not exist, we backtrack further. Like a normal DFS, once we 'visit' each predicate once (either prove that including it either can or does not lead to an inductive invariant), we need not recurse through it again. Once the wavefront terminates, the hierarchy of abducts are combined to form the overall invariant.

[3] H-HOUDINI stands for Hierarchical HOUDINI—a wordplay on 'Harry HOUDINI,' the illustrious magician's full name.

This can be done without ever performing a monolithic inductivity check, by construction of the composition of abducts.

We design the above process to be property-directed, incremental, memoizable and parallelizable. For property directed, we pre-filter the set of predicates that can be considered for each abduct to only be those that are consistent with positive examples (for which we believe the property holds) and are helpful in proving the property. For incremental, we further pre-filter predicates to those that can influence the current p_{target} within one step of the transition system. Both of the above decrease the cost of synthesizing each abduct (which is done using SMT queries) in the common case. When the recursion over $p \in \mathbf{A}$ completes and $\mathbf{A}$ is either accepted as the solution for p_{target} or the solver says there is no solution for p_{target}, p_{target} is considered solved and need not be explored again. Finally, synthesizing each abduct can be done in parallel.

We now describe the above more formally, proceeding as follows: Section 3.4.1 argues that the main premise in H-HOUDINI is sound; Section 3.4.2 presents the H-HOUDINI algorithm in detail; Section 3.10 provides proof sketches of soundness and completeness.

3.4.1 Hierarchical Decomposition is Sound

The main premise in H-HOUDINI is that we can decompose a monolithic invariant into a hierarchy of smaller invariants, and that to prove inductivity it is sufficient to perform SMT queries over only the smaller invariants. We now argue why this approach is sound, breaking the argument into two parts. First, we will demonstrate how an inductive invariant can be synthesized incrementally using relatively inductive queries and prove that this approach is sound provided certain assumptions hold at each step. Next, we will show how exploiting the hierarchy of the circuit to learn incrementally satisfies these assumptions.

Suppose we wish to prove the property $\mathbf{H}_0$. We start by finding an $\mathbf{H}_1$ such that $\mathbf{H}_0$ is relatively inductive w.r.t. $\mathbf{H}_1$:

$$\mathbf{H}_1 \wedge \mathbf{H}_0 \implies \mathbf{H}_0' \tag{3.1}$$

Next, repeat the process. We find an $\mathbf{H}_2$ such that:

$$\mathbf{H}_2 \wedge \mathbf{H}_1 \implies \mathbf{H}_1' \tag{3.2}$$

Let us assume that $\mathbf{H}_0$ and $\mathbf{H}_2$ are non-contradictory, i.e., $\exists x, x \models \mathbf{H}_0 \wedge \mathbf{H}_2$. Then, from Equation 3.1 and Equation 3.2, it follows:

$$\mathbf{H}_2 \wedge \mathbf{H}_1 \wedge \mathbf{H}_0 \implies \mathbf{H}_1' \wedge \mathbf{H}_0'$$

This means $\mathbf{H}_1 \wedge \mathbf{H}_0$ is relatively inductive to $\mathbf{H}_2$. We recursively repeat this process, giving the following n equations:

$$\begin{aligned} \mathbf{H}_{j+1} \wedge \mathbf{H}_j &\implies \mathbf{H}_j' \quad j \in [0, k) \\ \mathit{true} \wedge \mathbf{H}_j &\implies \mathbf{H}_j' \quad j \in [k, n) \end{aligned}$$

where $true \wedge \mathbf{H}_j \implies \mathbf{H}'_j$ denote base cases in the recursion, i.e., fragments of $\mathbf{H}$ that are inductive by themselves (e.g., module inputs, constants). Applying the same argument from above, we get:

$$true \wedge \mathbf{H}_n \wedge \ldots \wedge \mathbf{H}_1 \wedge \mathbf{H}_0 \implies \mathbf{H}'_n \wedge \ldots \wedge \mathbf{H}'_1 \wedge \mathbf{H}'_0$$

We can denote the monolithic invariant as $\mathbf{H} = \wedge_{i=0}^{n} \mathbf{H}_i$, in which case the above statement is equivalent to $\mathbf{H} \implies \mathbf{H}'$. Lastly, we have $\mathbf{H}_0$, the property of interest as a part of $\mathbf{H}$, and so $\mathbf{H} \implies \mathbf{H}_0$ trivially. Importantly, we did not need to prove inductivity of the monolithic invariant directly.

Now, if every $\mathbf{H}_i$ we construct allows every positive example, i.e., $\forall_{e \in \mathcal{E}}\ e \models H_i$, then $\mathbf{H}$ is the required inductive invariant derived incrementally. To summarize this discussion, hierarchical incremental invariant learning is sound as long as the following premise holds:

Premise for Soundness (P-S). $\mathbf{H}_i$ *should allow* all *positive examples:* $\forall_{e \in \mathcal{E}}\ e \models H_i$.

Remark. *(P-S) ensures* $\mathbf{H}_i$ *are not contradictory.*

The positive examples $\mathcal{E}$ act as witnesses to the fact that none of the $\mathbf{H}_i$ are contradictory as clearly they allow states in $\mathcal{E}$. This prevents the derived $\mathbf{H}$ from being vacuously true by pruning out all states $s \in \mathbb{S}$.

3.4.2 H-Houdini Algorithm

Now we describe the H-HOUDINI algorithm, a concrete instance of an incremental learner that exploits hierarchy to soundly and incrementally learn an invariant.

See algorithm 3. The algorithm has inputs similar to the original HOUDINI but with two differences. First, the predicate universe $\mathbb{P}$ is replaced by an oracle $\mathcal{O}^{\mathcal{A},\mathcal{E}}_{mine}$ which is instantiated with the set of positive examples $\mathcal{E}$ and optionally expert annotations $\mathcal{A}$. Second, as H-HOUDINI is white box, it takes in the transition system TS. Note, p_{target} is analogous to $\mathbf{P}$ (Section 3.4). H-HOUDINI outputs an inductive invariant ($\mathbf{H}$) that proves p_{target}, or `None` if no such invariant exists.

The algorithm proceeds as follows. We define a *solution* for a predicate p_{target} as an abduct that has been proven to be inductive or None (meaning p_{target} cannot appear in the final invariant). To start, if a solution to p_{target} has already been found, it is returned immediately via memoization (line 3). Otherwise, the algorithm enters a loop to find a new solution (line 7). Within the loop, the algorithm first sets $\mathbf{H}$ to p_{target}. Next, we invoke three subroutines.

First, $\mathcal{O}^{TS}_{slice}$ (the slicing oracle) takes as input p_{target} and outputs the set of state elements $\mathbb{V}_{slice}$ that influence the inductivity of p_{target} in one step of TS. When verifying sequential circuits (as we do in this paper), these are the state elements in the 1-step cone-of-influence (COI) for p_{target}.[4] That is, in our AND gate example (Section 3.4), if p_{target} is over the output register, $\mathbb{V}_{slice}$ is the set of input registers.

[4]Given a software verification task, the 1-step COI of a statement X is the data and control-dependencies of X.

Algorithm 3: The H-HOUDINI algorithm.

Input : $\mathcal{O}^{\mathcal{A},\mathcal{E}}_{mine}$: The predicate mining oracle, p_{target}: Target predicate/property, TS: Transition system

Output: **H**: invariant that proves p_{target} or None

$\mathbb{P}_{fail} = \varnothing$;
def H-HOUDINI ($\mathcal{O}^{\mathcal{A},\mathcal{E}}_{mine}$, p_{target}, TS) $\rightarrow$ *None*/**H:**
  if p_{target} *is memoized and soln* $\cap\ \mathbb{P}_{fail} = \varnothing$ **then**
    return solution to p_{target};
  end
  valid-solution = False;
  while *not valid-solution* **do**
    $\mathbf{H} = p_{target}$;
    $\mathbb{V}_{slice} = \mathcal{O}^{TS}_{slice}(p_{target})$;
    $\mathbb{P}_{\mathbb{V}} = \mathcal{O}^{\mathcal{A},\mathcal{E}}_{mine}(p_{target}, \mathbb{V}_{slice})$;
    $\mathbb{P}_{\mathbb{V}} = \mathbb{P}_{\mathbb{V}} \setminus \mathbb{P}_{fail}$;
    $\mathbf{A} = \mathcal{O}_{abduct}(p_{target}, \mathbb{P}_{\mathbb{V}})$;
    set **A** as the memoized solution to p_{target};
    if A *is None* **then**
      return None;
    end
    valid-solution = True;
    for p *in* **A do**
      $\mathbf{H}_{sol}$ = H-HOUDINI ($\mathcal{O}^{\mathcal{A},\mathcal{E}}_{mine}$, p, TS);
      if $\mathbf{H}_{sol}$ *is None* **then**
        valid-solution = False;
        $\mathbb{P}_{fail} = \mathbb{P}_{fail} \cup \{p\}$;
        break;
      end
      $\mathbf{H} = \mathbf{H} \wedge \mathbf{H}_{sol}$
    end
  end
  return H;
end

Second, $\mathcal{O}^{\mathcal{A},\mathcal{E}}_{mine}$ (the mining oracle) translates $\mathbb{V}_{slice}$ into a set of predicates $\mathbb{P}_{\mathbb{V}}$ that will be considered when synthesizing abducts. This takes into account the user-specified predicate language, the positive examples $\mathcal{E}$ and (optionally) additional expert annotations ($\mathcal{A}$).[5]

Third, $\mathcal{O}_{abduct}$ (the abduction oracle) takes $\mathbb{P}_{\mathbb{V}}$ and attempts to synthesize an abduct **A**

[5]The mining oracle we describe here is deterministic. Our implementation adopts an incremental variant that returns steadily larger subsets of $\mathbb{P}_{\mathbb{V}}$ over multiple calls, to encourage smaller abducts. See Section 3.4.2.3.

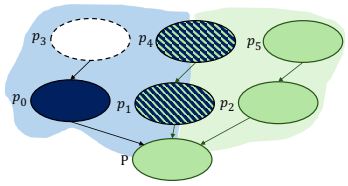


Figure 3.1: Dependency graph between predicates in a TS and $\mathbb{P}$. Each node is a predicate p_i, and edge $p_i \rightarrow p_j$ means p_i is in the 1-step COI of p_j. The hollowed out predicate p_3 does not hold inductively and cannot be included in an invariant. The figure shows that there are two potential solutions, the nodes in the blue area and the nodes in the green area—both of which share the common set of nodes in the middle. The blue nodes eventually fail because of p_3, causing H-HOUDINI to backtrack and discover the solution in green. p_1 and p_4 need only be analyzed once.

out of a subset of $\mathbb{P}_\mathbb{V}$ for p_{target} (line 12). If multiple abducts are found, one is returned; if $\mathcal{O}_{abduct}$ is called with the same p_{target} multiple times, we require a different abduct be returned each call (as in an iterator). If no abduct is found (line 15), the algorithm returns `None`, indicating that no solution exists. In our implementation, each abduct returned costs an SMT query. More details for $\mathcal{O}_{abduct}$ are given in Section 3.4.2.3.

If an abduct is found, it is added as the solution to p_{target}, and `valid-solution` is set to `True`. The algorithm then checks each predicate in the abduct recursively (line 18). For each predicate, it calls H-HOUDINI to find a solution. If any predicate fails to hold inductively (line 21), `valid-solution` is set to `False`, and the loop exits early to attempt a new solution for p_{target} (line 7). If all predicates hold, the invariant is updated with the solutions found, and the algorithm returns $\mathbf{H}$ (line 28).

3.4.2.1 Backtracking and Memoization

The above algorithm partially backtracks when it discovers that a solution is not possible. To illustrate this, look at the example in Figure 3.1. In this figure, we show a predicate dependency graph. Each node in the graph is a predicate p_i and an edge from $p_i \rightarrow p_j$ denotes that p_i is in the 1-step COI of p_j. Suppose p_3 is not inductive. We start the algorithm from predicate $\mathbf{P}$. Suppose there are two possible solutions for $\mathbf{P}$: the solution in the blue region $p_0 \wedge p_1$ or the solution in the green region $p_1 \wedge p_2$. These solutions would be returned through successive queries to $\mathcal{O}_{abduct}$ as discussed previously.

Suppose $\mathcal{O}_{abduct}$ first returns the blue solution. Then H-HOUDINI fires off on subtrees p_0 and p_1. Eventually, because p_3 fails to be inductive, H-HOUDINI discovers that p_0 cannot hold inductively and therefore backtracks to synthesize a new solution for $\mathbf{P}$. The next query to $\mathcal{O}_{abduct}$ for $\mathbf{P}$ returns the green solution as p_0 is no longer available. But the work we need to do to ratify the green solution is significantly reduced since we memoized the solution for p_1. The only additional work to do to complete the invariant is to synthesize the solution for the nodes colored green: p_2 and p_5. In this way, the algorithm only backtracks partially.

H-HOUDINI only squashes the path of failure while the rest of the synthesized solutions may be reused.

Backtracking is mainly caused by deficiencies in positive examples ($\mathcal{E}$). If $\mathcal{E}$ was exhaustive, then a positive example $e \in \mathcal{E}$ would have been a witness to invalidate p_3 and p_0 as candidate predicates, allowing us to synthesize the green solution first and eliminate the backtrack. With a robust set of examples, a majority of the backtracking can be eliminated.

3.4.2.2 Cycles

We note, the recursion can encounter a cycle. For example, if the hardware design has a backedge we can encounter $p_i \rightarrow p_j \rightarrow p_i$. This is benign. The pending (memoized) solution for p_i will be used to verify p_j. If later it is found that p_i has no solution, we must re-synthesize a solution for p_j. This is done by the early return (line 3) performing an intersection over $\mathbb{P}_{fail}$.

3.4.2.3 Details for Abduction Oracle $\mathcal{O}_{abduct}(p_{target}, \mathbb{P}_\mathbb{V})$

$\mathcal{O}_{abduct}$ takes a target predicate p_{target} and a set of predicates ($\mathbb{P}_\mathbb{V}$) and outputs an abduct $\mathbf{A}$, a conjunction over a subset of $\mathbb{P}_\mathbb{V}$ that shows p_{target} is 1-step inductive. Formally: $\mathbf{A} \implies p'_{target}$. Such an $\mathbf{A}$ can be efficiently computed using Craig's interpolants [50, 146]. We compute the interpolants quickly using the following (to our knowledge, novel) SMT query:[6]

$$\bigwedge_v \mathbb{P}_{\mathbb{V}v} \wedge p_{target} \wedge \neg p'_{target}$$

If the above query returns SAT, then there is no conjunction over $\mathbb{P}_\mathbb{V}$, such that p_{target} is relatively inductive. Thus, we return `None`. Otherwise, if the solver returns UNSAT, we extract the predicates in the UNSAT core from the solver and use it as the abduct. Note that since $\bigwedge_v \mathbb{P}_{\mathbb{V}v} \wedge p_{target}$ cannot contain any contradictions, the UNSAT-ness of the query has to be because of its interaction with $\neg p'_{target}$, thereby making extraction of these abducts sound.

To minimize work and final invariant size, we bias $\mathcal{O}_{abduct}$ to output the weakest (smallest) abduct possible. For any two abducts $\mathbf{A_1}$ and $\mathbf{A_2}$, $\mathbf{A_1}$ is weaker than $\mathbf{A_2}$ if $\mathbf{A_2} \implies \mathbf{A_1}$. Ideally, we desire a minimal abduct $\mathbf{A_m}$, defined as one where $\nexists\ \mathbf{A}$ such that $\mathbf{A_m} \implies \mathbf{A}$. Our implementation extracts small (ideally minimal) abducts by enabling the `minimal unsat cores` option in cvc5 which guarantees locally minimal unsat cores.

3.4.2.4 Parallelism

The serial implementation of H-HOUDINI lends itself to a high degree of parallelism. For example, the loop that recursively checks each predicate in the abduct (line 18-line 25) can be performed in parallel. We parallelize this loop in our implementation and evaluation.

[6] We tried other methods, e.g., cvc5's native $\mathcal{O}_{abduct}$ call. Our method was fastest in practice.

More aggressive forms of speculative parallelism, e.g., trying different abducts in parallel, are possible. We leave more detailed investigation of these to future work.

3.4.2.5 Formal requirements for soundness and completeness

H-HOUDINI requires any downstream analysis to satisfy two contracts for soundness and completeness. We present details for these in Section 3.10.

3.5 VeloCT: Preliminaries

The next three sections describe VELOCT, which instantiates H-HOUDINI to solve the safe instruction set synthesis problem (SISP) [61, 57, 60, 83]. This section gives terminology (taken mostly from [61]); Section 3.6 describes the analysis' design in detail; Section 3.12 describes a worked example of the analysis on a simple microarchitecture.

In VELOCT, the transition system TS is the hardware circuit described in Section 3.3. We also introduce a set of inputs Σ to TS, input during transitions, to be either ISA instructions or the special null input ϵ indicating 'no instruction' (following [61]). Next, we define the main concepts in the SISP.

Definition 3.5.1. Trace (π): A trace of TS over a sequence of inputs starting at state s_0 is a sequence of states in TS obtained by applying the transition relation T ($\rightsquigarrow$): $s_0 \rightsquigarrow^{a_0} s_1 \ldots \rightsquigarrow^{a_k} s_{k+1}$ where $a_i \in \Sigma$ denotes a_i is input during $\rightsquigarrow$.

Definition 3.5.2. Trace Indistinguishability: Assume that an attacker can observe a subset of elements $O \subseteq \mathbb{V}$. For state s, the values of O (a projection on s) are given by $s \lfloor O$. We say two traces $\pi = s_0 s_1 \ldots$, $\pi' = s'_0 s'_1 \ldots$, are *trace indistinguishable* if $|\pi| = |\pi'|$ and $\forall_j\ j \in [0, |\pi|]$, $s_j \lfloor O = s'_j \lfloor O$ where $|\pi|$ denotes the length of π.

Definition 3.5.3. Equal-modulo-secret I-states: Let us define a subset of state elements to hold secret values: $\mathbb{V}_{sec} \subseteq \mathbb{V}$. Two states s and s' are equal-modulo-secret I-states if $s \approx^{sec} s_0 \approx^{sec} s'$ where s_0 is the init state of TS and $\approx^{sec}$ is an equality relation over the projection of states excluding secrets, i.e., over $\mathbb{V}_{pub} = \mathbb{V} \setminus \mathbb{V}_{sec}$.

Definition 3.5.4. Safe Set ($\widehat{\Sigma^+}$): A set of instructions form a safe set $\widehat{\Sigma^+}$ if, for every sequence of instructions x over $\widehat{\Sigma^+}$, and a pair of equal-modulo-secret I-states $(s^{\mathtt{l}}, s^{\mathtt{r}})$, the pair of traces, $(\pi^{\mathtt{l}}, \pi^{\mathtt{r}})$ of TS starting from $(s^{\mathtt{l}}, s^{\mathtt{r}})$ and given x are trace indistinguishable.

Lastly, we define the *soundness* and *precision* of **H** in verifying safe set $\widehat{\Sigma^+}$. Since **H** is a relational invariant to prove the non-interference property from above, it is defined over a pair of states. For simplicity, we construct a product state s over a product variable space $\mathbb{V} = \mathbb{V}^{\mathtt{l}} \cup \mathbb{V}^{\mathtt{r}}$, where $\mathbb{V}^{\mathtt{x}}$ denotes that each identifier is renamed by adding a prefix $\mathtt{x}$. We say $s = (s_i^{\mathtt{l}} \cdot s_i^{\mathtt{r}})$ to denote the composition of $s^{\mathtt{l}}, s^{\mathtt{r}}$ to form the product state s.

Definition 3.5.5. Admits Traces ($\bowtie^{\mathbf{H}}$): Consider $\pi^{\mathtt{l}} = s_0^{\mathtt{l}}, s_1^{\mathtt{l}}, \ldots$ and $\pi^{\mathtt{r}} = s_0^{\mathtt{r}}, s_1^{\mathtt{r}}, \ldots$. For simplicity assume the shorter trace is padded with special null states ($\varnothing$). Next, construct product states, s_i, between corresponding states in $\pi^{\mathtt{l}}$ and $\pi^{\mathtt{r}}$, i.e., $\forall_{i \in |\pi^{\mathtt{l}}|} s_i = (s_i^{\mathtt{l}} \cdot s_i^{\mathtt{r}})$. Lastly, define a relation $\bowtie^{\mathbf{H}}$: $\pi \times \pi$ that relates the pair $(\pi^{\mathtt{l}}, \pi^{\mathtt{r}})$ if $\forall_{i \in |\pi^{\mathtt{l}}|} s_i \models \mathbf{H}$.

Definition 3.5.6. Soundness of H: $\mathbf{H}$ is sound if for all pairs of traces $\pi^{\mathtt{l}}, \pi^{\mathtt{r}}$ that are *not* trace indistinguisible, $(\pi^{\mathtt{l}}, \pi^{\mathtt{r}}) \notin \bowtie^{\mathbf{H}}$.

Definition 3.5.7. Precision of H: $\mathbf{H}$ is precise for a safe set $\widehat{\Sigma^+}$ if for all sequences of instructions x over $\widehat{\Sigma^+}$ and starting with all pairs of equal-modulo-secret I-states, the pairs of generated traces $(\pi^{\mathtt{l}}, \pi^{\mathtt{r}}) \in \bowtie^{\mathbf{H}}$.

Definition 3.5.8. Positive Example ($e \in \mathcal{E}$): Each example is a product state $s = (s_i^{\mathtt{l}} \cdot s^{\mathtt{r}})$ of two states $s^{\mathtt{l}}$ and $s^{\mathtt{r}}$. A positive example, w.r.t. a property $\mathbf{P}$, is an example which satisfies the property ($s \models \mathbf{P}$) *and* on application of the transition function T leads to either the terminating state or another positive example.

3.6 VeloCT

In this section, we will describe VeloCT, a framework to learn relational invariants that prove security properties, e.g., non-interference, for hardware designs. At a high-level, VeloCT takes a hardware design in RTL, an annotation that denotes attacker observable output, e.g., the instruction retirement signal, and a proposed set of safe instructions $\widehat{\Sigma^+}$— and outputs either an inductive invariant $\mathbf{H}$ that proves the safety of the proposed safe set or outputs None if it doesn't exist.

VeloCT implements H-Houdini for invariant learning and starts learning from $\mathbf{P}$. For simplicity assume O consists of a single state element (call it v_o). Taking inspiration from Def. 3.5.2, we need to prove that the values of $v_o^{\mathtt{l}}$ and $v_o^{\mathtt{r}}$ are always equal. This property is expressed formally in terms of an `Eq`-type predicate (defined in the next section) as: $\mathtt{Eq}(v_o^{\mathtt{l}}, v_o^{\mathtt{r}})$.

In what follows, we first define the predicate language (Section 3.6.1.1) and the implementation of the $\mathcal{O}_{mine}$ (Section 3.6.1.2). Then, we discuss how VeloCT generates examples in Section 3.6.2. Lastly, we conclude the section with a proof that $\mathbf{H}$ from VeloCT is sound (Def. 3.5.6) and precise (Def. 3.5.7) in Section 3.11.

3.6.1 Predicate Language and Mining

We now describe the predicate language and implementation of $\mathcal{O}_{mine}$ in VeloCT. A good predicate language, coupled with an automated way to mine predicates from examples, is essential for proving properties with H-Houdini with minimal annotations.

3.6.1.1 Predicate Language

We start with the predicate language proposed by ConjunCT [61]. We observe that for real-world processor designs we can actually simplify the predicate language. To prove a set of instructions form a safe set as defined in Def. 3.5.3, we do not need the expressiveness offered by `Impl`-type predicate introduced in ConjunCT. Rather, we can include a simpler predicate that restricts a register to hold certain safe values(s). With this intuition, VeloCT implements three *base* types of predicates:

`Eq(`$v_{\texttt{l}}, v_{\texttt{r}}$`).` This is the same `Eq-type` predicate from ConjunCT. It constrains v to hold the same value in `l` and `r` executions. Intuitively, it disallows executions where v is influenced by secrets.

`EqConst(`v`,` val`).` The `EqConst-type` predicate constrains v to take a constant value val. This allows us to express invariants where the safety is dependent on $v = val$.

`EqConstSet(`v`,` $[val_1, \ldots, val_n]$`).` The `EqConstSet-type` predicate is a generalization of the above `EqConst` predicate. Rather than restricting v to take a particular value, it restricts it to a set of values.

Note that both `EqConst` and `EqConstSet` also constrain the variable v to be equal in left and right executions (i.e., they are also, implicitly, `Eq`-type predicates).

The `InSafeSet` predicate. Critically, the predicate abstraction must be expressive enough to capture the target invariant. This creates an issue specific to SISP. Recall, SISP requires that compositions of safe instructions satisfy the property. Yet, a microarchitecture may still support unsafe instructions. Thus, we need a predicate that constrains the invariant to only consider executions made up of compositions of the safe instructions (ignoring/disallowing compositions that include unsafe instructions).

To address this, we instantiate a specialized flavor of `EqConstSet` – the `InSafeSet` predicate – to constrain state elements in the pipeline to include only bit patterns that are consistent with the safe instructions. For a given set of safe instructions, these bit patterns (mask and match values) are automatically generated from the RISC-V specification.

3.6.1.2 $\mathcal{O}_{mine}$: Predicate Mining

The predicate mining algorithm is based on a key insight: positive examples contain the *core of safety*, a skeleton that represents the *necessary* and *sufficient* conditions for an execution to be safe. The primary challenge is then to learn an invariant strong enough to prove safety and weak enough to generalize to safe executions beyond positive examples $\mathcal{E}$ (the precision definition Def. 3.5.7).

At a high-level, the algorithm follows these two rules: (i) only consider $\mathbb{V}_{Eq} \subseteq \mathbb{V}_{slice}$ that are equal in $\mathcal{E}$ (line 2), and (ii) check-and-add: add `Eq` (line 5), `EqConst` (line 7), and `InSafeSet` (line 11) predicates only when they are consistent with $\mathcal{E}$. Outside the loop, the check-and-add rule is extended to the expert predicates generated by $\mathcal{A}(\mathbb{V}_{Eq})$ (line 15). Finally, the set of predicates, $\mathbb{P}_{\mathbb{V}} = \mathbb{P}_{Eq} \cup \mathbb{P}_{expert}$ is returned.

Algorithm 4: $\mathcal{O}^{\mathcal{A},\mathcal{E}}_{mine}$: Predicate mining algorithm.

Input : p_{target}: Target predicate to mine predicates for, $\mathbb{V}_{slice}$: Subset of $\mathbb{V}$ output by $\mathcal{O}_{slice}$ to consider when mining predicates.
Output: $\mathbb{P}_{\mathbb{V}}$: Set of predicates $\mathbb{P}_{\mathbb{V}} \subseteq \mathbb{P}$ to use with $\mathcal{O}_{abduct}$.

def $\mathcal{O}^{\mathcal{A},\mathcal{E}}_{mine}(p_{target}, \mathbb{V}_{slice}) \rightarrow \mathbb{P}_{\mathbb{V}}$**:**
  $\mathbb{V}_{Eq} = \{v \mid v \in \mathbb{V}_{slice}$ and $\forall_{e \in \mathcal{E}}\ e[v^{\mathtt{l}}] = e[v^{\mathtt{r}}]\}$;
  $\mathbb{P}_{Eq} = \varnothing$;
  for v *in* $\mathbb{V}_{Eq}$ **do**
    $\mathbb{P}_{Eq} = \mathbb{P}_{Eq} \cup$ $\mathtt{Eq}(v^{\mathtt{l}}, v^{\mathtt{r}})$;
    if $\exists\ c\ \forall_{e \in \mathcal{E}}\ e[v] = c$ **then**
      $\mathbb{P}_{Eq} = \mathbb{P}_{Eq} \cup \mathtt{EqConst}(v^{\mathtt{l}}, c)$;
    end
    $p_{safe} = \mathtt{InSafeSet}(v^{\mathtt{l}})$;
    if $\forall_{e \in \mathcal{E}}\ e \models p_{safe}$ **then**
      $\mathbb{P}_{Eq} = \mathbb{P}_{Eq} \cup p_{safe}$;
    end
  end
  $\mathbb{P}^{*}_{expert} = \mathcal{A}_{mine}(\mathbb{V}_{Eq})$;
  $\mathbb{P}_{expert} = \{p \mid p \in \mathbb{P}^{*}_{expert}$ and $\forall_{e \in \mathcal{E}}\ e \models p\}$;
  $\mathbb{P}_{\mathbb{V}} = \mathbb{P}_{Eq} \cup \mathbb{P}_{expert}$;
  return $\mathbb{P}_{\mathbb{V}}$;
end

Even expert-provided predicates are validated against $\mathcal{E}$ before they are added to the final set. This validation ensures that experts can freely propose any predicates without risking unsoundness. We concretely discuss the expert predicates needed for the verification of BOOM in Section 3.7.2.

In Section 3.11.1 we show how this implementation of $\mathcal{O}_{mine}$ is sound and complete when using H-Houdini. Next, we discuss how to generate these positive examples for learning, and conclude by showing that it is sufficient to derive a precise invariant.

3.6.2 Example Generation

The high-level goal of example generation is to take the set of proposed safe instructions and produce a set of positive examples, $\mathcal{E}$, sufficient for learning an inductive invariant.

Our positive example generation strategy is simple yet robust. We simulate a pair of concrete executions for each safe instruction that differ only in the operand values given to the safe instruction. As we prove in Section 3.11, the specific values of the secret data are irrelevant to soundness or precision; they only need to differ between the two traces.

Each pair of traces yields multiple positive examples, one per state in the product trace, as defined in Def. 3.5.8. Specifically, we extract a positive example for each pair of states

where the safe instruction under analysis is in flight in the trace.

Cleaning positive examples. Positive examples must be carefully prepared so that they are what we refer to as *clean*. To be clean, each example must correspond to a pipeline state that only reflects the execution of the safe instruction under analysis, as opposed to the safe instruction plus potentially other unsafe instructions. Should the state reflect the execution of an unsafe instruction(s), the positive example may remove predicates that are important to learn the invariant.

Constructing clean positive examples is non-trivial. To run a safe instruction on the pipeline, our infrastructure executes start-up and tear-down code containing unsafe instructions. This code may make an example *dirty* for two reasons. First, if an unsafe instruction(s) is in flight in the same cycle as the safe instruction. Second, if an unsafe instruction (not concurrently in flight) leaves a residue in the pipeline that is still present when the safe instruction executes.

We handle the first issue by padding the safe instruction (before and after its execution) with NOPs. Given sufficient padding, this ensures that no unsafe instructions are in flight alongside the safe instruction. Handling the second issue requires more care for the out-of-order (OoO) processors we evaluate, as explained below.

3.6.2.1 Example Masking

Ensuring no unsafe instruction residue remains in the pipeline is non-trivial in OoO processors because OoO designs rely on instruction tables (e.g., the reorder buffer, instruction queues). Instruction state in these tables typically persists after the instruction(s) logically leaves the table. Consider an excerpt of a BOOM issue slot in a positive example:

Index	valid?	Uop	lrs1	...
1	0	<UNSAFE Uop>	0b00001	...

This entire table entry is unused/ineffectual as the valid bit is a 0 in this positive example. The `Uop` field (`Uop` for short) being set to an `UNSAFE Uop` has no bearing on this example's safety. At the same time, this singular example prevents us from adding a predicate that constrains `Uop` to allow only `SAFE Uops` as such a predicate would not allow this positive example. Yet, such a predicate is necessary for safety.

We introduce example masking to clean up such dirty traces. VeloCT uses annotations to identify valid bits in key structures, e.g., issue/reorder buffer units. Then, VeloCT pre-processes positive examples to set entries to their reset values if the entry is marked as `invalid`. We give a detailed description of the annotations necessary for example masking on BOOM in Section 3.7.2. Note that this was only required for complex OoO cores such as BOOM. Simple in-order cores like Rocketchip do not require example masking or annotations.

Since VeloCT is the first effort to scale invariant learning to a large OoO like BOOM, we focus on demonstrating feasibility rather than perfecting the approach. In the future, we

aim to explore other alternatives: e.g., a richer predicate language that includes `Impl`-type predicates from ConjunCT to conditionally constrain `Uop` to allow only `SAFE Uops` only when the entry is valid.

3.7 Evaluation

In this section, we evaluate VeloCT on real-world processor designs. First, Section 3.7.3 evaluates analysis efficiency and scalability. Second, Section 3.7.4 evaluates security.

3.7.1 Implementation & Evaluation Setup

We implement VeloCT in $\sim$ 2300 lines Python. The current implementation accepts hardware designs and assertions for safety in the `btor` [158] format. The current implementation implements algorithm 3 and parallelizes each iteration of the inner loop (as described in line 18).

Evaluation Setup. We ran our evaluation on a machine equipped with Dual Intel Xeon Gold 6148 CPUs (2 sockets, 40 virtual cores/socket), 256GB of memory and Ubuntu 20.04 LTS (kernel version 5.4.0). We also evaluated on an Anyscale Ray cluster (configured with compute-optimized C6i EC2 instances) [15]. Lastly, all code was evaluated on `Python 3.8` and with `CVC5 v1.1.12` as our choice of SMT solver.

Descriptions of Evaluated Designs. We evaluate VeloCT on Rocketchip [18] and all four supported variants of BOOM [240]: SmallBOOM, MediumBOOM, LargeBOOM, MegaBOOM. All targets were compiled using the standard configuration with the Chipyard [12] workflow. We added additional annotations in Chisel to flatten all modules in the generated Verilog. Finally, we use yosys [223] to create a miter (product) circuit, add assertion(s), and emit the output in the `btor` format [158] for VeloCT to consume.

For all designs, we instantiated the top-level module to be the `Core`, e.g., `RocketCore` and `BOOMCore`. See Table 3.1. This does not include the L1 caches. Since memory instructions are known to be unsafe, and no other instructions interact with these structures, caches can be ignored. Our analysis could certainly include them if desired.

Baselines. We compare performance/scalability to the state-of-the-art in safe instruction set synthesis ConjunCT [61]. ConjunCT's analysis is based on Houdini/Sorcar and can learn an invariant (Phase II of the ConjunCT analysis) in $\sim$ 8 hours; it was unable to evaluate on any BOOM variant due to the cost of monolithic SMT queries. We compare security / synthesized safe sets to ConjunCT and UPEC [57]. The latter hand crafts (does not learn) and checks an invariant for MediumBOOM.

Target	Size (in # bits)	Invariant Size
Rocketchip	10,358 bits	145
Small BOOM	48,465 bits	1,609
Medium BOOM	74,072 bits	2,560
Large BOOM	100,009 bits	4,002
Mega BOOM	133,417 bits	4,640

Table 3.1: Overview of evaluated design, their complexity (in # of state bits, in simulation/pre-synthesis), and learned invariant sizes (# of predicates).

3.7.2 Expert Annotation Efforts

VeloCT learns an invariant for Rocketchip with no expert annotations. VeloCT needs a modest number of annotations to learn an invariant for BOOM. There are two kinds of annotations: (i) Annotations required to augment the predicate set with BOOM-specific `EqConstSet` predicates, and (ii) Annotations required for example cleaning.

Annotations to Augment Predicate Set. Recall from Section 3.6.1.1, our current implementation does not automatically insert/mine `EqConstSet` predicates. We augmented $\mathcal{O}_{mine}$ to emit these as expert annotations in two circumstances. First, BOOM decodes RISC-V instructions into micro-ops (uops). Therefore, to constrain executions to only consist of instructions from the safe set, we introduce `InSafeUop` to complement the `InSafeSet` predicate. The `InSafeUop` predicate allows state elements to only take constants that correspond to safe uops. Second, we manually constrain the ALUOpcode signal to correspond to the proposed safe set.

Annotations for Example Masking. Example masking (Section 3.6.2.1) is used to ensure that the positive examples are clean. To do this, we identify and annotate various bits in the design that semantically hold `valid` bits and pair each `valid` bit with its corresponding entries. Concretely, we identified the valid bits for the following microarchitectural structures: ALU issue buffer, CSR issue buffer, FPU issue buffer, JMP unit, and rs2 operand type bits. When the valid bit is semantically 0, we reset the value of its corresponding entries. This step took a grad student (unfamiliar with BOOM) less than a day to find all the required annotations for one design, and another day to generalize to larger BOOM variants.

3.7.3 VeloCT: Performance Evaluation

We now evaluate for performance and scalability.

Scalability and Parallelism. Figure 3.2 shows invariant search time for each design, scaling the number of available cores. We do not evaluate larger designs on smaller core counts due to time constraints and only evaluate LargeBOOM and MegaBOOM on 160 cores. The main takeaway is that doubling the core count proportionally improves analysis

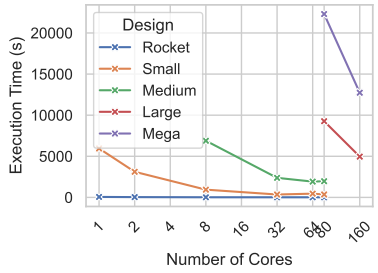


Figure 3.2: Execution Time of VeloCT (in s) vs. # of Parallel Cores on designs of various sizes. VeloCT's execution time consistently halves on doubling the number of cores until a saturation point is reached.

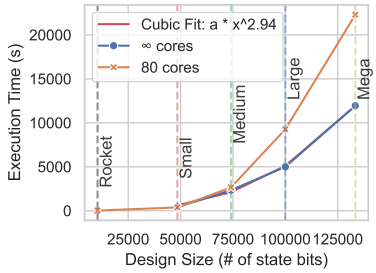


Figure 3.3: Execution Time (in s) of VeloCT vs. Design Size. Plot shows results for an 80-core evaluation and for ∞ cores (on an Anyscale cluster). VeloCT scaling with ∞ cores shows cubic growth.

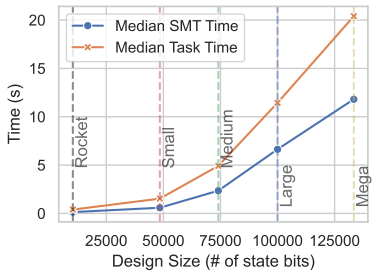


Figure 3.4: Median SMT Query Time and Task Time (in s) vs. Design Size (# of state bits). The time per SMT query scales linearly with the design size.

time up to a point, which provides an estimate of the parallel span (i.e., time given infinite cores). It is noteworthy that the span increases with design size. This indicates that larger designs will benefit from yet additional parallelism. We note that we were able to generate an invariant for Rocketchip in under 10 seconds, representing a roughly 2880× speedup compared to ConjunCT.

Figure 3.3 performs a deeper dive, comparing the time given a fixed 80 cores (on our local cluster) with the time given an "infinite" number of cores (on the Anyscale cluster, which provides more cores on demand). We do not evaluate Rocketchip on Anyscale as 80 cores is sufficient. As expected, time given infinite cores drops relative to the time given 80 cores proportionally with design size. Especially interesting, the time given infinite cores shows that the analysis time scales proportionally to the cubic of the design size.

Median Task and SMT Time / Task. We now explore the extent to which the above results can improve further. The fundamental cost in the analysis is the time to perform SMT queries. Thus, it is important to understand what percentage of CPU cycles are going towards SMT queries. This is shown in Figure 3.4, which compares SMT time to overall task time (i.e., all time spent in the H-Houdini function body). We see that non-SMT activities account for about 50% of the time, even for larger designs, indicating room to improve performance with better engineering. As expected, the time per SMT query grows with design size (which influences the complexity of the average query) and, interestingly, grows linearly with size across the larger designs. We note, the above only shows median SMT query time yet we consistently saw long tails. For example, the 95th and 99th percentile for time-per-task for MegaBOOM was 565.94s and 1000.92s respectively, while the longest task took 9310s.

Total Number of SMT Queries. Orthogonally, we can improve analysis time by reducing

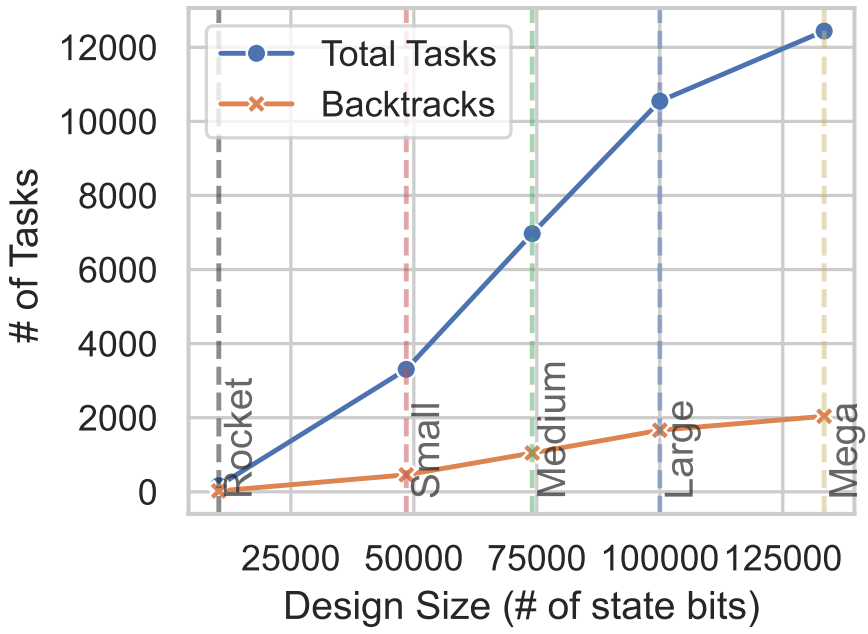


Figure 3.5: # of Tasks and # of Instances of Backtracking vs. Design Size (# of bits). Number of tasks grows linearly with design size. Number of tasks backtracked is relatively low indicating good coverage from positive examples.

the total number of SMT queries that need to be performed. A task performs multiple SMT queries only when backtracking occurs. Thus, we show frequency of backtracking relative to the total number of tasks in Figure 3.5. The main takeaway is that the number of backtracks is relatively small but non-negligible, and accounts for roughly the same % of tasks in each design. If the set of positive examples was exhaustive, the number of backtracks would be 0.

3.7.4 VeloCT: Security Evaluation

We conclude the evaluation with a security evaluation, verifying that the safe set results match prior work.

We show the set of safe instructions that we verified for each target in Table 3.2. To summarize, we verified that most non-memory or control-flow instructions are safe. Our result for Rocketchip matches ConjunCT, except that we found `mul`-family instructions to be unsafe. We verified this manually as correct, and occurs because we evaluated on RV64 (whereas ConjunCT evaluated on RV32). We also monolithically verified the correctness of Rocketchip invariant. Our result for BOOM matches Kunz et al. [57] for all non-FP/non-CSR type instructions (which we categorized manually as unsafe). Interestingly, `mul`-family instructions are safe on BOOM. On BOOM, we were unable to verify the safety of the `auipc` instruction. On talking with the BOOM developers, we found out that even though

Target	Verified Safe Instruction Set
Rocketchip	`add, addi, sub, xor, xori, and, andi, or, ori, sll, slli, srl, srli, sra, srai, lui, auipc, slt, slti, sltu, sltui`
BOOM (All Variants)	`add, addi, sub, xor, xori, and, andi, or, ori, sll, slli, srl, srli, sra, srai, lui, slt, slti, sltu, sltui, mul, mulh, mulhu, mulhsu`

Table 3.2: Safe instruction sets synthesized by VeloCT.

the latency of `auipc` is supposed to be operand-independent, the instruction indeed has variable timing behavior. We leave investigating why this is, and whether an alternate implementation of `auipc` could be made safe, to future work.

3.8 Related Work

Over the years, many different techniques have been proposed to learn invariants: using abstract interpretation [113, 89, 100], Craig's interpolants [5, 4, 145, 112, 118], symbolic execution [51], SAT-based [29, 98, 196, 66, 195, 111, 48, 27, 132, 99, 65, 116], machine learning [188, 230, 190, 88, 243, 187], machine learning-based [87, 82, 157], randomized or enumerative searches [229, 228, 108, 77, 79, 78], and more recently, even LLMs [225, 197, 165, 224, 120, 221]. More broadly, many of the invariant learning work can be viewed as specific instances in the SID [183, 182] framework.

The most successful and widely deployed of these of these are a class of invariant learning algorithms pioneered in IC3 [29, 66] by Bradley et. al. Since then, IC3 has inspired a long line of future work [195, 111, 48, 27, 132, 99, 65, 116] which improved the original algorithm and generalized it to other domains, like software [114, 27] and distributed protocols [142]. IC3 still remains a dominant algorithm in proving properties today, as evidenced by rIC3 [196], an implementation of IC3-based algorithm winning the most recent HWMCC'24 competition [115].

3.8.1 IC3-like Invariant Learning

We first start by briefly describing IC3-like invariant learning algorithm. For a more complete treatment of IC3, please refer to the original paper. At a high-level, IC3 maintains a sequence of frames, where each frame is a conjunction of clauses which represents the set of states allowed by the frame. The algorithm operates in *major* and *minor* iterations: major iterations add new frames while minor iterations refine existing frames. Each frame, F_i, represents an over-approximation of states reachable in i steps from `INIT`.

The algorithm checks if taking a step from the latest frame in the sequence, say F_k, results in a property violation. If it does not, then a new frame F_{k+1} is initialized as P (property) and all the clauses that are inductive from $F_k \rightarrow F_{k+1}$ are propagated to F_{k+1}. If there is a property violation, then a counterexample s is generated. IC3 then uses the counterexample to generate a blocking clause, a disjunction over literals or their negation, to eliminate the counterexample. If there is no such clause, produces a counterexample as a witness to the property violation.

Notice that the negation of the counterexample, $\neg s$, is itself a blocking clause. However, this only eliminates the single counterexample s and not the other related counterexamples. Ideally, IC3 wants a single clause to rule out multiple similar counterexamples, ones that have the same root-cause. To achieve this, IC3 tries to find an *inductive generalization* of $\neg s$, a subset of $\neg s$ which is relatively inductive to the frame F_{k-1}. For better performance, we want the inductive generalization to be as *strong* as possible, i.e., have fewer literals and block out as many bad states. In a nutshell, IC3 repeatedly uses a relative induction SAT query to find the minimal inductive set of $\neg s$. Since the process to find the true minimal set is expensive, IC3 uses heuristics to stop minimization when the inductive subset is deemed *good enough*. This process is then repeated to push inductive generalizations to the higher frames. Finally, at the end of each major iteration, the algorithm propagates clauses forward from $F_i \rightarrow F_{i+1}$ if the clause is inductive. The algorithm terminates with an inductive invariant if $clauses(F_i) = clauses(F_{i+1})$.

At the surface, IC3 (and friends) and H-Houdini may seem similar: they're both incremental algorithms that use *relative inductivity* to learn inductive invariants. But that is where the similarity ends. The core philosophy of the two algorithms are so fundamentally different (discussed below) that the resulting approaches have completely different characteristics:

- *Bit-level vs. Word-level.* IC3 is fundamentally a bit-level model checking algorithm, while H-Houdini operates at the word-level (through predicate abstraction). Word-level invariant learning has several advantages over bit-level: (i) solving word-level problems (SMT) is often more efficient than the corresponding bit-level problem (SAT) as solvers can take advantage of higher level repeated structures through clever heuristics and theory specific reasoning, and (ii) inductive invariants generated at word-level are more interpretable than bit-level as the former maintains common abstractions familiar to experts, e.g., registers and wires, and uses higher level operators, e.g., arithmetic, relational operators. Recognizing these benefits of word-level over bit-level, the research community has made many attempts to *lift* IC3 to a word-level implementation, most notable of which is AVR [98] and IC3IA [47], but, bit-level implementations still win out in performance in many cases (see rIC3 [196] @ HWMCC'24 [115]). Lifting IC3 to an efficient word-level implementation is still an open research challenge.

- *How to add a clause (predicate)?* The philosophy behind IC3 and H-Houdini can be crisply captured by articulating the fundamental query the algorithms use to generate clauses (or predicates):

IC3: *"What single clause can block the root-cause of this particular counterexample?"*	vs.	H-Houdini: *"What conjunction (set) of predicates are necessary to ensure that (target) property holds?"*

In order to do this, IC3 uses a counterexample generated from a failing property or a relative inductivity check, while H-Houdini doesn't require a counterexample. Rather, H-Houdini is based on abduction with a predicate abstraction and uses positive examples to prune out bad guesses early. Alternatively, IC3 eliminates root-causes of failures by adding clauses, one at a time, while H-Houdini eliminates all root-causes at once by adding a conjunction of predicates.

- *Reliance on a good generalization strategy.* IC3's effectiveness is directly tied to finding general blocking clauses that can eliminate *root-causes* of failure. Without a good generalization strategy, IC3 does not scale well as it needs to iterate over many counterexamples all originating from the same root-cause. Indeed, the importance of generalization procedure is well understood by the community as evidenced by a plethora of follow-up work [195, 111, 48, 27, 132, 99, 65, 116], and the author's own experiments in the original work. However, finding the most general inductive clause is expensive and the algorithm resorts to heuristics to make forward progress. The analogous problem in H-Houdini is using a conjunction of predicates that is too strong as an abduct. As a result, H-Houdini may discover that some predicates are not invariants, forcing a backtrack. However, because of the parallelization and memoization in H-Houdini the cost of exploring a *bad* predicate is mostly hidden away: any work done to analyze the *necessary* set of predicates is memoized and reused, only analyzing the new predicates in the next solution. Lastly, backtracking can be significantly reduced by generating better and more positive examples: a process that is a lot simpler than designing a robust generalization algorithm.

- *Parallelism.* The core of IC3 is to arrive at an inductive invariant by eliminating all counterexamples. At each check, the SAT solver may return any one of the possible counterexamples. The counterexample returned by the SAT solver, can to a certain extent, be influenced by the random seed to the solver. IC3 exploits this fact to instantiate many processes, each with a different seed for the SAT-solver, in the hope that each IC3 instance will discover counterexamples and clauses arising from different root-causes. The processes can then communicate the learned clauses to avoid re-discovering the same counterexamples, allowing parallel progress. However, this strategy of parallelism has two limitations: (i) the IC3 algorithm has no control over if the returned counterexamples are actually different. As a result, many parallel processes may waste efforts in eliminating similar, or even the same, counterexample, and (ii) the communication of the clauses between the processes requires all the processes to proceed in lockstep. This limits the amount of parallelism. H-Houdini on the other hand parallelizes the proof search over the different predicates that are a part of the conjunction. Therefore every process contributes to making forward progress without

any redundant work. Lastly, H-Houdini can also use solver randomness to push the amount of available parallelism even further, e.g., by exploring multiple abducts at the same time. However, in our experience, typically H-Houdini already uses all the available cores, e.g., > 200 cores for large designs like BOOM. Therefore, we haven't yet found the need to explore the above option.

- *Generality vs. Exploiting expert knowledge.* IC3 automatically generates bit-level blocking clauses while H-Houdini requires an expert-defined predicate abstraction. This difference enables IC3 to be a push-button solution to solve general verification problems while H-Houdini requires human-effort per property of interest. This however means that IC3 needs to explore a large invariant search space in order to prove a property, thereby limiting its scalability to large designs and properties. Making H-Houdini push-button, while maintaining scalability, is an important and interesting direction of future research.

- *Number and types of SAT queries.* IC3 performs many more and many different types of SAT queries when compared to H-Houdini which performs only a single type of SMT query. Broadly, IC3 has three different types of SAT queries: (i) property check, (ii) forward relative inductivity checks, and (iii) relative inductivity checks for blocking clauses. All three of these checks are relatively inexpensive. However, the sheer number of checks performed, particularly of type (ii) and (iii), quickly add up: multiple type (iii) checks are required to find an inductive generalization for a counterexample. In contrast, H-Houdini does not require any forward propagation checks and only has one type of check, that is most similar to (iii) which is only queried once per target predicate.

3.8.2 MLIS and ICE Learning

MLIS techniques [40, 243, 87, 73, 82, 157, 77, 67] have a long history in invariant learning, starting with DAIKON [67] to find likely invariants from traces and Houdini [82] which is an algorithm to find inductive invariants for unannotated programs. The MLIS setting has strong connections to synthesis in other domains [11, 193, 194]. Garg et al. [87] first introduced ICE-learning which was later generalized and successfully applied to learn invariants in various settings [73, 40]. The main drawback with ICE-learning techniques is the need for monolithic inductive queries, which do not scale well with a large predicate language and to large systems under verification like BOOM. H-Houdini solves this using relative inductive queries, making it a RICE-learner (*Relative* ICE) as introduced by Vizel et al. [211]. To the best of our knowledge, H-Houdini is the first MLIS-based incremental invariant learning algorithm developed in the RICE framework.

Counterexample/Oracle Guided Inductive Synthesis. More generally, MLIS is a special-case of a much broader class of learning technique called Counterexample Guided Inductive Synthesis (CEGIS) [194]. The key characteristic in CEGIS is the interaction between

a learner, which proposes concepts, and a teacher (or oracle) which provides examples/counterexamples. Beyond invariant synthesis, CEGIS has been a centerpiece in many other works, e.g., program synthesis [194]. Oracle Guided Synthesis (OGIS) [117], is a generalization of CEGIS which allows more general interfaces between the teacher and the learner. As shown in [167], ICE learning can be cast as an instance of OGIS.

3.8.3 Invariant & Specification Mining

There's a long line of research which try to infer specifications of systems, particularly in the software engineering community. These methods try to guess *specifications* or *likely invariants* using static analysis [169, 135, 140], dynamic analysis [67, 109, 85, 110, 227, 136, 56], machine learning [13, 129, 220, 189], or a combination of these techniques [208, 222]. Although not the focus of this work, this line of research is highly synergistic with H-Houdini (or other MLIS): the mined likely invariants and specifications can be directly used as a substitute for the predicate language. We refer readers to [134] which does a deeper survey into specification mining techniques across several domains and applications.

3.8.4 Abductive Reasoning in Program Analysis

Abductive reasoning, introduced by Pierce [166], has been applied in program analysis to infer missing preconditions [91] and to scale shape analysis for large software through bi-abduction [33, 32]. Dillig et al. [59] used it to synthesize inductive invariants for loops, but their approach lacks completeness guarantees, cannot exploit hierarchy or parallelism, and requires extensive backtracking. In contrast, H-Houdini offers strong completeness guarantees, utilizes hierarchical and parallel processing and reduces backtracking by leveraging positive examples.

Hardware Constant-Time Verification. Several recent projects focus on verifying constant-time hardware [93, 205, 75, 57, 58, 76] and constant-time contracts [219, 198, 61]. But, none of these existing works scale to large OoO cores such as BOOM—which is a major contribution of our work.

3.9 Discussion & Conclusion

To conclude, this work proposed H-Houdini: a novel invariant learning algorithm that combines the best of MLIS and SAT-based invariant learning approaches to scale invariant learning to large systems. We manifest H-Houdini as the analysis VeloCT, which demonstrates scalable invariant learning for the safe instruction set problem (SISP). VeloCT demonstrates, for the first time, that we can scale invariant learning to large hardware designs such as BOOM.

While we expect H-Houdini to generalize beyond the SISP, more research is needed to make it completely "push button" and capable of verifying arbitrary properties. We see

three synergistic categories of future work. First, work that further improves automation. That is, while H-HOUDINI can efficiently and automatically search for an invariant, it requires the user to specify a predicate abstraction, implement $\mathcal{O}_{mine}$ and specify a means to create positive examples. Nominally, these steps should also be automated. Second, work that demonstrates H-HOUDINI on verification tasks beyond the SISP. We are confident that H-HOUDINI will apply largely "out of the box" to other 2-safety properties, such as other forms of traditional non-interference. Beyond 2-safety, developing extensions to N-safety properties (common in security verification), liveness properties, LTL-style properties and functional correctness checking are all exciting directions for future study. Third, integrating H-HOUDINI as an invariant learning backend in existing verification frameworks, e.g., UCLID5 [168, 184] or pono [144], will allow the community to easily specify new problems and experiment with H-HOUDINI on a broader set of verification problems.

3.10 H-Houdini Analysis

We now provide proof sketches for soundness and completeness for H-HOUDINI.

3.10.1 Contracts

First, we require that any implementation of H-HOUDINI satisfy the following contracts, given a predicate universe $\mathbb{P}$. First construct $\mathbb{P}^+ = \{p \mid p \in \mathbb{P} \text{ and } \forall_{e \in \mathcal{E}}\ e \models p\}$, the set of all predicates consistent with $\mathcal{E}$.

Contract 1. *(Slicing and mining oracle completeness.)* Consider a p_{target}. Let $\mathbb{A} = \{\mathbf{A} \mid \mathcal{O}_{abduct}(p_{target}, \mathbb{P}^+) = \mathbf{A}\}$ be the set of *all* possible abducts for p_{target}. Further assume each $\mathbf{A}$ is minimal.

- Consider $\mathbb{V}_{slice} = \mathcal{O}^{TS}_{slice}(p_{target})$ on line 9. Construct $\mathbb{V}^* = \{v \in \mathbb{V} \mid \exists \mathbf{A} \in \mathbb{A}\ \exists\ p \in \mathbf{A}\ p \text{ is over } v\}$. We require $\mathbb{V}^* \subseteq \mathbb{V}_{slice}$.
- Consider $\mathbb{P}_{\mathbb{V}} = \mathcal{O}^{\mathcal{A},\mathcal{E}}_{mine}(p_{target}, \mathbb{V}_{slice})$ on line 10, where $\mathbb{V}_{slice}$ is given by $\mathcal{O}_{slice}$. We require $\forall\ \mathbf{A} \in \mathbb{A} : \mathbf{A} \subseteq \mathbb{P}_{\mathbb{V}}$.

Contract 2. *(Consistency with positive examples.)* Consider $\mathbb{P}_{\mathbb{V}} = \mathcal{O}^{\mathcal{A},\mathcal{E}}_{mine}(p_{target}, \mathbb{V}_{slice})$ on line 10. We require $\mathbb{P}_{\mathbb{V}} \subseteq \mathbb{P}^+$.

3.10.2 Soundness

We argue that H-HOUDINI is a concrete instance of the incremental learner from Section 3.4.1. The initial property we aim to prove, $\mathbf{H}_0$, serves as the starting point. Each incremental step $\mathbf{H}_i$, synthesized as an abduct, is formed by a conjunction of a subset of predicates from $\mathbb{P}_{\mathbb{V}}$. That is, let $\mathbf{H}_i = \bigwedge_k p_k$. Let $\mathbf{A}^k_i$ be the result of $\mathcal{O}_{abduct}$ on p_k. Then,

by Contract 1, $\mathbf{H}_{i+1} = \bigwedge_k \mathbf{A}_i^k$, formed by taking conjunctions over abducts for p_k. Finally, by Contract 2, H-HOUDINI satisfies the premise for soundness (P-S).

3.10.3 Completeness

We argue that H-HOUDINI is complete with respect to $\mathbb{P}$. That is, if an invariant $\mathbf{H}$ exists in $\mathbb{P}$, H-HOUDINI will find $\mathbf{H}$. This follows from Contract 1, which stems from our focus on 1-step relative induction.

H-HOUDINI will only fail to synthesize an inductive invariant if $\mathcal{O}_{abduct}$ returns `None` for a given property/predicate p_{target}. Thus, H-HOUDINI will return `None` for the top-level property only if the $\mathcal{O}_{abduct}$ query for that property fails. This can occur in two cases: either during the initial synthesis for the top-level property or during re-synthesis after a string of recursive failures (backtracking). Since $\mathcal{O}_{abduct}$ is complete, H-HOUDINI will fail only if no invariant exists within $\mathbb{P}$. Therefore, H-HOUDINI is also complete.

3.11 Soundness & Precision of VeloCT

3.11.1 $\mathcal{O}_{mine}$ satisfies H-Houdini Contracts

First, notice that all p added to $\mathbb{P}_\mathbb{V}$ are checked to be consistent with $\mathcal{E}$ (line 2, line 5, line 7, line 15) before adding to the set. Thus, $\mathbb{P}_\mathbb{V}$ satisfies Contract Contract 2 of H-HOUDINI.

We now check compliance to Contract Contract 1. By earlier arguments, any abduct $\mathbf{A} \in \mathbb{A}$ for 1-step induction of p_{target} can, by definition, only have predicates over variables in 1-step COI of variables in p_{target}. $\mathcal{O}_{slice}$ precisely returns the 1-step COI in $\mathbb{V}_{slice}$ and therefore, the first part of Contract Contract 1 holds. The second part of contract follows from first. That is, $\mathcal{O}_{mine}$ generates all possible predicates consistent with $\mathcal{E}$ over $\mathbb{V}_{slice}$, i.e., there is no p over $\mathbb{V}_{slice}$ such that $p \in \mathbb{P}^+$ and $p \notin \mathbb{P}_\mathbb{V}$. Why? because $\mathcal{O}_{mine}$ is such that the set of predicates over $v \in \mathbb{V}_{slice}$ is the same in $\mathbb{P}_\mathbb{V}$ and $\mathbb{P}^+$. From above arguments, $\mathcal{O}_{mine}$ satisfies Contract Contract 1 of H-HOUDINI.

By satisfying both Contract Contract 1 and Contract 2, VELOCT's use of H-HOUDINI is sound and complete.

Finally, we show the precision (Def. 3.5.7) and soundness (Def. 3.5.6) of $\mathbf{H}$ obtained from VELOCT.

3.11.1.1 H is Sound

This is trivially true: $\mathbf{P} = \texttt{Eq}(v_o^\texttt{l}, v_o^\texttt{r})$ is a part of $\mathbf{H}$ and is inductive. Therefore, any state that eventually leads to violating $\mathbf{P}$ will not be allowed by $\mathbf{H}$. This proves $\mathbf{H}$ is sound (Def. 3.5.6).

3.11.1.2 H is Precise

We break this proof up into three parts.

(I): For any instruction allowed by **H**, **H** *allows all its executions, i.e., with any operand value.* Every $e \in \mathcal{E}$ forces the secret values in the register file to be unequal on `l` and `r`. Therefore, we cannot add an `Eq`, or further, `EqConst` or `EqConstSet`, on instruction operand values. Therefore, we cannot restrict an instruction's operand values. (I) follows.
(II): **H** *allows all executions of every safe instruction.* We have at least one example per safe instruction. Therefore, **H** needs to allow at least one execution of every safe instruction to be consistent with $\mathcal{E}$. Further, from (I), **H** will allow all executions from every safe instruction. (II) follows.
(III): **H** *allows compositions of safe instructions.* From (II), as $\mathbf{H} = \bigwedge_i p_i$, every p_i also satisfies (II). Consider p_0, p_1, p_2 over state elements v_0, v_1, v_2 respectively. Further, let the topology be such that v_0 is some composition of v_1, v_2. Now, as p_1, p_2 allow all executions of all safe instructions, v_1 and v_2 are allowed to independently take values that occur in 2 different safe executions. Consequently, for p_0 to be inductive, p_0 should allow v_0 to take any value that occurs in the composition of allowed safe executions. Since p_0 was an arbitrary choice, this argument holds for all p_i. Since every p_i allows all compositions of all safe instructions, (III) follows. Since (III) is equivalent to Def. 3.5.7, **H** is precise.

3.12 H-Houdini Running Example

In this section, we will demonstrate how H-Houdini works for the VeloCT setting using the example circuit shown in Figure 3.6 (a). The figure shows a simplified version of an execute stage in a CPU shown in Figure 3.6. This execute stage consists of 2 functional units, an `ADD` and a `MUL`, operating on 2 32-bit operands $\texttt{Op}_1$ and $\texttt{Op}_2$. The final output from the execute stage is the result in `Res` and a control signal, `Valid`, to indicate when the result is valid. Similarly, each FU outputs a result, $\texttt{Res}_{add/mul}$, and a valid signal, $\texttt{Valid}_{add/mul}$, which is then forwarded to the final result/valid through a MUX.

Description of FU. The operational implementation of the `ADD/MUL` units are shown in verilog-like pseudo code in Figure 3.7. At a high-level, the `ADD` FU computes the result and valid bits in a single cycle. The `MUL` FU implements a classic iterative multiplier that takes 32 cycles to output the result, but, implements a *zero-skip* optimization: if one of the operands is a zero, the multiplier immediately outputs the result (0) in a single cycle. To simplify later discussion, the slice of the circuit used to generate $\texttt{Valid}_{mul}$ is shown visually as a circuit diagram in Figure 3.6 (b-c).

Property to Prove. For this example we will prove the 2-safety property from VeloCT using H-Houdini. We start by considering two identical copies of the circuit shown in Figure 3.6.[7] We will add a superscript of L and R to all the state elements and wires, e.g., $\texttt{Res}^L$ vs. $\texttt{Res}^R$, to differentiate between the copies. The two copies of the circuit are non-interacting and execute independently of each other. Concretely, we will learn an inductive

[7]For experts: this is a miter or a product program construction.

invariant to prove that the two copies of `Valid` are always equal at every cycle, represented as `Eq(Valid)` $\iff$ $\texttt{Valid}^L = \texttt{Valid}^R$.

To make the explanation more intuitive, we will present the algorithm as an interactive exercise between the authors and readers. Along the way we will construct the solution graph Figure 3.8, where each node in the graph represents a property and an edge from node i to node j means that property j requires property i to hold for j to be inductive. The graph is also annotated with callouts (of the form "*i:X*") to indicate the steps. We will refer to these in our explanation below.

Our objective is to prove the property `Eq(Valid)`. For simplicity, we first consider how to ensure this property holds after simulating the circuit for a single cycle. Specifically, if we start from a state where `Eq(Valid)` holds and take one step, can we guarantee that `Eq(Valid)` will also hold in the subsequent state? From the circuit, we observe that the value of `Valid` in the next cycle is determined by the values of `Opcode`, $\texttt{Valid}_{add}$, and $\texttt{Valid}_{mul}$, which constitute the state elements in the 1-step cone-of-influence (COI) of `Valid`.

This leads us to our first *abductive query* (`1-A`): what constraints must be imposed on these state elements to ensure that `Eq(Valid)` holds in the next cycle? A solution to this query is the conjunction $\texttt{Eq(Valid}_{mul}\texttt{)}$ AND $\texttt{Eq(Valid}_{add}\texttt{)}$. Hence, if we start from a state where $\texttt{Valid}_{mul}$ and $\texttt{Valid}_{add}$ are equal, `Eq(Valid)` will always hold after the next state transition.

Next, we extend this reasoning: can we ensure that `Eq(Valid)` holds for two consecutive cycles? This is equivalent to asking: can we ensure $\texttt{Eq(Valid}_{mul}\texttt{)}$ AND $\texttt{Eq(Valid}_{add}\texttt{)}$ hold for one additional cycle. Since this is a conjunction, we can decompose it into two sub problems: one for $\texttt{Eq(Valid}_{mul}\texttt{)}$ and one for $\texttt{Eq(Valid}_{add}\texttt{)}$.

This pattern of abductive reasoning is repeated across queries `2-9A`, allowing us to construct a solution graph (Figure 3.8). We make two remarks. First, `Eq(in_use)` depends on `Eq(count)`, and vice versa. However, this circular dependency does not cause issues because each property is processed only once. The solution for `Eq(count)` derived using `Eq(in_use)` does not require us to rethink the solution for `Eq(in_use)`, as its solution is already known and remains unchanged. Second, more generally, no property needs to be reasoned about twice (aside from backtracking, discussed next) as no solution changes once it is found. For instance, abductive reasoning in `9:A` does not introduce new queries, allowing us to reuse previously memoized results (shown as `10:M`).

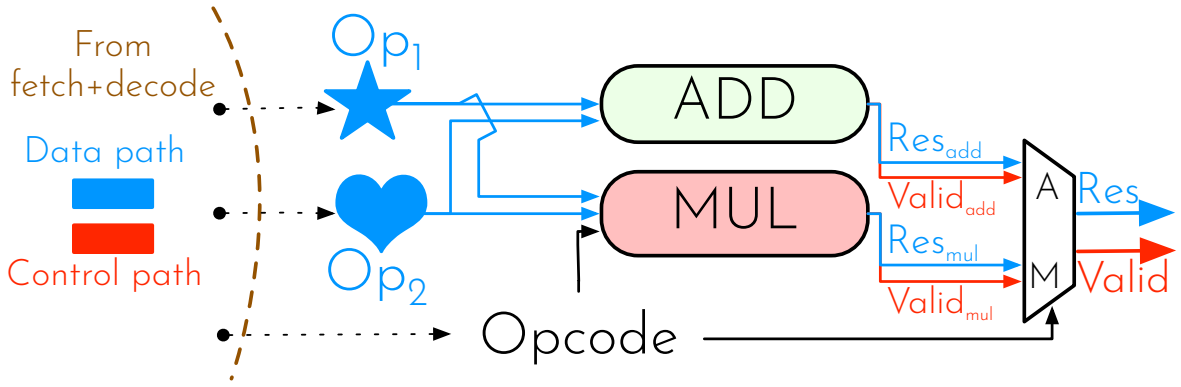
Upon completing our abductive reasoning, we can determine, for each property p, the set of properties that must hold to ensure that p holds in the next step. This gives us a conjunctive set of conditions, ensuring relative inductivity. Let $\mathbf{H} = \wedge p$ represent the conjunction of all required properties. After one step, if all properties remain true in the next state, we have established $\mathbf{H} \implies \mathbf{H}'$. Consequently, we have derived the necessary inductive invariant to prove `Eq(Valid)`.

Backtracking. What if the property $\texttt{Eq(Op}_1\texttt{)}$ failed to be inductive? This would invalidate the solution we have $\texttt{Eq(Valid}_{mul}\texttt{)}$, so we would need to step back, i.e., backtrack, and find other ways to make $\texttt{Eq(Valid}_{mul}\texttt{)}$ hold (`11-B`). Unfortunately, without a way to restrict

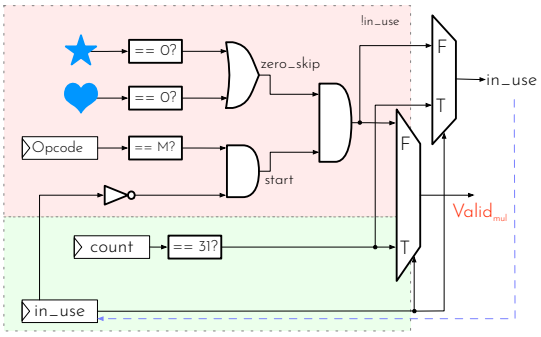
values of $\texttt{Eq(Op}_1\texttt{)}$, we cannot make $\texttt{Eq(Valid}_{mul}\texttt{)}$ hold. Thus, we need to backtrack all the way back to `Eq(Valid)` (`12-B`). Can we find a different way to make `Eq(Valid)` hold in the next step? (`13-A`). Yes! Another solution is to have $\texttt{Eq(Valid}_{add}\texttt{)}$ AND `Opcode := ADD` hold. We can continue to repeat the same abductive inference process `(14-A)` reusing memoized solutions from before (`15-M`) to reduce work.

Parallelism. For the purposes of this example, we walked through each step of the reasoning serially. All of this can be done in parallel. For example, the inferences `2-3:A` are completely independent from `4-9:A` and can be done in parallel to `4-9:A`, while exploiting memoization opportunities to avoid repeated queries (`10:M`).

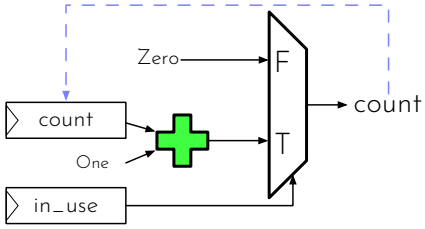
Positive Examples. Lastly, while parallelism and memoization are able to improve performance and reduce slowdown due to backtracking, it would be even better if we could avoid backtracking altogether. Positive examples are concrete execution traces that the invariant must admit. Any property violated when considering such a trace cannot be included in the final invariant and therefore can be eliminated early. For example, a positive example executing ADD can tell us $\texttt{Eq(Valid}_{mul}\texttt{)}$ does not hold, allowing us to correctly infer the second solution and eliminate the backtrack.



[
Running example block diagram.](a) A highly simplified execute stage in a pipeline containing two functional units (FU): ADD and MUL. Each FU takes two operands as inputs and outputs the result in $\texttt{Res}_{add/mul}$ and a valid signal $\texttt{Valid}_{add/mul}$ to indicate when the result is ready. The final result and valid signals are selected through the MUX depending on the current `Opcode`.



[
Slice of MUL FU from example.](b) Slice of MUL FU to compute `in_use` and $\texttt{Valid}_{mul}$. From the figure we deduce that the values of both `in_use` and $\texttt{Valid}_{mul}$ are dependent on $\texttt{Op}_1$, $\texttt{Op}_2$, `Opcode`, `in_use`, and `count`.



[
Slice of MUL FU from example.](c) Slice of MUL FU to compute `count`. From this figure, we can see that `count` is dependent on `in_use` and `count`.

Figure 3.6: High-level diagrams to illustrate H-HOUDINI. (a) shows the high-level circuit diagram for the example. A slice of the implementation of the MUL FU (described in Figure 3.7) is shown in (b) (`in_use` and $\texttt{Valid}_{mul}$) and (c) (`count`). The red region in (b) contains the update expression in the case `in_use` is false, and the green region contains the update expression when `in_use` is true. The dashed purple line indicates the next state update.

```
ADD(Op1, Op2, Opcode) → (Res_add, Valid_add):
  @clock
    if Opcode == ADD:
      Res_add <= Op1 + Op2
      Valid_add <= 1
    else:
      Valid_add <= 0

MUL (Op1, Op2, Opcode) → (Res_mul, Valid_mul):
  start = Opcode == MUL and !in_use
  zero_skip = Op1 == 0 or Op2 == 0

  @clock
    case in_use:
      if multiplier[0] == 1:
        Res_mul <= Res_mul + multiplicand
      multiplicand <= multiplicand << 1
      multiplier <= multiplier >> 1
      count <= count + 1
      if count == 31:  #  Done
        in_use <= 0
        Valid_mul <= 1
    default:  #  Reset
      multiplicand <= zero-extend(Op1, 64)
      multiplier <= Op2
      count <= 0
      Res_mul <= 0
      if start and zero_skip:
        Res_mul <= 0
        Valid_mul <= 1
      elif start:
        in_use <= 1
```

Figure 3.7: Verilog-like description of the ADD and MUL FU. ADD computes on its two operands and outputs the results and valid in a single cycle. The MUL FU implements a classic 32-bit iterative multiplication algorithm with a zero-skip optimization: non-zero operand values take a slow path (32 cycles) to output the result and valid. Otherwise, if either of the operands are zero, the result and valid bits are computed in a single cycle.

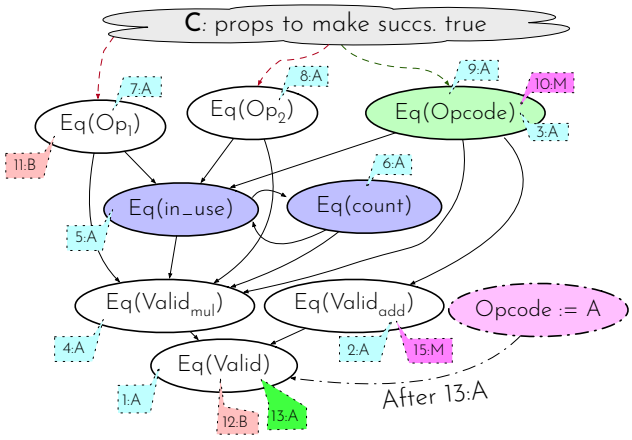


Figure 3.8: Graphical representation of H-HOUDINI invariant learning for example circuit in Figure 3.6. Each node in this graph representation is a predicate, and an edge from predicate i to j denotes that i is required for 1-step inductivity of j. The graph is labeled with callouts of the form *"i-X"*, where i denotes the step number and the X denotes the action: A for abduct, B for backtrack, M for memoization. Hence, the graph is meant to be read in order of the i's.

Chapter 4

SynthCT:
Towards Portable Constant-Time Code

This chapter presents the final component of our framework: a software-level approach to automatically harden code using the security specifications developed and verified in Chapter 2 and Chapter 3. Specifically, SynthCT *is a program synthesis–based framework capable of scaling to large and complex instruction set architectures (ISAs), such as x86-64, and handling instructions known to be unsafe, like* `div`*.* SynthCT *enables developers to write microarchitecture-agnostic code, automatically transforming it into secure, high-performance implementations tailored to specific microarchitectures. This approach provides provable security guarantees, a feature lacking in current practices for developing sensitive code. This chapter addresses Challenges: (C4).*

Recent attacks have demonstrated that modern microarchitectures are fraught with microarchitectural side channels. Constant-time (CT) programming is a software development methodology where programs are carefully written to avoid these channels. In a nutshell, the idea is to only pass secret data to *safe* instructions, i.e., those whose execution creates operand-independent hardware resource usage.

Yet, current CT programming practices have significant security and performance issues. CT code is written and compiled once, but may execute on multiple different microarchitectures. Yet, what instructions are safe vs. unsafe is fundamentally a microarchitecture-specific issue. A new microarchitectural optimization (or vulnerability) may change the set of safe instructions and break CT guarantees.

In this chapter, we develop SynthCT to address the above issues. Given a specification of safe/unsafe instructions, SynthCT automatically synthesizes translations for all unsafe instructions in the ISA using only instructions from the safe set. The synthesized translations can be used as a part of a late-stage compiler pass to generate hardened binaries for a specific microarchitecture. This closes the security hole as the specification, and hence the safe translations, can target each microarchitecture individually. This also allows CT code

to reclaim some performance, e.g., use more complex/higher-performing instructions, when they are deemed safe for a specific microarchitecture.

Using the techniques we develop in SynthCT, we are able to synthesize translations for a majority of the x86_64 ISA. Specifically, SynthCT is able to generate safe translations for 75% of the ISA using only the remaining 25% of the ISA. Interestingly, the majority of the instructions that SynthCT was unable to generate translations for are instructions that experts believe are safe instructions on today's x86_64 microarchitectures.

4.1 Introduction

Microarchitectural side-channel attacks are a major privacy threat. By observing hardware resource utilization from myriad sources—virtual memory accesses [226, 214], hardware memory accesses [161, 231], branch predictor usage [1, 72], arithmetic pipeline usage [14, 104, 6], speculative execution [35, 125] and more [163]—an attacker can learn substantial amounts of information about a victim program, such as its memory access pattern, control flow behavior, arithmetic operation operands, etc. Given the large number of hardware resources that leak different types of information, it is imperative to develop comprehensive software defenses.

Such comprehensive software-only protection against microarchitectural side channels is achieved through constant-time programming [22, 24, 152, 170, 81, 186, 242, 68, 148, 201, 176, 3, 14, 49, 207, 38]. In this paradigm, security critical applications are carefully written and compiled to prevent private information from conditionally influencing hardware resource usage. In a nutshell, constant-time programming ensures: (i) that the program does not have private data-dependent branches, and (ii) that the program does not have *unsafe* instructions with private data-dependent operands. Here, an instruction is *unsafe* if its execution, e.g., timing/hardware resource usage, depends on its operands. Otherwise, the instruction is *safe*.

However, the current constant-time development methodology has a significant security and performance problem. The fundamental issue is that, much like other software, constant-time code is written and compiled once, and reused for a long time across many different microarchitectures. Yet, what instructions are safe vs. unsafe is fundamentally microarchitecture-specific. If an instruction presumed to be safe at compile time is not safe on some current/future microarchitecture, we get a potential security vulnerability. This is especially worrisome now, as Moore's law's slowing incentivizes microarchitects to adopt more exotic optimizations to get performance, rendering potentially many instructions unsafe [209]. On the flip side, if an instruction presumed to be unsafe at compile time is safe on some current/future microarchitecture, we incur unnecessary performance overhead as being able to utilize more instructions tends to reduce overhead.

Prior work on constant-time programming is complementary and has not addressed this issue. Such work either (i) generates *mostly constant-time code* from a higher-level language, e.g., a compiler-based DSL such as FaCT [37], (ii) or hand-writes constant-time code for

specific critical operations [14]. (By *mostly constant-time*, we mean a program that treats control-flow and memory instructions as unsafe.) Both of the above assume that which instructions are safe does not change across microarchitectures.

In this chapter, we develop SynthCT to address the above security and performance issues. SynthCT enables microarchitects to specify what instructions are safe on a per-microarchitecture basis, called the *safe set*, and uses program synthesis techniques to *automatically* generate translations for every unsafe instruction in a target program to instructions in the safe set. Specifically, SynthCT takes as input: (i) a *mostly constant-time target program* in binary form, (ii) the formal instruction semantics for the target program's ISA, and (iii) a safe-set specification for the microarchitecture the target program is to be run on, and automatically replaces any unsafe instructions in the target program with semantically equivalent translations made up of safe instructions. SynthCT can also improve performance by choosing the best possible set of safe instructions for a given microarchitecure while avoiding applying translations for safe instructions in the target program.

Importantly, SynthCT relies on each microarchitecture publishing which instructions are safe vs. unsafe. While this is not common today, there has been both industry [16] and academic [234, 106] traction in publishing similar specifications, suggesting they may become common practice in the future.

SynthCT builds on state-of-the-art program synthesis techniques and formulates the problem of synthesizing unsafe instructions as a Counter-Example Guided Inductive Synthesis (CEGIS) problem. Yet, scaling CEGIS techniques to modern ISAs such as x86_64, as needed by SynthCT, is non-trivial due to the extremely large synthesis search space.

In SynthCT, we design and implement multiple techniques to restrict the synthesis search space and scale CEGIS to modern ISAs. Specifically, the search space grows primarily as a function of two factors: (i) the number of available instructions in the ISA (e.g., 451 non-floating point/vector instructions, or over 1000 instruction variants total, in x86_64) and (ii) the maximum length of allowed synthesized programs. To address (i), we develop *component selection*, a procedure that prunes the ISA down to a manageable subset of instructions which are most likely to result in successful translations for synthesis. The component selection strategy makes use of a key observation: instructions that are semantically similar, and hence, more useful in synthesis, also share a high degree of structural similarity in the instruction's AST. To address (ii), we develop *instruction factorization*, a divide-and-conquer procedure that partitions a complex instruction into smaller pieces, each of which results in a disproportionately smaller synthesis search space. On top of these, we implement multiple optimizations to memoize work and guide synthesis towards more likely solutions earlier—in all cases guaranteeing that translations are semantically equivalent to the input instructions.

Putting it all together, SynthCT synthesizes translations for 75% of x86_64 ISA, i.e., is able to translate a large part of the ISA in terms of a small set of core instructions. Interestingly, the instructions that we do not have translations for are extremely simple and happen to match what experts today believe are safe instructions on microarchitectures running x86_64.

4.2 Contributions

In SynthCT we make the following contributions:

1. We develop SynthCT, the first framework to automatically generate translations from unsafe → safe instructions in an ISA.

2. We develop techniques to overcome several challenges in framing synthesis of unsafe instructions as a CEGIS problem. In specific, we develop techniques to reduce the synthesis search space and make the problem tractable on commercial ISAs.

3. We systematically evaluate synthesized solutions from SynthCT for both performance and security. In our evaluation, we find that SynthCT is able to generate safe translations for a majority (75%) of the ISA using only a small set of core instructions (the remaining 25%). In case studies, we show how SynthCT is capable of synthesizing safe translations for complex unsafe instructions, e.g., the divide instruction (`DIVL-R32`).

We believe SynthCT advances the state-of-the-art in constant-time programming. SynthCT is a robust, modular framework which is capable of being re-targeted to other microarchitectures and ISAs with minimal engineering effort. We have open-sourced SynthCT as two repositories:

- SynthCT: https://github.com/FPSG-UIUC/synthCT
- Synthesized translations: https://github.com/FPSG-UIUC/synthCT-artifacts

4.3 Threat Model and Scope

We consider a standard threat model for constant-time programming, where a victim program runs on a shared machine in the presence of adversarial software. The adversary's goal is to learn private data in the victim program through microarchitectural side channels. (This is equivalent to the SGX adversary [206, 234, 242, 170], modulo a caveat stated below in "Non-goals".) The program itself is considered public. We trust that, for a given microarchitecture, every instruction marked safe in the safe set specification executes with operand-independent hardware resource usage.

4.3.0.1 Security/Functionality goal

Given a mostly constant-time program P that takes input y and safe set S (a subset of the ISA), our goal is to automatically construct a program P' with the following properties:

1. *Functionality:* We require that $P(y) = P'(y) \ \ \forall y$.

2. *Security:* We require that for each instruction $I \in P'$, we have $I \in S$.[1]

By definition of our threat model, satisfying Property 1 implies non-interference with respect to the program input and the microarchitectural side channel attacker, on a non-speculative microarchitecture. That is, it implies $\forall y, y'$ $\mathsf{View}(P', y; \mathsf{uArch}_S) = \mathsf{View}(P', y'; \mathsf{uArch}_S)$ where $\mathsf{View}(.)$ returns the program's hardware resource usage trace in space and time when run on non-speculative microarchitecture uArch_S with safe set S.

4.3.0.2 Complementary: Spectre mitigations for constant-time programming

Speculative execution attacks (e.g., Spectre [125]) use a program's *transient execution* to form *accessors* and *transmitters* that access and transmit (leak) a secret [124]. Analyzing what accessors can be constructed for a given program and microarchitecture, i.e., analyzing transient reachability, is out of our scope and is covered in complementary work [160, 107, 41, 212]. Analyzing what transmitters are possible for a given microarchitecture is in scope: 'transmitter' is a synonym for 'unsafe instruction.' In other words, SynthCT enables complementary works [207, 38] to produce microarchitecture-specific Spectre hardened code, taking into account that machine's safe set.

4.3.0.3 Non-goals/limitations

Physical side channels (e.g., power [127] or EM [156]) are out of scope. Similar to other constant-time defenses, we treat SGX-based attacks that monitor analog information, e.g., the RAPL interface [138], as out of scope.

We also do not prevent secrets from leaking through a program's control flow and memory access pattern. That is, we do not block leakage through cache/memory- and control flow-related side channels [1, 161, 231]. We assume the input program is already hardened against these attacks, i.e., is *mostly constant time*. We elaborate on how microarchitecture can *still* undermine the security of mostly constant-time code in Section 4.4. Writing and generating mostly constant-time code is a complementary concern, and has become practical due to a large body of prior work [22, 24, 152, 170, 81, 186, 242, 68, 148, 201, 176, 3, 14, 49, 37, 139].

4.4 What Instructions May Be Unsafe?

Before describing SynthCT, it is important to understand why instructions become unsafe, and which instructions are likely to flip between safe and unsafe, depending on the microarchitecture. To start, *mostly constant-time code* (Section 4.1) assumes that control-flow and memory instructions are unsafe. Due to their significant influence on program

[1] Requiring that all instructions are safe is overly strong. More precisely, constant-time programming requires that unsafe instruction operands are not a function of private data. Accommodating programs in this relaxed definition would require a complementary, well-understood analysis that can easily be applied on top of SynthCT.

execution, it is 'safe' to assume that control-flow instructions are always unsafe. Likewise, due to myriad memory-system optimizations such as caches, which enable cache-based side channels [161, 231], it is safe to assume that memory instructions (loads, stores) are also unsafe on performance-competitive processors.

This paper's focus is therefore non control-flow and non memory operations. Prior work has studied how a small number of "complex" arithmetic instructions, such as floating point [14], divide [49] and multiply [104] are unsafe ("variable time") on certain microarchitectures. Given tools today, the only fix is to assume they are unsafe on all microarchitectures. This is secure, but overly conservative. SynthCT can improve performance when code is run on microarchitectures where such instructions are safe.

Worse, there is a large family of ISA-invisible microarchitectural optimizations that may render a significantly larger set of arithmetic instructions unsafe [209]. For example:

• Computation simplification / elimination optimizations (e.g., [172, 233, 19]) have been proposed for many arithmetic operations to take advantage of operation-specific identity and absorption properties. For example, that $x \,\&\, 1 = x$.

• Computation reuse optimizations (e.g., [192, 191, 151]) memoize computation when the same instruction(s) are executed twice with the same operands.

• Value prediction (e.g., [149, 175, 185]) saves cycles when an instruction returns a predictable result.

• Significance compression (e.g., [30, 34, 213]), related to serial computation, impacts performance depending on the position of the high-order on bit in each program word.

These optimizations are a major security concern. They can be implemented on any microarchitecture at any time.[2] Further, they are often implemented on a per-instruction basis as they often require instruction-specific logic (e.g., the identity rule for AND is different than that of OR). This makes it unlikely that which are safe will be consistent across microarchitectures. SynthCT can improve security when code is run on microarchitectures where such instructions are unsafe.

4.5 Design Overview

In this section, we present an overview of SynthCT. Figure 4.1 shows an overview of SynthCT's workflow. At a high-level, SynthCT takes as input an ISA specfication and microarchitectural safe-set specification and generates a library of safe translations that implement every instruction in the ISA using only the instructions that are safe on that specific microarchitecture. The generated library may then be used to secure mostly constant-time code (or binaries) using a late-stage compiler pass (or a binary-rewriting mechanism). The final output is a binary that is constant time (with respect to that microarchitecture). Therefore, SynthCT operates in three distinct phases: an offline synthesis phase, a microarchitecture targeting phase, and an online deployment phase. Now we discuss each step and artifacts in the workflow in more detail.

[2]Some indeed have been already. For example, [104] is an implementation of significance compression.

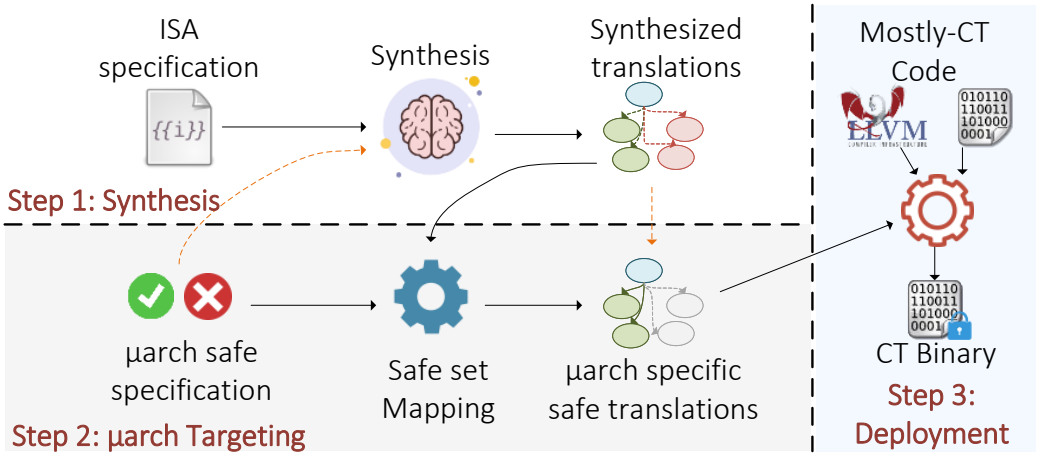


Figure 4.1: Overview of SynthCT. SynthCT takes as input the specification of instructions in an ISA and uses program synthesis to generate a library of translations. When targeting a particular microarchitecture, SynthCT uses a safe set specification for the microarchitecture to generate a library of *safe* translations for all unsafe instructions. The set of safe translations can then be used at deployment time to secure mostly CT source code or binaries through a compiler or binary rewriting pass. Parts of the main workflow are shown with filled lines and a possible alternate workflow is indicated with dashed lines.

4.5.0.1 ISA Semantics

SynthCT takes as input the semantics for each instruction in the target ISA. The semantics describe the precise functional operation of each instruction in some intermediate representation. In our current implementation, SynthCT takes x86_64 semantics written in the K-framework [53], but the framework is conceptually agnostic to which ISA and semantics IR. The semantics for each instruction describes how the output registers and the flag states are computed. The semantics can be reduced to an abstract syntax tree (AST) representation that uses the K-language (/opcodes). Therefore, there is one AST per output register and flag that the instruction sets. An example of such an AST is shown in Figure 4.3 for the instruction "Add with Carry" (`ADCQ`) and the output register. Similar ASTs describe how each of the other x86_64 flags are computed by the instruction. Note that the semantics may not describe *how* the instructions are implemented in hardware nor do they indicate what instructions may be safe. These are simply functional specifications.

4.5.0.2 Safe/Unsafe Specification

SynthCT also takes as input a microarchitecture-specific input that specifies the set of safe and unsafe instructions for that particular microarchitecture. Safe sets may be provided by the hardware manufacturer, or reverse engineered like unofficial currently-used safe sets [14].

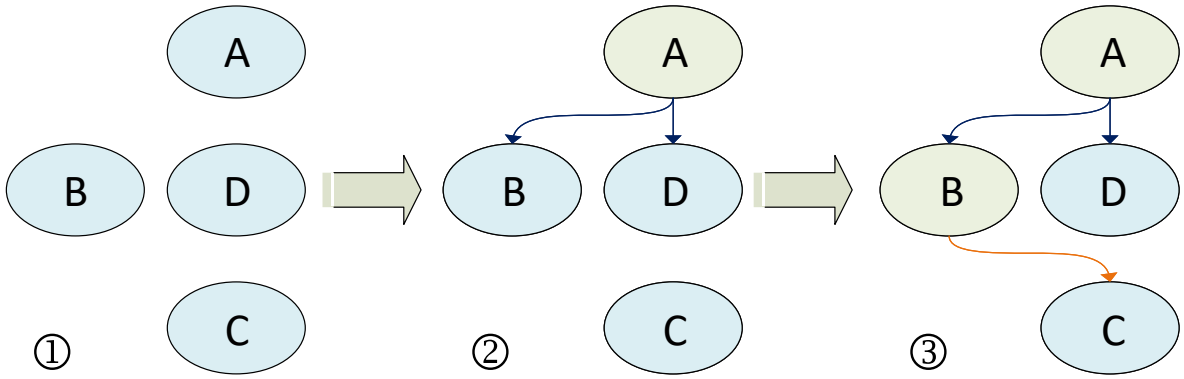


Figure 4.2: Iterative Synthesis. SynthCT performs synthesis iteratively and updates the synthesis graph with generated translations. ① shows the initial synthesis graph with instructions A, B, C, D having no translations. In ② and ③ SynthCT synthesizes translations for A and B and updates the graph.

We hope, long term, that this and related work [106] encourages processor manufacturers to publish formal, explicit microarchitecture-specific safe sets.

4.5.1 Step 1 (Offline): Synthesis

The synthesis takes as input the semantics for all instructions in the ISA and generates a library of translations. In the first stage, SynthCT does not assume a specific safe set and instead tries to find translations from any instruction into any other instruction(s). Therefore, the synthesized set of translations is microarchitecture agnostic. Alternatively, if the target microarchitecture and its safe set are known at synthesis time, they may be used in the synthesis step to generate microarchitecture-specific translations. We discuss this alternate work flow at the end of this section.

4.5.1.1 Iterative Synthesis

The synthesis process discussed above is iterative. SynthCT constructs a graph using all discovered translations. Such a *synthesis graph* is shown in Figure 4.2. In Figure 4.2, SynthCT starts of with a graph containing a node for every instruction in the ISA and no edges (①). As SynthCT synthesizes solutions, e.g., in ② instruction A is synthesized using instructions B and D, edges are added to the graph. Finally, as shown in ③, SynthCT synthesizes a solution for instruction B using instruction C, and therefore, instruction A may also be translated using instruction C and D. This allows SynthCT to discover translations that may otherwise not be synthesizable in a single step due to the complexity of the solution.

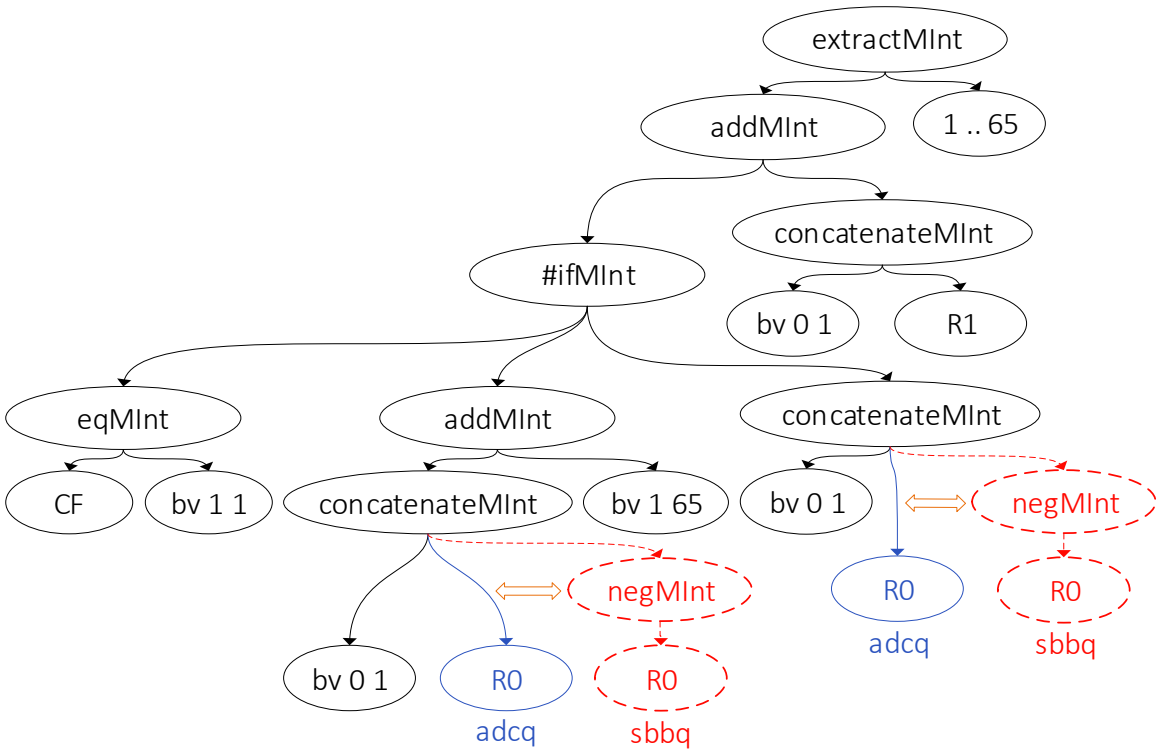


Figure 4.3: Semantics AST for the `adcq` instruction's output register (black and blue nodes in the AST), written in K [53]. The figure also shows the structural similarity between `ADCQ` and `SBBQ`: to derive `SBBQ` from `ADCQ`, one replaces the blue nodes with the red nodes.

4.5.2 Step 2 (Offline): Safe-set Mapping

SynthCT then uses the microarchitecture-agnostic set of translations and the safe-set specification for a given microarchitecture and generates the set of translations specific to that microarchitecture, i.e., every unsafe instruction in the ISA has translations using only the instructions from the safe set. This step ensures that every instruction is expressed with the best possible set of safe instructions, according to some performance model. For safe instructions this is trivial as they need not be translated. For unsafe instructions it boils down to using the best performing synthesized solution that uses only safe instructions. SynthCT uses program length as a proxy for performance, but a lower-level, more accurate and microarchitecture-specific performance model may also be used for selecting the best synthesized solutions. The end result from this step is a library of translations that is specialized to harden software for a specific microarchitecture.

4.5.3 Step 3 (Online/Compile time): Software Hardening / Deployment

During deployment, the library of translations generated by SynthCT is used to secure *mostly* constant-time code.[3] We envision this step to be integrated as a late-stage compiler pass in popular compilers, e.g., GCC or LLVM, to replace any unsafe instructions selected by the compiler backend by their safe translations. Optionally, this may also be implemented as a post-compilation pass using binary rewriting to secure mostly constant-time binaries, e.g., binaries produced by DSLs like FaCT or handwritten to be constant time [14, 37, 22, 24].

The key component of SynthCT is Step 1, Synthesis. In the sections that follow we will first frame the synthesis problem as a CEGIS problem, then, highlight several challenges in applying the CEGIS formulation directly, and finally, present techniques that SynthCT implements to solve these challenges. The deployment step can be implemented with minimal additional engineering effort, e.g., writing a compiler pass that uses synthesized translations stored in a machine-readable format.

4.5.4 Alternate Workflow

If the safe set specification and the target microarchitecture is known at synthesis time, SynthCT can take such a specification as input in step 1, shown by a dashed line in Figure 4.1, and directly synthesize safe translations that are specific to the microarchitecture.

In the following section, we highlight several challenges facing both workflows. To summarize these challenges: On the one hand, if the size of the safe set is large, then both workflows face similar scalability issues. On the other hand, if the size of the safe set is small and simple, then the more complex instructions in the ISA may not be synthesizable in one-shot.

4.6 SynthCT: Formulation and Challenges

This section describes the design and implementation of SynthCT. First, we formulate the problem of instruction synthesis as a Counter Example Guided Inductive Synthesis (CEGIS) problem in Section 4.6.1. Then, we show several challenges that we need to solve to make synthesis tractable for an an ISA like x86_64 in Section 4.6.2. Finally, in the sections that follow, we discuss several techniques we develop in SynthCT to tackle these challenges.

[3] That is, code whose control flow and access pattern to data memory is independent of private data, but may pass private data to unsafe instructions.

4.6.1 Formulation as a CEGIS Problem

4.6.1.1 CEGIS Sketch

We formulate the problem of instruction synthesis to an off-the-shelf CEGIS tool by specifying: (i) a specification of the synthesis goal and (ii) the sketch used to generate candidate programs. In our setting, the specification is the semantics of the *target instruction*, I_t, in bitvector logic. Candidate programs that implement the target instruction are generated from the sketch shown below. Our sketch is simple and general: A *program* is expressed as a list of x86 instructions, and each instruction is parameterized by zero or more register operands (as needed by the instruction).

```
Program (P): (list Inst*)
Inst (I): (choose* {??})
{??}: x86 Instruction { ??: register operands }
```

The {??} are *holes* that need to be filled in by synthesis to generate concrete programs. Here, the first such hole can be filled in by choosing an appropriate *component*[4], i.e., an instruction, from a set of available components, i.e., all instructions or a subset of instructions from the ISA. The second hole can then be filled in by choosing register operands for the selected instruction from the pool of available registers. Based on this sketch, the CEGIS procedure has three degrees of freedom in generating concrete programs: (i) The length of program to synthesize, (ii) The choice of instructions (components) for each "line" in the program, and (iii) The register operands to each instruction.

4.6.1.2 Verifier

The verifier checks if the synthesized program and the specification are semantically equivalent. To do so, the verifier interprets the specification and the candidate program starting from blank (symbolic) states $\mathbf{S}$ and $\mathbf{S}'$ respectively. Then, it performs an equivalence check over the corresponding register and flags states in $\mathbf{S}$ and $\mathbf{S}'$. If the final states are equivalent, then the candidate program implements the specification and we have a synthesis solution. Otherwise, the verifier produces a counterexample that is used to refine the future solutions. In our setting, the verifier uses the SMT solver, Z3 [154], to perform equivalence checks and generate counterexamples.

4.6.1.3 CEGIS Implementation

For synthesis, SYNTHCT uses Rosette [202] a solver-aided DSL that extends Racket to make it easy to develop various tools, e.g., a symbolic interpreter, synthesis engine etc.

In theory, above formulation is sufficient to synthesize programs for all instructions I_t in an ISA in terms of a safe set of instructions. However, in practice, this formulation of

[4] 'Component' is a synthesis term. In this paper a component is an instruction opcode.

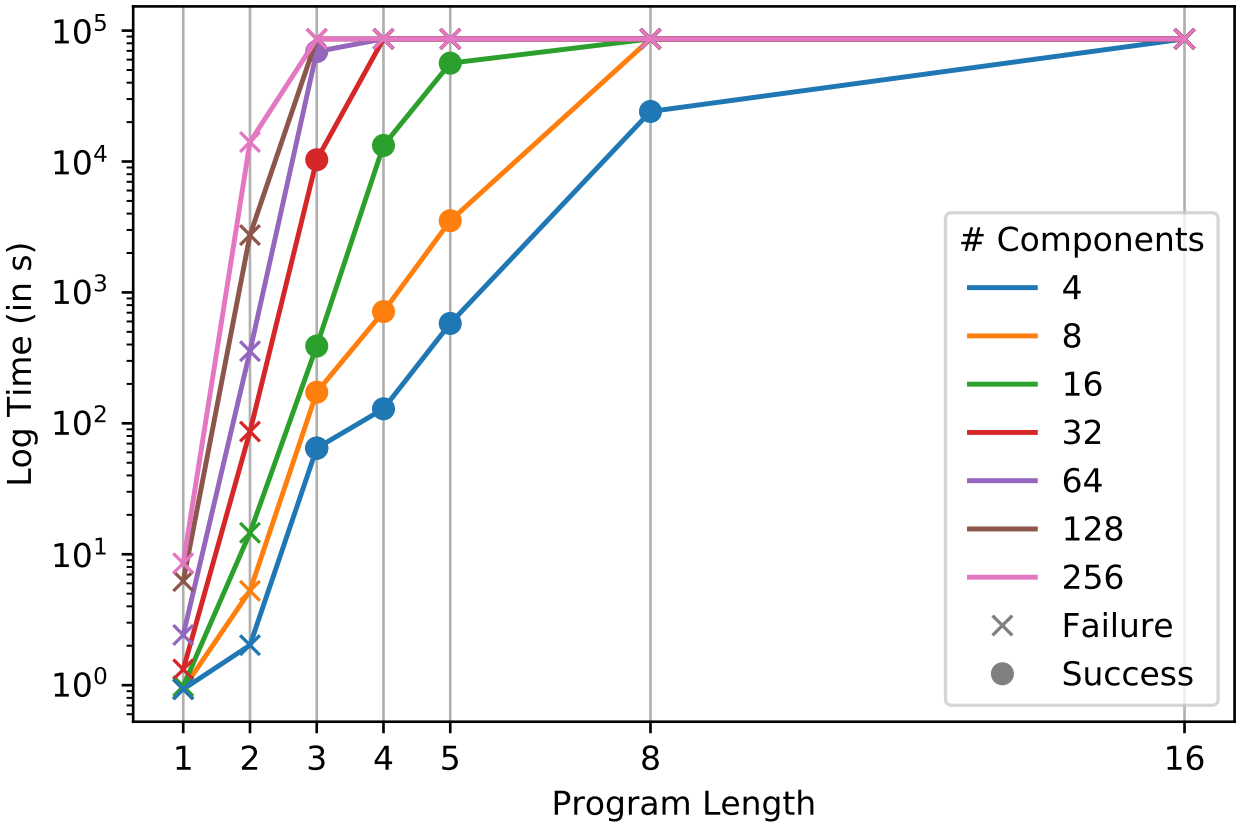


Figure 4.4: Synthesis Scalability. The graph shows synthesis time, for the running example `ADCQ`, vs. program length for different numbers of components. Component sets for synthesis are expert selected such that at least one known solution exists. Points with ● denotes a success and ✕denote failure, i.e., unsat/timeout. Timeout is set to one day for this experiment.

synthesis does not scale to an ISA as large and complex as x86_64 due to the large synthesis search space, as we see below.

4.6.2 Challenges

To test the scalability limits of our synthesis problem formulation, we perform several synthesis experiments by varying the parameters: (i) The length of synthesized programs, (ii) Number of available instructions for synthesis, and (iii) Number of registers available for synthesis. For this experiment, we consider the synthesis of the "Add with Carry" (`ADCQ`) shown previously in Figure 4.3, with six registers available to use in synthesis. Figure 4.4 shows the log of synthesis running time vs. synthesis program lengths for different com-

ponent set sizes.[5] The timeout is set to one day for this experiment. The set of available components are expert chosen to ensure that at least one possible solution exists for `ADCQ`. Points in the parameter space that lead to successful synthesis solutions are marked by a ● while ones that fail, either due to unsat (no possible solution) or a timeout, are marked by an ✕.

From the graph, we see that synthesis time is a function of both synthesis program length and the number of available instructions to use in synthesis. The synthesis time increases exponentially with the increase in both the program length and the number of available instructions. We see that setting the program length to 3 gives us the first successful synthesis and increasing the program length results in exponential increases in the time to success — only the curve with a small number of available components (4) succeeds for program length 8 and all available instruction set sizes fail due to timeout after one day for programs of length 16. We see a similar trend when exploring different-sized sets of available components — increasing the set size exponentially increases the time taken for success while keeping program length constant, with larger set sizes, e.g., 128, having successes only for very short program lengths (3) and the largest set (# of available components = 256) having no successes.

This experiment makes it clear that both parameters have a 'goldilocks zone' in which synthesis succeeds and takes a reasonable amount of time. On the one hand, we require that the program length, # of available components and # available registers is small. On the other hand, if we do not sufficiently provision each of these, synthesis cannot find a solution.

4.6.2.1 Setting the program length

For example, if we make the program length too short, e.g., points in the graph where program length is 1 or 2, then the synthesis fails as there may be no programs of such short lengths that can implement the target instruction. Making matters worse, given an arbitrary target instruction to synthesize, it is not possible to say ahead of time if such a short solution exists. In fact, in x86_64, we see several such instructions, e.g., Leading Zero Count (`LZCNTQ`), that are complex and cannot be synthesized with programs of short length. Thus, we need to solve two problems related to setting program length: a) how to choose the right program length for a given target instruction and b) how to scale synthesis to handle complex instructions that require intractably large program lengths.

4.6.2.2 Setting the component set

When looking at the number of components to make available for synthesis, we face a similar issue. Smaller sets of available components lead to quicker synthesis successes. As

[5]Setting the program length to N instructions or components forces synthesis to find only solutions with that number of instructions. If there exists a solution with fewer than N instructions, say S, synthesis will still need to "find" a solution S' (in a larger search space) that is, e.g., S padded with `nop`s (either literal nops or sequences that are semantically nops) to reach N total instructions.

shown from our experiments, providing the entire ISA to synthesis as choices is infeasible. For example, the synthesis time given a candidate component set size of 256 (256 out of 451 instructions) for programs with lengths > 2 is more than a day. Therefore, the naive solution of using all instructions in the ISA in synthesis is infeasible and we need to develop a way to automatically (without expert guidance) select a good subset.

A strawman idea to address this "component selection" problem is to randomly subsample the ISA. The smaller the size of the subset, the lower the probability of picking a 'good' set of components, that contains the components needed for a solution, while a larger subset will exponentially increase the synthesis time. To check whether this simple strategy is sufficient, consider the probability of choosing a good subset of size 32 (as 32 was the largest component size yielding successes in Figure 4.4). Then there are $\binom{451}{32}$ possible subsets out of which only $\binom{448}{29}$ have the required three components for the solution. Therefore, the probability of picking a good subset is ≈ 0.000326 which is extremely low. Choosing subsets with fewer instructions yields similar results. Therefore, to keep the synthesis time tractable while generating synthesis successes, we need a better way to automatically select component subsets for synthesis based on the target instruction.

4.6.2.3 Setting the number of available registers

In the above experiments, we keep the number of available registers a constant (6). We observe similar scaling trends when varying the number of registers. Due to limitations in the current sketch, synthesized programs cannot spill and restore registers to/from memory. Therefore, having too few registers means that certain solutions that need more registers cannot be synthesized, while having too many registers increases synthesis time.

In this paper we will focus on addressing issues with program length and component sets (see previous two sub-sub sections). Combined with solutions to these issues, we find that the naive strategy for registers—limiting the number of available registers to a smaller number than the actual number of registers in x86_64 (16 general purpose registers)—is sufficient. There may be room to develop heuristics to automatically pick the required number of registers based on the complexity of target instruction or to iteratively increase the number of registers when synthesis fails to produce a solution. We leave developing such heuristics for future work.

4.7 SynthCT Design

From the above experiments, we must address several challenges.

First, we must coordinate the program length with the number of components we actually expect will be required. This is especially when the program length is large, e.g., as we have described with the `LZCNTQ` example, as synthesizing programs of this length is intractable. Second, regardless of the program length, we need to devise a way to sample the ISA so as to

include components that are likely to lead to successful synthesis. We now give a high-level overview of how we solve these challenges:

4.7.0.1 Component selection

As the naive random sampling is unlikely to generate many synthesis successes, we develop a better strategy to pick a good set of components to maximize the likelihood of synthesis success. The key insight behind our component subset selection is that *structural similarity between instruction ASTs can be an indicator for semantic similarity*. We build on this insight to develop our *component selection* strategy in Section 4.7.1.

4.7.0.2 Instruction factorization

As the program length needs to be short, certain complicated instructions in x86, e.g., Leading Zero Count (`LZCNTQ`), cannot be synthesized because no loop-free program of short length can implement such instructions. To solve this, we develop *instruction factorization*, a divide-and-conquer strategy to synthesize solutions in parts, in Section 4.7.2. Sometimes, factorization is insufficient to implement certain complicated instructions, e.g., the Divide instruction. To synthesize such instructions we introduce several additional techniques: *node splitting* and *pseudo-instructions* (Section 4.7.3).

4.7.0.3 Register handling

To reduce the synthesis search space, we restrict the number of registers to just 6 registers, compared to 16 available general purpose registers in x86_64. We found that combined with the solutions discussed above, this simple strategy of restricting the number of registers was sufficient for synthesis. In the future, we may explore more sophisticated strategies to tailor the number of registers based on the synthesis need.

4.7.0.4 Miscellaneous issues

Lastly, we discuss other miscellaneous challenges and solutions in synthesizing x86 instructions in Section 4.7.4.

4.7.1 Component Selection

To make the synthesis problem tractable we need to restrict the synthesis search space by allowing synthesis to only choose components from a small subset of instructions rather than the entire ISA. From our experiments, it is clear that choosing components at random is unlikely to select a good subset of components for synthesis. In this section, we design the component selector, that for a given instruction to synthesize, picks a set of components that maximizes the likelihood of a synthesis success.

Key Insight. The key insight behind our component selection is that instructions that are semantically similar, and hence more useful for synthesis, have *structurally similar* semantics. In other words, their ASTs are structurally similar to each other. For example, in the `ADCQ` example from Figure 4.3, one can form the AST for the "Subtract with Borrow" (`SBBQ`) instruction by replacing the blue nodes in the AST with red nodes. This makes sense as a subtract is addition with a negation. If we give SYNTHCT a set of components that includes `sbbq`, it does synthesize the desired solution:

```
adcq R0, R1:
  movq 0x0, R3
  sbbq-r64-r64 R0, R3
  subq-r64-r64 R3, R1
```

To restate the component selection problem: given an instruction I_t to synthesize, return a list of instructions, or components, from the ISA ranked according to their similarity to the synthesis target I_t. Generally, we find that sub-graphs of the target instruction's AST are sub-graphs of a candidate component's AST. For example, the `ADCQ`↔`SBBQ` example from Figure 4.3. Therefore, we need a fuzzy way to estimate similarity between instructions and assign a score based on how similar the instructions are. Performing subgraph matching on all instruction ASTs is too expensive; we therefore need a lightweight way to estimate the degree of similarity between ASTs.

4.7.1.1 Implementation

SYNTHCT implements the fuzzy similarity estimation in two steps:

1. Graph Embedding: to convert variable-sized graphs, i.e., instruction semantics expressed as an AST, to a fixed-size vector representation,

2. Similarity Estimation: that uses vector representation of ASTs to assign similarity scores.

For graph embedding, we use `graph2vec` [155], an unsupervised machine learning technique similar to `word2vec` [147] used in NLP settings. `graph2vec` captures structural similarity between graphs and generates similar embeddings for graphs that are structurally similar to each other. Our implementation uses `graph2vec` as implemented in the python library karateclub [174]. `graph2vec` takes a number of hyper-parameters for training, we refer the readers to [155] for full details. Empirically, we found the following learning parameters to be sufficient to find good results for synthesis (Section 4.8.5.1): We set the number of WL iterations to 3, the number of training epochs to 30, and the size of the output vectors to 128. We did not try to optimize these hyper-parameters further or do an exhaustive search. We leave optimizing these hyper-parameters as future work.

The output from the graph embedding stage is a fixed-length vector representation of the semantics of each instruction. Then, to find instructions that are most similar to the query

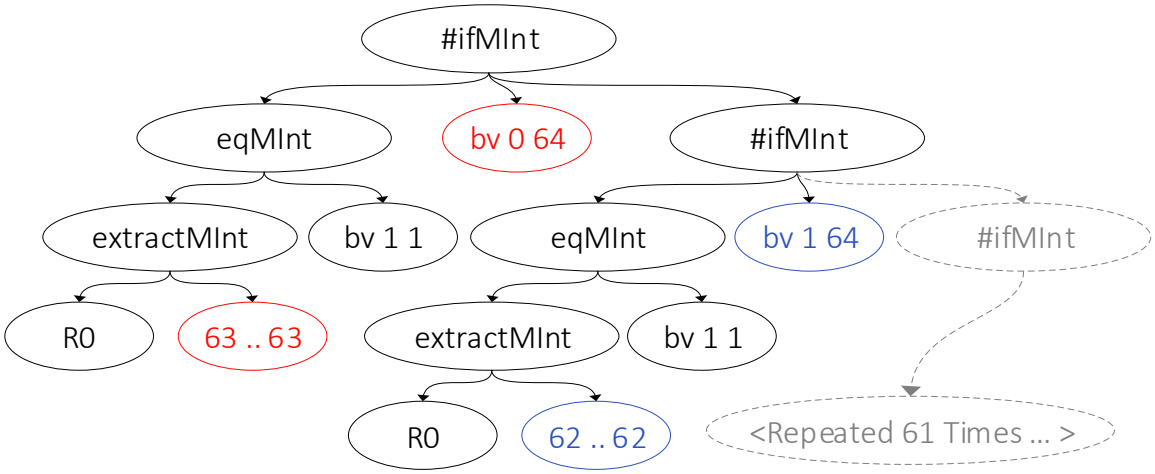


Figure 4.5: Simplified Semantics for Leading Zero Count Instruction (`LZCNTQ`). The figure only shows the first two levels of the AST. Observe that the structure of `if-then-else` is repeated multiple times, but with the conditional comparing different bits and the `then` branch returning a different value. The differences in the two levels are highlighted in red and blue respectively.

instruction we use K-Nearest Neighbors (KNN) search. We use Euclidean distance for the distance metric and set the number of neighbors to 32 ($K = 32$) to get the 32 instructions most similar to the query instruction (as 32 was the largest size that yielded successes in reasonable time; c.f. Section 4.6.2).

4.7.2 Instruction Factorization

In order to keep synthesis times tractable, we are restricted to synthesizing programs of short lengths (e.g., 4 instructions). However, there is no guarantee that *any* instruction from *any* ISA will be synthesizable with a program of such short length. For example, certain instructions in x86_64 such as the leading zero count (`LZCNTQ`) instruction have complex semantics and require hundreds to thousands of simpler instructions to synthesize (Section 4.8.2). We therefore need additional techniques to scale and generalize synthesis.

In Figure 4.5 we show the simplified semantics of `LZCNTQ` to illustrate potential complexity in instruction semantics. The original, un-simplified AST has 771 different nodes with a depth of 64. Taking a closer look, the AST has the same substructure repeated multiple times (as shown in dashed gray lines). Indeed, this is because the same operation is repeated on different bits, i.e., bit 0 through 64, of the operand.

To be able to synthesize such complex instructions while keeping synthesis time tractable, we develop *instruction factorization*, a divide-and-conquer technique that breaks down a

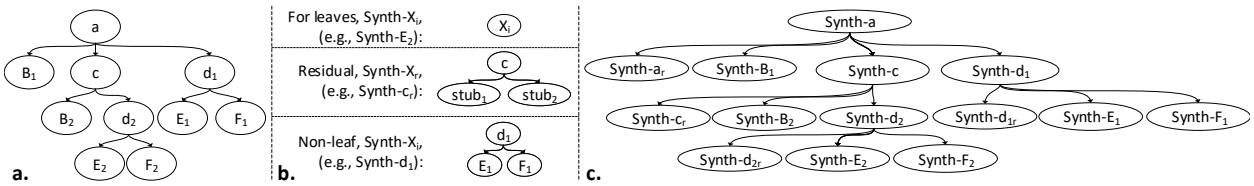


Figure 4.6: Instruction factorization example. (a) An example instruction AST to be factorized. Lower-case labels are AST primitive operations; upper-case labels are subtrees of AST operations. Subscripts denote different instances of the same AST subtree. (b) The different types of synthesis tasks spawned to synthesize the AST. (c) The synthesis workflow graph for (a), depicting the order in which synthesis tasks are evaluated (bottom to top).

complex instruction semantics AST to multiple, simpler ASTs. Each of the smaller ASTs are treated as semantics for simpler *factors*. The factors are then synthesized as separate synthesis tasks, like regular instructions in SynthCT. Once the intermediate factors have solutions, the original larger instruction is synthesized exclusively using these smaller factors.

4.7.2.1 Factorization Strategies

There are multiple ways to *factorize* an instruction's semantics. For a good factorization, we need to find a minimal-size partition of the semantics AST, such that each of the sub-ASTs that are generated by the partition are synthesizable in parts. In general, it is not possible to say if a particular partition is synthesizable without investing time into trying to synthesize solutions from it.

In this work, we primarily use a bottom-up factorization strategy. The intuition is to try the simplest ASTs first, i.e., starting from trying to synthesize sub-trees from the target instruction's semantic's leaves with height = 1, and incrementally combine the simpler solutions to synthesize larger and larger subtrees.

To illustrate this strategy, consider the example in Figure 4.6 that shows an example AST, the generated factor types, and the bottom-up factorization workflow graph. In the AST (Figure 4.6a), nodes are labelled such that a lowercase label (letter) indicates a single node in the original AST and an uppercase label refers to a subtree which may have one or more nodes. When the same node label appears two or more times, that means the same node or subtree appears that many times. We differentiate between occurrences via subscripts.

SynthCT synthesizes the AST by partitioning it into several types of smaller synthesis tasks (Figure 4.6b). To synthesize AST leaves, SynthCT creates a synthesis task to translate just the leaf. As the AST can be recursively viewed as a single node with subtrees for operands, we create a synthesis task for each node and set the subtrees as stubs. We then recursively factorize the subtrees, eventually reaching the leaves, e.g., the subtree E_1 eventually reaches its leaves and their corresponding subtree of tasks are abstracted behind the task Synth–E_1. To synthesize AST non-leaves (e.g., d_1), SynthCT first tries to syn-

thesize the root node (e.g., d_1) in isolation, leaving the children as stubs/residuals (denoted $\mathsf{Synth–d}_{1r}$). It then tries to synthesize the whole rooted AST after both the root node and children have been resolved (denoted $\mathsf{Synth–d}_1$).

Finally, the workflow graph (Figure 4.6c) shows the order of in which SynthCT schedules the synthesis tasks to synthesize the overall AST. SynthCT starts synthesis of factors from the leaves of the workflow graph and proceeds to synthesize the parents when all the subtasks return success. Child tasks beneath a single parent are attempted left-to-right; hence, the residual task is synthesized first, followed by tasks for each child subtree. This way, synthesis works bottom-up, synthesizing the simplest factors first before synthesizing larger, more complicated factors, eventually synthesizing the complete instruction. Additionally, when synthesizing non-leaf factors, the sub-factors of the target factor are available as components to use in synthesis.

4.7.2.2 Optimization

Factorization may create a secondary problem: there may be too many simple, overlapping synthesis tasks to run in a reasonable amount of time. Consider the example from earlier (Figure 4.6) and its corresponding bottom-up workflow graph (in Figure 4.6c). In this example, notice that the subtrees rooted at d_1 and d_2 are equivalent, and yet, we have separate synthesis tasks, Synth-d_1 and Synth-d_2, in the workflow graph. In fact, we see massive memoization opportunities in the wild when synthesizing several x86_64 instructions, e.g., the aforementioned Leading Zero Count (`LZCNT`) shown in Figure 4.5, Population Count (`POPCNT`), Trailing Zero Count (`TZCNT`) etc. Indeed, this is because `LZCNTQ` (and similarly the other examples) performs the same repeated operations on bits 1 through 64 of its operands. To reduce the number of synthesis tasks we exploit this observation and memoize the synthesis results. When a new factor needs to be synthesized, SynthCT first computes the hash of the subtree and checks if a solution has already been computed by another synthesis task.

4.7.3 Node Splitting and Designing Pseudo-Instructions

Instruction factorization solves the problem where the AST is dominated by many vertices, each with relatively simple functionality. We now tackle the opposite problem: where the AST is dominated by few vertices, each with complex functionality, e.g., a divide opcode. These opcodes are not translated in K internally as they have a one-to-one correspondence with opcodes in SMTLIB, thereby enabling direct formal reasoning such as equivalence checking.

The fundamental reason we cannot handle complex intermediate vertices is because SynthCT has no structural information in the semantics to exploit during component selection, i.e., since each individual vertex is complex and abstracts away multiple operations, picking a good set of components to implement the functionality of that said vertex is not possible.

Pseudo-Instruction	Description
movq-imm-r64 i, r0	Move an immediate value `i` into the register `r0`
pmovq-r64-r64 r0, r1	Move from register `r0` to `r1`
pmov-flag-r64 f, r0	Move value of flag `f` into register `r0`
pmov-r64-flag r0, f	Move value in `r0` into flag `f`
psplit-r64-r64 r0, r1, r2	Split value in `r1` into `r0` and `r1` at index specified in `r2`
pconcat-r32-r32 r0, r1, r2	Combine `r0` and `r1` into `r1` by shifting value in `r0` by value in `r2`
pcmov-r64-r64-r64 r0, r1, r2	Conditionally move `r1` into `r2` depending on if the condition `r0` is true
pnot-r64 r0	Logical not
por-r64-r64 r0, r1	Logical or
pand-r64-r64 r0, r1	Logical and
pxor-r64-r64 r0, r1	Logical xor
pnop	No-op (NOP)

Table 4.1: List of pseudo-instructions implemented in SynthCT.

Therefore, SynthCT needs additional assistance in order to be able to exploit structural information and synthesize such complex instructions.

In the following two subsections, we develop two techniques to handle this source of complexity: (i) expert-written pseudo-instructions to provide synthesis with some useful primitives and encourage creative solutions, and (ii) node splitting to manually represent complex intermediate K-operations as simpler operations.

4.7.3.1 Pseudo-Instructions

The intuition behind pseudo-instructions is to have some extremely useful expert-chosen primitives, e.g., generating bitmasks, extract bits `i` through `j` from a register etc., be available in all synthesis tasks. These expert-chosen primitives are treated as normal unsafe instructions and are synthesized by SynthCT as usual. They can then be used as subroutines in subsequent synthesis jobs. Well chosen pseudo-instructions solve three problems: (i) re-discovering implementations for basic primitives wastes synthesis cycles, (ii) by abstracting away useful primitives that need multiple instructions to implement behind a pseudo-instruction, we effectively compress the program lengths of synthesis solutions that need these primitives, and (iii) if chosen correctly, pseudo-instructions will encourage creative solutions that synthesis may not have found otherwise. Pseudo-instructions are ISA independent and therefore can be used in synthesis of instructions in other ISAs as well, although, their usefulness may vary depending on instructions and complexities of the ISA.

The current set of pseudo-instructions we designed for SynthCT along with their descriptions are shown in Table 4.1. Just like regular instructions, pseudo-instructions are synthesized by SynthCT, except that the set of instructions used as components during the synthesis is a set of simple, fixed instructions. These need not be in the safe set for

a specific microarchitecture and, if not, will themselves require translations to the safe set. In other words, the pseudo-instructions do not go through the component selection process. This is currently an engineering limitation[6] which can be addressed in future work.

4.7.3.2 Node Splitting

We use node splitting to simplify complicated K intermediate opcodes, e.g., the K `divide_64` opcode in the division family of instructions, `ROL/ROR` in the rotate family of instructions, and `MUL` in the multiply family of instructions. K-opcodes are finite and shared across ISAs, i.e., instructions from any ISA need to be expressed in terms of these fundamental K-opcodes to be compatible with formal reasoning in the K-framework. Thus, node splitting is a one-time manual implementation effort, per complex K-opcode, and need not be re-done if SynthCT is re-targeted to a different ISA.

Once such translations are implemented in SynthCT, ASTs of instructions that contain the complex opcodes are first simplified by replacing the opcode with the translation before proceeding with the rest of SynthCT's workflow. We implement such translations in SynthCT for the the rotate K-opcode, `rol`, and the K-opcodes used in the divide instruction, `div_quotient_int32` and `div_remainder_int32`. This will be discussed in our evaluation Section 4.8.3. More such translations may be implemented in SynthCT on-demand with minimal effort.

The overall algorithm uses node splitting with instruction factorization and component selection as follows. First SynthCT uses node splitting to simplify complex vertices in an instruction AST using translations built into SynthCT. This process converts an AST with a few complex vertices into an AST with simpler, but larger number of vertices. Second, we use instruction factorization to synthesize the now larger AST through the divide-and-conquer mechanism that we described earlier. We note that due to program length constraints, node splitting would likely be ineffective without factorization.

Pseudo-instructions vs. node splitting. Both pseudo-instructions and node-splitting help SynthCT to synthesize complex instructions, albeit in slightly different ways. Pseudo-instructions are intended to be more generic and help synthesize more creative solutions that would otherwise not be possible, while node-splitting is specific to certain complex K-opcodes that are opaque and need to be simplified.

4.7.4 Other Challenges

Synthesizing x86_64 instructions, in particular, also introduces several additional challenges.

[6] Pseudo-instructions in SynthCT are directly implemented in racket (/rosette) rather than K and therefore cannot go through the usual flow through SynthCT that other instructions written in K can go through. Therefore, with the current prototype implementation we cannot perform the usual component selection described in Section 4.7.1.

4.7.4.1 Synthesizing Multiple Solutions

The earlier sections describe how SynthCT generates a single solution. But to develop a rich set of translations so that they may be applicable to a wide-range of safe sets we ideally need to generate multiple synthesis solutions for a single instruction, each that utilizes minimally- or non-overlapping subsets of components.

To achieve this diversity, on a synthesis success, SynthCT generates multiple new synthesis tasks for the same instruction as a feedback mechanism. Each of the new tasks are generated by removing one of the instructions used in the synthesized solution. To do so, SynthCT uses the same K-Nearest Neighbor (KNN) heuristic developed in Section 4.7.1 to pick the K most similar instruction to the target instruction and then filters out the instruction(s) that were used in the synthesized solution. Therefore, a solution that uses `N` distinct instructions in the solution generates `N` new synthesis tasks as feedback. This sampling-without-replacement strategy encourages diversity by not re-using instructions across solutions.

4.7.4.2 Side-effects: Equivalence of Flags

Many x86 instructions also set ISA flags as side-effects, i.e., `RFLAGS`, in addition to the output registers(s). For example, the running example, the `ADCQ`, instruction sets the following flags: `OF`, `SF`, `ZF`, `AF`, `CF`, and `PF`, according to the result. To achieve complete equivalence to the instruction, not only does the output register need to be equivalent, but the flag state needs to be equivalent as well.

We treat synthesis of flags as a separate synthesis task where the objective is to synthesize these side-effects and ignore the output register. Once a solution that sets the flags is synthesized, it can be combined with the solution for the output register(s) to achieve complete equivalence. Glue code saves register values before the main computation, then, saves the contents of the output registers after the main computation, and restores the initial operand values for the flag computations. Lastly, it restores the saved result from main computation to the output register using a single `MOV` with no side-effects.

Of course, it may be more efficient to synthesize the flags with the output registers in one-shot in a single synthesis task. Yet, this may prevent synthesis from discovering solutions that only implement the computations needed to set the output registers of the target instruction correctly. Such partial solutions may be sufficient most of the time, as described below.

Including code to set the flags correctly introduces additional overhead. However, in many cases, the instruction side effects are ignored, i.e., the flags are never used before being clobbered again. Therefore, we store both solutions separately: the main solution that sets the output registers, and the additional code needed implement side-effects. When translating instructions in *mostly constant-time code*, we choose from one of the two variants depending on if the flags are live at that program point or not. This allows us to reduce

the overhead and not needlessly emulate the instruction side-effects when the flags are never used before being clobbered.

4.7.4.3 Exceptions

Some instructions may generate operand value-dependent exceptions. For example, the divide instruction (`DIVQ`), generates an exception when the divisor is 0. These exceptions are problematic for SynthCT, and constant-time code in general. On the one hand, throwing the exception breaks constant-time programming. On the other hand, masking the exception breaks semantic equivalence (which is an important goal in SynthCT). By default, we can adopt well-established strategies (e.g., [170]) for masking exceptions in constant-time code, but leave implementing these solutions to future work.

4.8 Evaluation

4.8.0.1 Framework

SynthCT is implemented in python in about 8000 lines of code. This code is responsible for parsing semantics, performing AST transformations, factorization, generating individual synthesis tasks, and orchestrating the feedback for the synthesis. Additionally, another 700 lines of code is implemented in racket to set up synthesis, implement a machine model, implement pseudo-instructions and interpret the generated K programs. The implementation uses formal semantics for x86_64 written in K [53]. For synthesis, SynthCT uses Rosette [202] a solver-aided DSL that extends Racket to make it easy to develop various tools, e.g., a symbolic interpreter, synthesis engine etc. Currently, SynthCT only synthesizes instructions with register operands. To handle memory operands, or instructions with immediate operands, we can augment the synthesized translations with an additional `mov` instruction to load (or store) from memory or load immediate operands into registers. Alternatively, the register operands in synthesized translations may be replaced by a memory expression when the target instruction uses a memory operand (assuming the instruction behaves identically when operating on registers v/s memory, albeit performing an additional load/store to memory).

4.8.0.2 Experimental Setup

All experiments below were performed on a server machine running Ubuntu 18.04 within a docker container. The two machines used had 80 cores each, with 128GB/256GB of ram respectively. Synthesis tasks were run in parallel to use as many cores as possible. Synthesis runs were performed in batches with our longest run lasting 3 days. Instructions requiring factorization were run separately in isolation. Over the course of all our experiments, SynthCT generated a total of 2617 synthesis solutions, generating at least one translation

for 242 (/366)[7] non-vector, non-floating point instructions. We did not try synthesizing vector and floating point instructions. While we cannot show all translations for space, we show several representatives in Table 4.5 in the appendix. All results that we discuss below consider full equivalence of translations, i.e., output registers + flags (Section 4.7.4.2).

In the subsections that follow, we evaluate SynthCT for: (i) security, to understand and assess the safe sets resulting from SynthCT's synthesized translations, (ii) performance, to quantify the overhead introduced by using SynthCT's safe translations, (iii) two case-studies to illustrate the use of node-splitting (Section 4.7.3) and factorization (Section 4.7.2) in synthesis of the rotate left (`ROLL`) instruction and the divide (`DIVL-R32`) instruction, and (iv) lastly, highlight some secondary metrics of the synthesis process.

4.8.1 Security Evaluation

In this section, we evaluate the security implications of the generated set of translations. More specifically, we analyze the potential safe sets that are emergent from SynthCT synthesized translations. Then, we compare these emergent safe sets with those introduced in the literature.

4.8.1.1 Methodology: Deriving Safe Sets from the Synthesis Graph

We represent SynthCT's translations as a synthesis graph (Section 4.5). To recap, the synthesis graph (shown in Figure 4.2) has a node for every instruction (and by extension, pseudo-instructions and factors encountered in the synthesis process). For a solution synthesized for a target instruction, I_t, the graph records the instructions used by the solution by adding an edge from instruction I_t to all the instructions used in the solution. We store additional metadata for each edge, e.g., a unique solution, the line number in the synthesized program, and concrete values of operands to use. We iteratively build up the synthesis graph as SynthCT synthesizes translations for all instructions in the ISA.

Once we build a synthesis graph corresponding to a set of translations, the next step is to identify the emergent safe sets. Specifically: we define SynthCT's safe set as those instructions that have no translations, i.e., outgoing edges in the synthesis graph (conversely, instructions that have at least one outgoing edge may not be needed in the safe set). By definition, these instructions cannot be rewritten in terms of any other instruction. Thus, any microarchitecture using SynthCT will require that its microarchitectural safe set be a superset of the SynthCT safe set (more details are given in Section 4.8.1.3).

Deriving the safe sets from the synthesis graph is non-trivial because the synthesis graph may contain cycles. For example, consider synthesis of instructions A and B where solution to A may use instruction B and vice-versa, thereby creating a cycle in the synthesis graph. In this case, neither A nor B are leaves, but at least one of them needs to be included to form the safe set. In other words, the safe set is not unique. Therefore, the first step is to reduce

[7] The remaining instructions (451-366) are all `mov` + `cmov` instructions that we do not run synthesis tasks for.

ADCB*(2)	SUBB*(2)	SUBB*(2)	ADCQ or SBBQ	ADCL or SBBL
ADCW or SBBW	ADDB*(4)	NOTL or ANDL	ROLL or RORL	
CMPXCHGB*(4) or XCHGN*(4) or ANDB*(2) or ORB*(2) or XORB*(2)	ANDB*(2)	SALQ or SHLQ	SHLL or SALL or BLSRL	BSRL or BSFL
BSRQ or BSRW	ORB*(2)	XORB*(2)	NEGB*(2)	INCB*(2)
ROLW or RORW	RCRB or RCLB	RCLW or SHLW or SALW	ROLB*(2) or RORB*(2)	ROLL or RORL
SALB*(2) or SHLB*(2)	SALL or SHLL	SALW or SHLW	SARB*(2)	SET*(9 sets, 60 total)

Table 4.2: Safe set instructions involved in cycles. One instruction from each group needs to be chosen to be included in the final safe set. '*' denotes multiple instruction variants shown in a compressed form. The number of instructions represented/compressed is indicated with parentheses. For example, the "SALB*(2) or SHLB*(2)" group contains 4 instructions; 1 of which must be included in the final safe set.

the synthesis graph to a directed acyclic graph (DAG) by eliminating such cycles in the graph. To do so, we first identify the cycles in the graph by using Tarjan's algorithm [199] to identify the strongly connected components (SCC) in the graph. Next, we replace the nodes in the cycle with a single node. The new node is labeled using a disjunction of node labels (instructions) that form the cycle. We then redirect any incoming edge to the cycle, i.e., an edge from a node not in the cycle to a node that is a part of the cycle, to the new node. Similarly, any outgoing edge from the cycle is redirected from the new node. We apply this procedure multiple times until all cycles in the synthesis graph are eliminated. The leaves of the newly generated DAG representation of the synthesis graph gives us the list of potential safe sets: every leaf represents a single instruction, or a choice between several instructions (represented by the disjunction as a result of replacing cycles in the graph).

4.8.1.2 SynthCT safe set

Using the methodology described in the above subsection, we derive safe sets for SynthCT synthesized translations. Specifically: Table 4.3 represents the core safe set that must be included in all SynthCT safe sets. Table 4.2 shows additional safe set instructions involved in synthesis graph cycles (Section 4.8.1.1). By combining the instructions in Table 4.3 with one instruction per group in Table 4.2, we can form one of many final safe sets. The core safe set contains 53 instructions and we need to select 1 instruction from each of the 37 groups of instructions to form a complete safe set, resulting in a total of 90 instructions in a safe set. Therefore, using only 90 / 366 (25%) of the instructions we can implement the remaining 276 (75%) instructions in the ISA.

4.8.1.3 Comparing to a microarchitecture safe set

The next question we want to answer is: what are the implications of these generated safe sets and how do they compare to a microarchitecture specific safe-set specification? Recall that in the second phase of SynthCT, we use a microarchitecture specific safe-set specification to generate safe translations of all instructions in the ISA. The goal is to find translations

Category	Opcode	B	W	L	Q
Bitwise Operations	NOT	✓	✗	✗	✗
	AND/XOR	✗	✗	✗	✓
Divide and Multiply	DIV/IDIV	✓	✓	✗	✓
	IMUL/MUL	✓	✓	✓	✓
Bit Rotates	RCL/RCR	✓	✓	✓	✓
	ROL	✓	✗	✗	✗
	ROR	✓	✓	✗	✗
Bit Shifts	SAR/SHR	✓	✓	✓	✓
	SARX	-	-	✓	✓
	SHL	✗	✗	✗	✓
	SHLX	-	-	✓	✗
Bit Counting	TZCNT	-	✓	✗	✗
	POPCNT	-	✓	✗	✓
	LZCNT	-	✓	✗	✗
Exchange	XCHG	✗	✗	✓	✗
	CMPXCHG	✗	✓	✓	✓
Miscellaneous	BEXTR	-	-	✗	✓
	BTRW	-	✓	✗	✗
	BZHI	-	-	✗	✓
	BSWAP	-	-	✓	✓
	CMOVE	-	-	✗	✓

Table 4.3: SynthCT core safe set. Rows represent different opcodes. Columns represent bitwidths: **B:** Byte, **W:** Word (2 Bytes), **L:** Long (4 Bytes), and **Q:** Quad Word (8 Bytes). ✓indicates that an instruction variant is a part of the safe set. ✗denotes that the instruction variant is not a part of the safe set. ✓ denotes that the opcode is common to the LibFTFP safe set, while ✓ denotes that the instruction is absent in the LibFTFP safe set. - indicates that the instruction does not operate on the corresponding bitwidth.

add	and	cmp	imul	mov	
movabs	movsd	movsx	movsxd	movzx	mul
neg	not	or	sar	sbb	seta
setae	setbe	sete	setg	setl	setle
setne	shl	shr	sub	test	xor

Table 4.4: Safe Instruction Set from LibFTFP. Control-flow and memory instructions are excluded for clarity. Opcodes present in LibFTFP and not in SynthCT's safe set are highlighted in red. As the table only shows the LibFTFP safe set, opcodes safe in SynthCT but not in LibFTFP are not shown. Opcodes common to both LibFTFP and SynthCT safe sets are highlighted in green. Opcodes with partial overlap, i.e., only some variants of instruction are in SynthCT safe set, are highlighted in orange. Instructions in the mov* and set* family are excluded from coloring as SynthCT has a choice in selecting between some of these variants, but does not necessarily need to contain all. See Table 4.3 to compare the SynthCT safe set against the LibFTFP (this) safe set.

for every instruction in the ISA to instructions in the uarch safe set. Therefore, when given such a safe-set specification, we can have two different scenarios:

Case 1: Microarchitecture safe-set specification and SynthCT safe sets are compatible. In this case, at least one of the SynthCT-generated safe sets is a subset of the microarchitecture safe set. Since, by definition, all instructions in the ISA can be translated using only the instructions in SynthCT's safe set and the microarchitecture safe set is a superset of SynthCT's safe set, every instruction in the ISA can be translated using only the instructions in the microarchitecture safe set. Having more instructions in the microarchitecture safe set simply means more choices of instructions that may be used in the safe translations. This is a success case as now any program compiled for the given ISA can be secured by using the set of safe translations from SynthCT.

Case 2: Microarchitecture safe-set specification and SynthCT safe sets are incompatible. In this case, none of SynthCT's safe sets are subsets of the microarchitecture safe set. Since instructions in the SynthCT safe set are leaves in the synthesis graph, this means there is one or more instruction that cannot be translated to the microarchitectural safe set (namely, the SynthCT safe set instructions that are not contained in the microarchitectural safe set). This is a failure case. Not all instructions (and hence programs) in the ISA can be expressed using the safe set of instructions for that microarchitecture. Further synthesis tasks and expert assistance may be needed to synthesize translations for instructions that are missing in the microarchitecture safe set to make the two sets compatible.

4.8.1.4 Comparison to Prior Work

In this subsection, we compare the SynthCT safe set with safe sets from prior work, e.g., LibFTFP [14]. The list of safe instruction is directly taken from their work and reproduced

in Table 4.4. We have excluded all control-flow and memory instructions, since those are likely to be unsafe across microarchitectures (Section 4.4). Their work does not explicitly list the different variants of instructions, i.e., the different bitwidths of operands, but rather only the opcodes. Therefore, we assume that an opcode encodes all variants of that particular instruction.

The key takeaway is that most instructions that are considered safe in SynthCT are also what LibFTFP (and hence experts in constant-time programming) considers safe, this corresponds to almost all instructions in Table 4.2 and all ✓ in Table 4.3 or alternatively the green + orange + black opcodes in Table 4.4. Notable exceptions to this trend are the instructions in the bit-counting and miscellaneous categories: these are complex x86_64 instructions for which SynthCT was unable for synthesize translations for certain variants of the instruction. This is likely due to strict timeout limits and imperfect component selection during our factorization strategy. We aim to improve this to shrink the safe set further in future work.

That said, SynthCT need not have translations for every variant of every instruction. For example, SynthCT's safe set only needs to include `XORQ` and 1 (out of 4) variants of `XORB` (as the other 3 variants of `XORB` map trivially to the one with a translation). Lastly, there are certain opcodes, e.g., `CMP` and `TEST`, that SynthCT does not include in its safe set as we were able to synthesize translations for all variants of these instructions using other instructions in the safe set.

4.8.2 Performance Evaluation

In this section, we evaluate the performance of translations synthesized by SynthCT. We show the distribution of lengths of synthesized safe translations in Figure 4.7. Note that these are lengths of programs when unsafe instructions are written completely in terms of the safe set of instructions introduced in Section 4.8.1. We split the graph into two for clarity, one for the majority of instructions that do not use factorization, and another for instructions that need factorization for synthesis.

As we see from the graph, most instructions have translations made up of less than 6 instructions. Each instruction may have multiple translations/solutions (Section 4.7.4.1) of different lengths. We show the minimum/maximum solution lengths to showcase the solution diversity. The second graph shows the complexity of solutions that SynthCT can generate with factorization. The three bit-counting instructions all take 800+ instructions to implement: `LZCNTL` (864), `LZCNTQ` (2893), `TZCNTQ` (3277). This shows the strength of our factorization solution: with the help of factorization, SynthCT is able to scale to generate programs of length 3000+, even though the individual synthesis tasks are limited to synthesizing programs of small lengths.

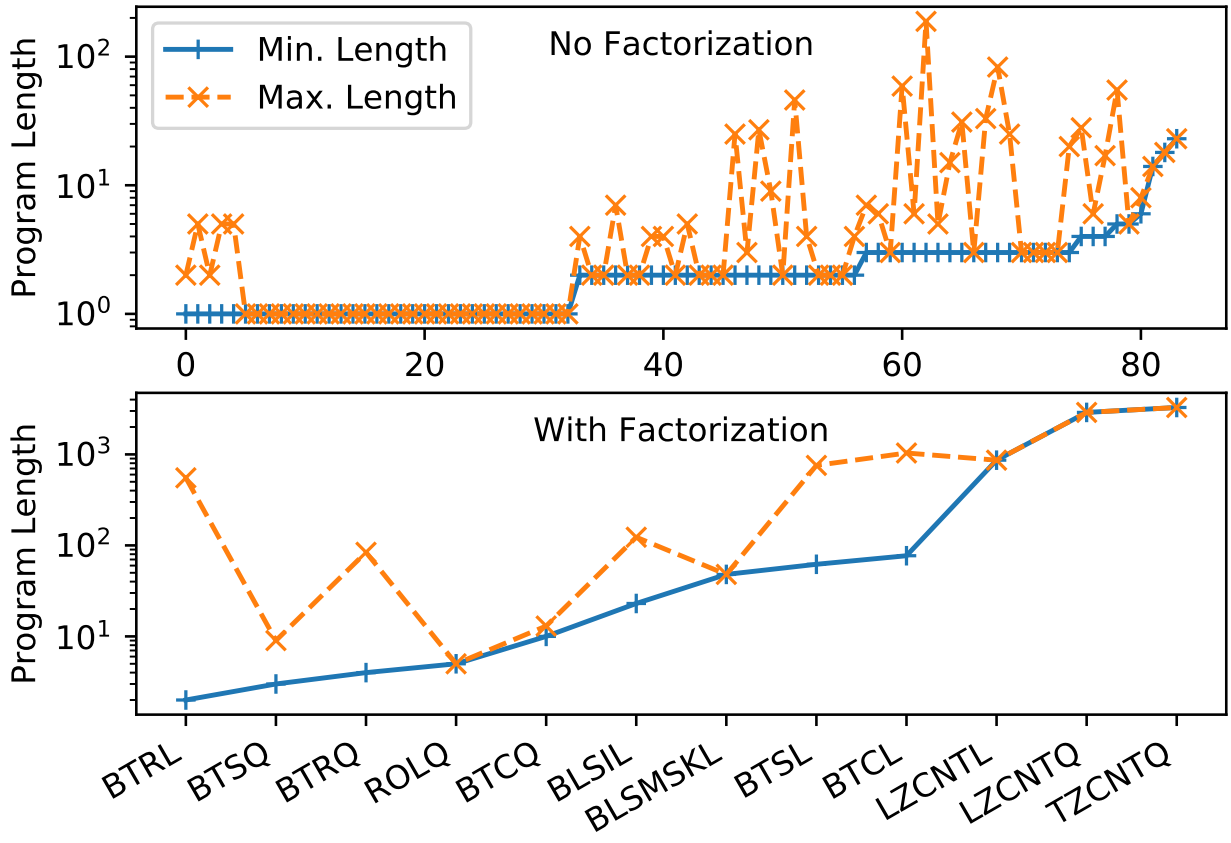


Figure 4.7: Distribution of lengths of Synthesized Translations. Each x-tick denotes an index for an unsafe instruction, or a group of unsafe instructions (when they form a cycle).

4.8.3 Case-Study: Rotate Left

Now, we present a case-study on synthesis of the rotate family of instructions, specifically Rotate-Left (`ROLL`), to illustrate the effectiveness of the node-splitting strategy introduced in Section 4.7.3. During the initial synthesis run, we were unable to get synthesis successes for rotate instructions to synthesize a translation into simpler instructions, e.g., using bitshifts and other bitwise operations. As highlighted earlier, this is primarily due to the complex and opaque K `rol` opcode used in the rotate family of instructions. Without further assistance, SynthCT is unable to meaningfully pick components that increase the likelihood of successful synthesis.

To assist SynthCT in synthesis, we introduce a new transformation in SynthCT to node split the `rol` intermediate opcode into simpler K opcodes. Both the original opcode and its translation is shown is Figure 4.8. The code snippet only shows the rotate part of `ROLL`; the instruction AST has other unrelated nodes to extract and concatenate the relevant bits of its operands (shown in the appendix, Figure 4.13 for reference). Note that before

```
; assume %1 is already mod-ed
; %w = width of rotate rol, %0, and %1
(rol %0 %1) -> (bvor (bvlshr %0 (bvsub %w %1))
                     (bvshl %0 %1))
```

Figure 4.8: Node splitting transformation rule for the ROL opcode.

implementing the transform in SynthCT, we first manually verified the equivalence between the original opcode and the transformed AST by querying the SMT solver for equivalence through Rosette. Once node splitting is applied, the newly generated AST for `ROLL` (shown in the appendix, Figure 4.14) proceeds through the synthesis process as usual. As SynthCT now has more structural information to work with, the component selector can pick better components for synthesis. However, the new AST is larger than the original after translation. We therefore use instruction factorization (Section 4.7.2) to synthesize the solution for `ROLL` in parts.

To summarize, using the composition of two techniques, node splitting and factorization, along with some expert-written transformations, enables SynthCT to synthesize solutions for some of the more complicated instructions in the ISA. These expert-written transforms are a one time effort and scale to multiple ISAs.

4.8.4 Case-Study: Division (DIVL-R32)

In this section, we present a case study to synthesize the division instruction, specifically `DIVL-R32` — a 32-bit division from the x86_64 ISA, that is known to be *unsafe* on today's microarchitectures. Similar to the rotate instructions, we use a combination of node splitting and instruction factorization to synthesize a safe translation for division. Compared to a rotate operation, however, division has far more complex semantics that introduces new scalability challenges in our above mentioned techniques.

The semantics for the `DIVL-R32` instruction is shown in Figure 4.10a. As seen from the figure, the structure of `DIVL-R32` is simple. All the complexity to compute the *quotient* is hidden behind the `div_quotient_int32` K-opcode that takes two operands, the dividend and the divisor, and computes the quotient. The semantics to set the remainder is similar and omitted for brevity. In order to synthesize `DIVL-R32` we must first split the complex quotient opcode into simpler K-operations.

4.8.4.1 Division Algorithm

For simplicity, we use the standard long-division algorithm to compute the quotient and the remainder. The racket implementation of our division is shown in Figure 4.9. As with rotate, we begin by proving that our implementation of divide is semantically equivalent to `DIVL-R32` using an SMT solver. After proving equivalence, we lift the expressions for

```
1
2  (define (implement-DIVL-R32 S r1 r2 r3 r4 r5)
3  (let (
4    [local-r1 (extract 31 0 (state-Rn-ref S r1))]
5    [local-r2 (extract 31 0 (state-Rn-ref S r2))]
6    [local-r3 (extract 31 0 (state-Rn-ref S r3))]
7  )
8  ; Assume r2 is edx and r1 is eax.
9  ; r3 is "real" argument to instruction (divisor)
10 ; r3 outputs quotient and r4 outputs remainder
11 (begin
12 (define dividend (concat local-r2 local-r1))
13 (define divisor (zero-extend local-r3
14                                (bitvector 64)))
15 ; Avoid division by 0
16 (assume (not (bveq divisor (bv 0 64))))
17
18 (set! r3 (bv 0 64))
19 (set! r4 (bv 0 64))
20
21 (for ([r5 (in-range 63 -1 -1)])
22   (set! r4 (bvshl r4 (bv 1 64)))
23   (set! r4 (bvor r4 (bvand (bvlshr dividend
24                                    (bv i 64))
25                            (bv 1 64))))
26   (if (bvuge r4 divisor)
27     (begin
28     (set! r4 (bvsub r4 divisor))
29     (set! r3 (bvor r3 (bvshl (bv 1 64)
30                              (bv r5 64)))))
31     #f))
32
33 (state-Rn-set! S r3
34                (zero-extend (extract 31 0 r3)
35                             (bitvector 64)))
36 (state-Rn-set! S r4
37                (zero-extend (extract 31 0 r4)
38                             (bitvector 64)))
39 )))
40
```

Figure 4.9: Long-division algorithm to node split `DIVL-R32`, implemented in racket. The algorithm is first checked for equivalence with the reference `DIVL-R32` semantics before implementing node splitting in SynthCT.

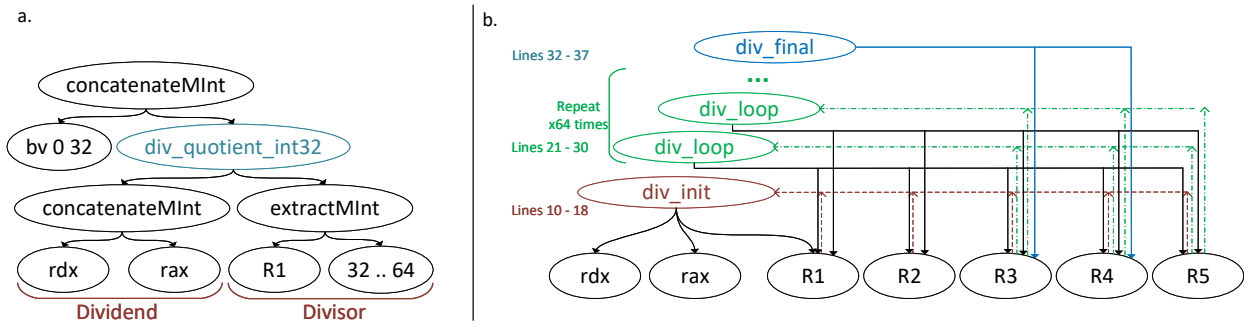


Figure 4.10: `DIVL-R32` semantics AST (a) Semantics of `DIVL-R32` from the K-framework. `DIVL-R32` takes the dividend as implicit inputs in registers `rdx` and `rax` and the divisor as an explicit input in register `R1`. The complex semantics of computing the quotient (and remainder) is hidden behind the opaque K-opcode: `div_quotient_int32` (in blue). (b) Iterative node split of `DIVL-R32`. The algorithm from Figure 4.9 naturally splits `DIVL-R32` into three parts: (i) `div_init` (lines 10 - 18), (ii) `div_loop` (lines 21 - 30), and (iii) `div_final` (lines 32 - 37). Solid lines represent register reads by the newly split opcode and dashed lines represent register writes. The semantics of `DIVL-R32` after the split (in (b)) is equivalent to the original semantics (in (a)).

quotient and remainder directly from our racket implementation to implement node splitting in SynthCT.

4.8.4.2 Scalability Challenges

The expressions generated for the quotient (and remainder) from the long-division algorithm are very large. As shown in Figure 4.9 the loop body to compute the quotient and remainder is repeated `bitwidth` times (in case of `DIVL-R32`, this is 64) leading to a very large AST. While factorization can generate smaller, manageable ASTs for factors, the number of factors to synthesize makes synthesis intractable.

4.8.4.3 Iterative Node Splitting

Rather than splitting `div_quotient_int32` down to the simplest K-operations in one single step, we iteratively split the quotient (and remainder) K-opcode into simpler, smaller pieces in each step. Looking at the long-division algorithm, it naturally breaks into three smaller pieces: (i) An initial setup step that initializes values, (ii) a loop body that is repeated `bitwidth` (64) number of times, and (iii) a final post-loop step that extracts bits corresponding to the quotient (and remainder) and sets the output registers. This split of `div_quotient_int32` along with the relevant lines of the algorithm is shown in Figure 4.10b. Now, each of these splits be broken down further into simpler K-opcodes and synthesized as independent synthesis tasks, using techniques such as factorization as required. Once synthesized, the solutions to the splits are glued together to generate the translation for the

original `DIVL-R32`. This process is exceedingly simple: the solutions to the three splits, i.e., `loop_init`, `loop_body`, and `loop_final`, are concatenated ensuring that the `loop_body` solution is repeated `bitwidth` (64) times. Note that rather than generating a separate split for each iteration of the loop body we can abstract away the loop iteration variable as a symbolic operand to the `loop_body` split. This allows us to synthesize `loop_body` once and instantiate with different values for `iter` (63...0) to generate loop body for the different iterations of the loop.

4.8.4.4 Manual Effort

During the synthesis of `DIVL-R32`, we noticed that a particular factor of `loop_body` failed to synthesize initially. On debugging, we isolated the reason for failure to be the inabilty to synthesize a factor that performed a `bvuge`, an unsigned greater-than or equal comparison, to set a register. As the x86_64 ISA does not contain an instruction to set a register based on a comparison directly (only indirectly through a `cmp` and `setcc`) our component selection failed to pick the right set of components to be able to synthesize this factor. To address the issue, we added a new pseudo-instruction to SynthCT, `PSETCC (r1 r2 r3 r4)`, that sets `r4` based on the result of comparison between `r2` and `r3`. The relational operator to use for comparison, e.g., equal v/s greater-than etc., is controlled by the value in register `r1`. Lastly, we manually implement `PSETCC` using a `cmp` and `setcc` from x86_64 due to limitations with the current component selection strategy.

4.8.5 Secondary Metrics

4.8.5.1 Effectiveness of Component Selection

In this section, we look at the effectiveness of our component selection strategy (Section 4.7.1) and seek to verify our hypothesis: structurally similar instructions are also semantically similar, and hence more likely to be used in successful synthesis. To do so, we look at the number of times an instruction ranked as the i-th nearest neighbor by the component selection algorithm is used in the translation. The histogram plot is shown in Figure 4.11. Recall, for all synthesis tasks $K = 32$ components were chosen. Instructions ranked 0 are most similar to the target synthesis instruction, I_t, and instruction ranked 31 is the least similar. As we see from the histogram, instructions that are more similar to the target instruction, and hence ranked higher by the component selection strategy, are more often used in a synthesis success when compared lower ranked instructions. This plot shows the effectiveness of our component selection strategy and validates our hypothesis from earlier.

4.8.5.2 Synthesis Time for Successes Distribution

Lastly, we look at the time taken for synthesis successes to assess if the strategies we develop in SynthCT actually result in reasonable synthesis time. To do so, we look at the time taken (in seconds) for successful synthesis tasks. The cumulative distribution frequency

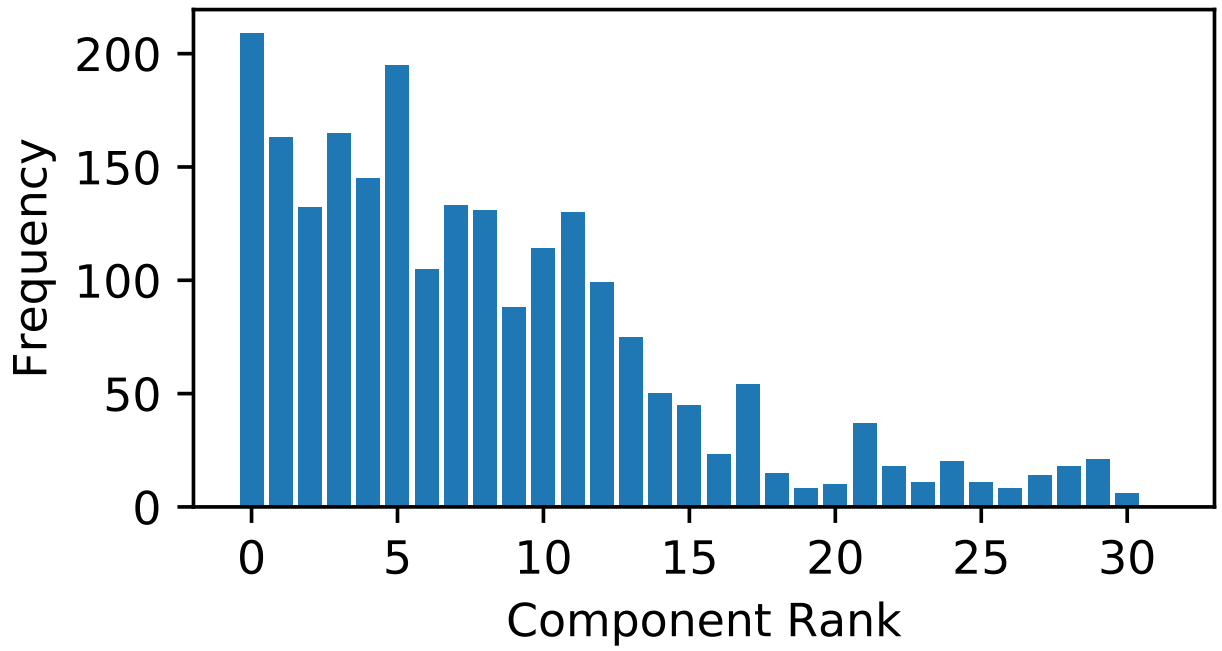


Figure 4.11: Effectiveness of component selection. The histogram shows the number of synthesized translations that use a component ranked at index i (denoted on the X-axis). All synthesis tasks used a fixed set of 32 components drawn using the KNN strategy described in Section 4.7.1. Components most similar are at index 0 while least similar component is index 31. Higher numbers for better ranked components is indicative of a good component selection strategy.

(CDF) for synthesis time is shown in Figure 4.12. From this graph, we see that most successful synthesis tasks generate solutions in a short amount of time: 95% of the synthesis successes are achieved within 2000 seconds. Note that this graph shows the synthesis time for individual synthesis tasks. Instructions that need factorization run multiple smaller synthesis tasks that eventually result in successful synthesis of the original instruction. The synthesis time for such instructions is the sum of time spent in the synthesis of individual synthesis tasks. This total time is not shown in the current figure. From this, we can conclude that the combination of: (i) component selection, (ii) small program lengths, and (iii) limiting the number of registers, does indeed keep the synthesis time for individual synthesis tasks small while allowing for synthesis of translations for the majority of the ISA.

4.9 Related Work

There is a rich literature that studies how to write and run applications in constant time (sometimes called "data-oblivious programming"). For example, application-centric works propose constant-time cryptography [22, 24], machine learning [186, 159, 131], databases [242,

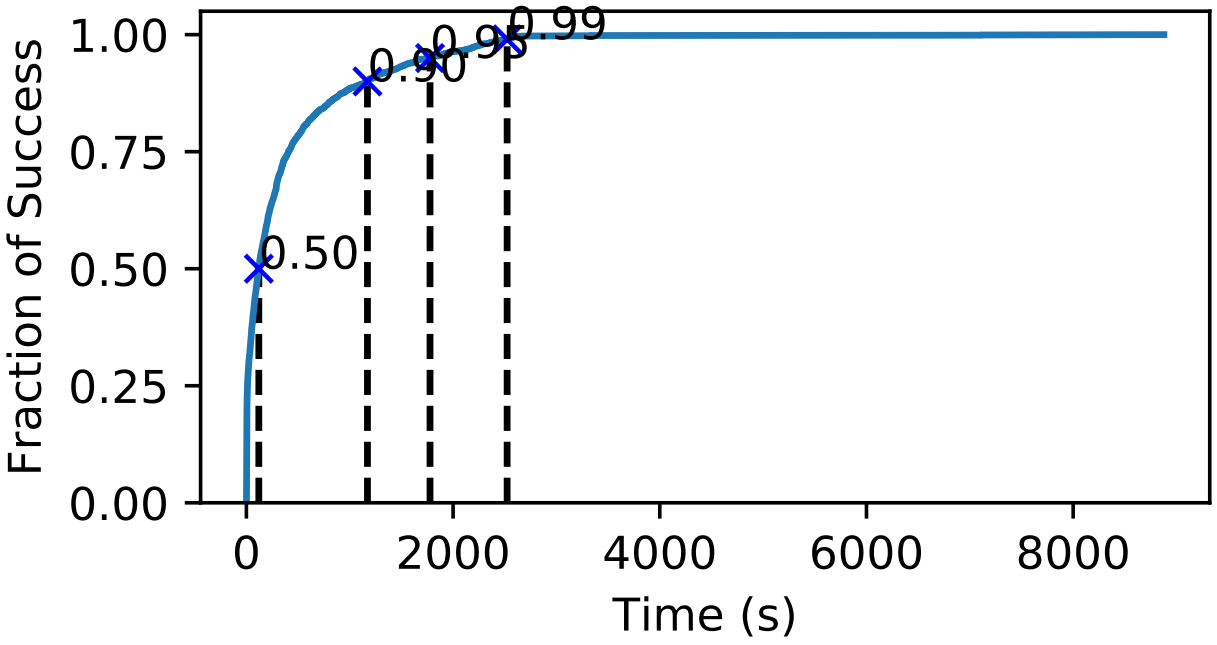


Figure 4.12: Cumulative Distribution of synthesis time for successful synthesis tasks, using data from 2615 synthesis successes. The Y-axis shows the fraction of synthesis successes that take time (in seconds) shown on X-axis. 95% of synthesis successes are achieved quickly, in < 2000 seconds, showing the effectiveness of the combination of strategies developed in SynthCT.

68, 148], memory and datastructures [176, 3], general purpose code [152, 170, 49, 80], utilities [201] and floating point functions [14]. Other works study how to write and compile (e.g., [37, 207, 38]) high-level programs to constant-time ones. ISA abstractions study how to design interfaces usable by both software designers and hardware architects to uphold constant-time security guarantees [234, 200].

SynthCT is complementary to these works. They all produce what we call *mostly constant-time code* (Section 4.1) and assume a fixed set of safe instructions. SynthCT can be used to improve their security/portability and performance, by mapping their code to microarchitecture-specific safe sets.

The closest work to SynthCT conceptually is Arm's DIT [17], data-oblivious ISAs (OISAs) [234] and HW/SW contracts [105, 106]. DIT and OISAs are ISA-level specifications of what instructions are safe and therefore guarantee a consistent safe set across microarchitectures. Widespread acceptance of such abstractions across vendors would therefore decrease the need for SynthCT; yet, such widespread acceptance is far off because ISA changes have an extremely high barrier to entry. Case in point, there is no indication that other vendors beyond Arm will support such features. Lastly, SynthCT is related to HW/SW contracts for secure hardware [105, 106]: a safe set is a concrete example of such

contracts (broadly construed).

There is a large body of related work that uses program synthesis to synthesize semantically equivalent alternate implementations of existing code, e.g., for better performance (superoptimizers [178]) or lower energy utilization [179]. However, none of the existing works have considered the problem of synthesizing an instruction(s) when only a subset of the ISA is allowed to be used in the synthesis. We consider these techniques complementary. For example, it would be interesting to consider a genetic algorithm from [179] as a subroutine for synthesis instead of CEGIS as in SynthCT.

4.10 Discussion

4.10.0.1 Vector and Floating-point instructions

We believe vector instructions can be supported with additional engineering effort, e.g., expanding parsing of x86 semantics and modelling vector registers in synthesis. As vector instructions are multiple instances of non-vector instructions, our techniques such as instruction factorization should reduce them to multiple simpler problems. Floating-point instructions are more challenging. In theory, using a combination of node splitting and factorization, SynthCT should be able generate safe translations for floating-point instructions. However, precisely modelling floating-point operations in bitvector theory in a way that performs well with SMT solvers is known to be challenging [52]. We have therefore left floating point for future work.

4.10.0.2 Integrating with compiler-flows

SynthCT is currently implemented as a post-compilation, binary tool for ease of implementation and generality. However, SynthCT can also be implemented at the compiler level to make it more convenient to use as a part of the compiler toolchain. Additionally, a compiler-level implementation may also generate better performing translations, e.g., by enabling optimizations and better register allocation. Implementing SynthCT as a part of a compiler toolchain also enables integration with security-aware DSLs designed for constant-time programming, e.g., FaCT [37], and allows us to only selectively translate *unsafe instructions* with *secret operand values* rather than all unsafe instructions.

4.10.0.3 Alternate deployment opportunities

SynthCT translations can also be deployed as a part of processor frontend or microcode in a software-transparent way. Using the translations, unsafe instructions may be selectively translated on-the-fly depending on if a security mode bit is set. This allows software to turn the protection on for only sensitive pieces of code. Yet, implementing translations in hardware is more intrusive and may introduce new tradeoffs. For example, the number of

instructions making up the translation may become a first-order constraint as microcode ROM is a scarce resource.

4.11 Conclusion

In this work we develop SynthCT, a framework to facilitate writing *portable* constant-time code, i.e., that is both secure and performant across different microarchitectures. The high-order bit is that with minimal programmer intervention, we can scale to complex ISAs such as x86_64, and even tackle some of the more complex instructions in said ISAs (such as integer division).

Long term, we hope SynthCT (along with other spiritually-similar works [234, 106, 17]) encourages systems and hardware designers to expose abstract but explicit security-critical information in contracts such as ISAs (or their microarchitecture-specific counterparts). It is worth stating that the safe-set specification used in this work is an extremely simple, and already useful, exemplar abstraction of this kind — but is by no means the end of the story. Pushing the idea of microarchitecture-specific specifications further, one can imagine more expressive security contracts (e.g., one that specifies a partition on unsafe instruction operands, which could be used directly by SynthCT to improve CT performance) or even contracts geared for other metrics such as performance (e.g., more formal models of instruction interactions and their resulting pipeline throughputs).

4.12 Example Translations

This section shows examples of synthesized translations. See Table 4.5.

Unsafe Instruction(s)	Safe Instructions in a Translation
ADCB-RH-RH	MOVQ, SBBB-RH-RH, (SUBB-RH-RH V SUBB-RH-R8)
DECL-R32	(XORQ-R64-R64 V PXOR-R64-R64), MOVQ, (NOTL-R32 V ANDL-R32-R32)
INCL-R32	PNOT, NEGL-R32
BLSIL-R32-R32	(NOTL-R32 V ANDL-R32-R32), MOVQ, SHLQ-R64-CL, SHRQ-R64-CL
	SUBQ-R64-R64, (XORQ-R64-R64 V PXOR-R64-R64), SARQ-R64-CL
	NEGL-R32, PSET-FLAG
BTSL-R32-R32	MOVQ, SHLQ-R64-CL, SHRQ-R64-CL, SUBQ-R64-R64
	(XORQ-R64-R64 V PXOR-R64-R64), SARQ-R64-CL, NOP, PSET-FLAG
CLTD	IMULL-R32, MOVQ
PCMOV-R64-R64-R64	CMOVEQ-R64-R64, (ANDQ-R64-R64 V PAND-R64-R64)
LZCNTQ-R64-R64	MOVQ, SHLQ-R64-CL, SHRQ-R64-CL, SUBQ-R64-R64
	(XORQ-R64-R64 V PXOR-R64-R64)
	(ANDQ-R64-R64 V PAND-R64-R64), CMOVEQ-R64-R64, SARQ-R64-CL, PSET-FLAG
POPCNTL-R32-R32	POPCNTQ-R64-R64, (NOTL-R32 V ANDL-R32-R32)
SUBL-R32-R32	BEXTRL-R32-R32-R32, SBBL-R32-R32
XORL-R32-R32	(XORQ-R64-R64 V PXOR-R64-R64), XCHGL-R32-EAX, SARQ-R64-CL
	PSPLIT
(INCW-R16 V DECW-R16 V NEGW-R16)	MOVW-R16-R16, NEGL-R32

Table 4.5: Example translations for unsafe instructions. The table shows which safe instruction opcodes (Section 4.8.1.3) are used to translate several representative unsafe instructions. The "V" denotes a disjunction: These instructions form an equivalence class and we have the freedom to choose one of the instructions from the class to utilize in the final translation.

4.13 Case Study: ROLL-R32-CL

This section shows the semantics AST of the original `ROLL-R32-CL` and `ROLL-R32-CL` after splitting. See Figures 4.13 and 4.14.

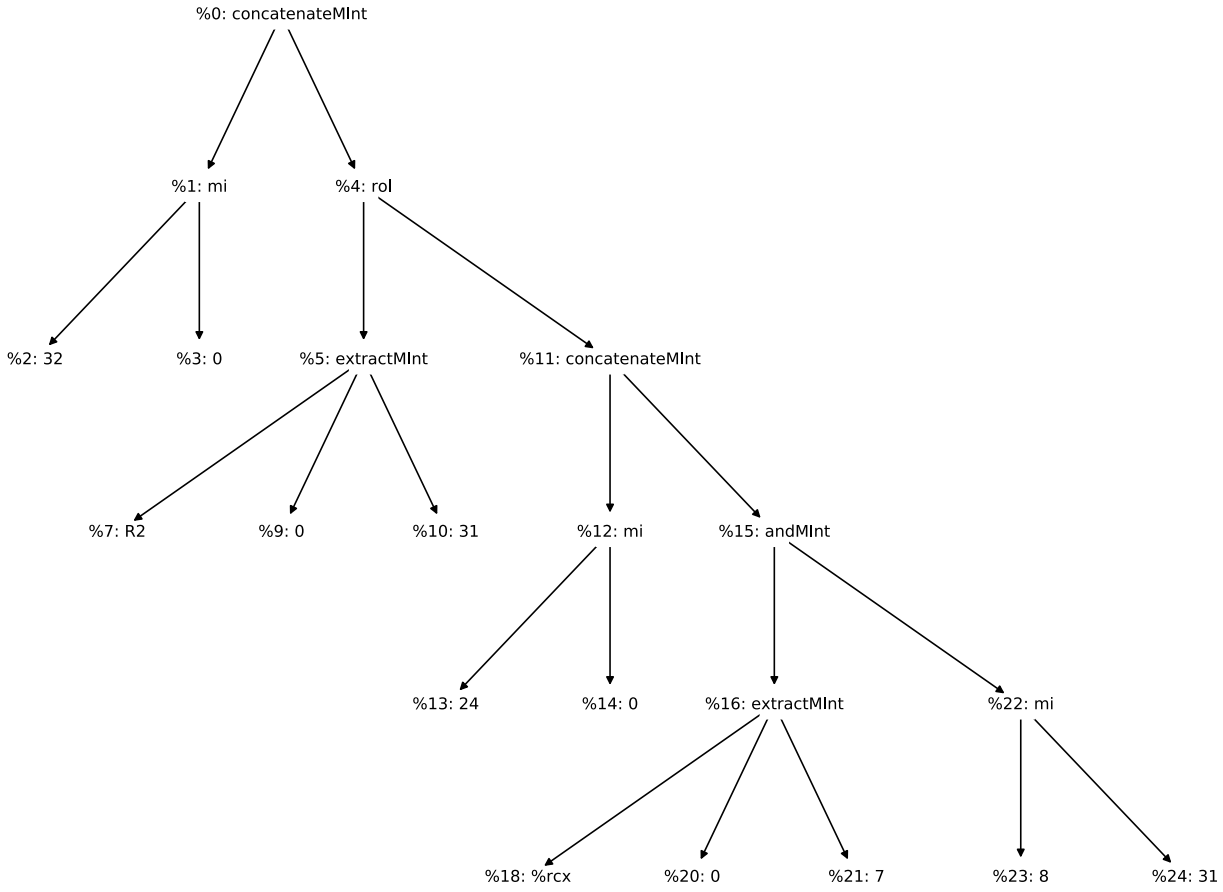


Figure 4.13: The original ROLL-R32-CL instruction (used in case-study Section 4.8.3). The AST was generated directly from $\mathbb{K}$ semantics.

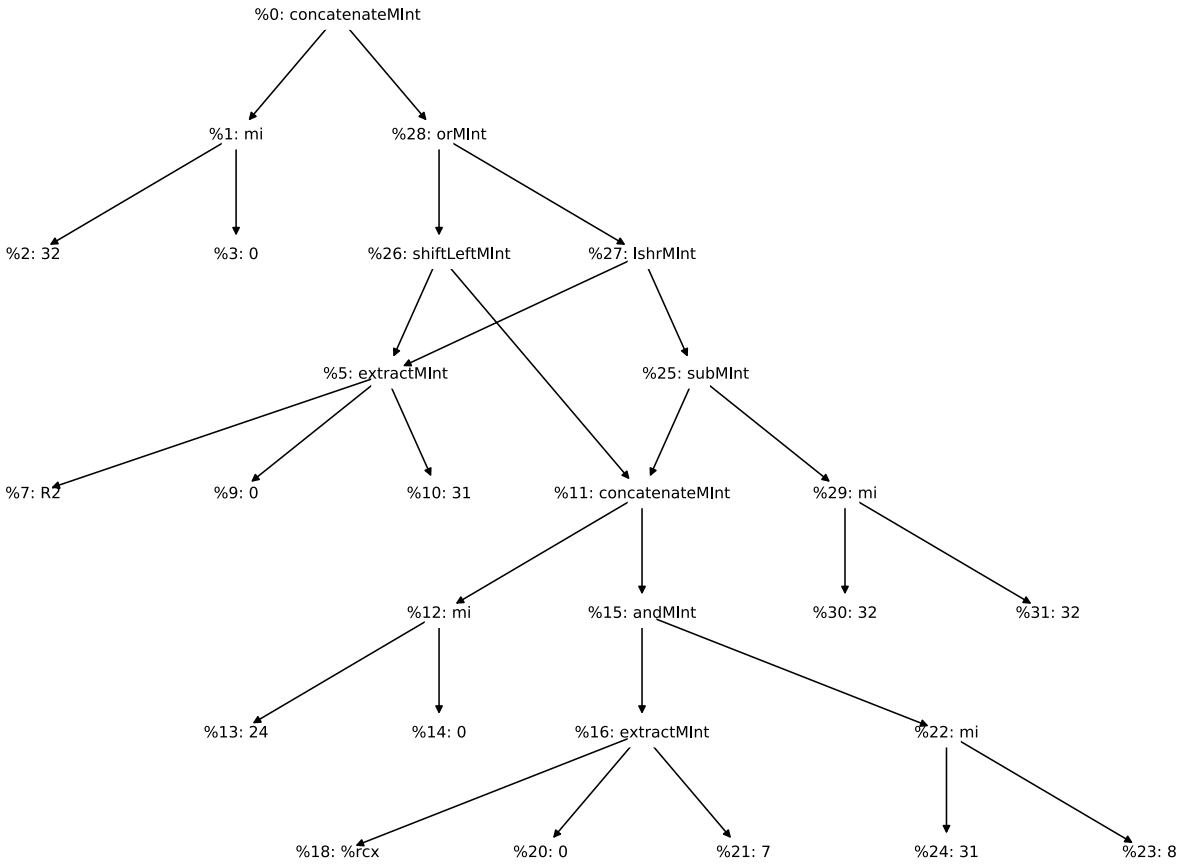


Figure 4.14: The ROLL instruction after splitting the `rol` opcode. The `rol` opcode from the previous figure (Figure 4.13) is replaced by the equivalent translation in Figure 4.8, as described in the case-study (Section 4.8.3). Notice that the node labelled as **(%4: rol)** has now been replaced by the subtree rooted by the node labelled **(%28: orMInt)**. The new AST is semantically equivalent to the old AST and is used in further synthesis steps, e.g., factorization.

Chapter 5

Conclusion & Future Work

In summary, this thesis provides the first-steps towards a comprehensive hardware-software solution to protect critical software, e.g., cryptography, against microarchitectural timing side-channels without hardware modifications, disabling optimizations, or slowing down general purpose compute. The key idea: automatically verify and lift software-facing, security-centric abstractions from microarchitecture implementations. Then use the security specification with compiler-like automated tooling to harden software against side-channels for a particular microarchitecture. To enable this vision, the key technical advancement in this thesis is the development of a new scalable invariant learning algorithm, H-Houdini, that enables security verification on large hardware designs.

Now we will briefly discuss two interesting threads of future research directions:

(1) The property developed and verified in this thesis, SISP, is a simple, yet useful property in practiced. Related work, concurrent to ours, have suggested more expressive properties, in the form of hardware-software contracts [107, 42, 106]. Nascent work in this area have made progress is proving such contracts on hardware designs [198, 218], however, they do not scale to large designs such as BOOM. An interesting research direction: *Can we generalize* VeloCT *to scalably prove more expressive properties, e.g., hardware-software contracts, on large designs like BOOM?* This would enable the development of more general and expressive software-facing abstractions, which we can once again use with systems like SynthCT to generate high-performance, provably secure code.

(2) In so far, we've shown that H-Houdini is a scalable algorithm for proving *security* properties. An open and interesting research question: *Can we generalize* H-Houdini *to prove more general properties? Would* H-Houdini *show similar scalability results for the more general class of properties?* Generalizing H-Houdini to scalably prove general properties will allow verification to be applied to many more practical systems, enabling a safe, bug-free, trustworthy future.

Bibliography

[1] Onur Aciicmez, Jean-Pierre Seifert, and Cetin Kaya Koc. "Predicting Secret Keys via Branch Prediction". In: *IACR'06* (2006).

[2] Andrew Adiletta and Berk Sunar. "Spill The Beans: Exploiting CPU Cache Side-Channels to Leak Tokens from Large Language Models". In: *arXiv'25* ().

[3] Adil Ahmad et al. "Obliviate: A Data Oblivious Filesystem for Intel SGX". In: *NDSS'18*.

[4] Aws Albarghouthi and Kenneth L. McMillan. "Beautiful Interpolants". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 313–329. DOI: 10.1007/978-3-642-39799-8_22. URL: https://doi.org/10.1007/978-3-642-39799-8%5C_22.

[5] Francesco Alberti et al. "Lazy Abstraction with Interpolants for Arrays". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*. Ed. by Nikolaj S. Bjørner and Andrei Voronkov. Vol. 7180. Lecture Notes in Computer Science. Springer, 2012, pp. 46–61. DOI: 10.1007/978-3-642-28717-6_7. URL: https://doi.org/10.1007/978-3-642-28717-6%5C_7.

[6] Alejandro Cabrera Aldaya et al. *Port Contention for Fun and Profit*. IACR'18. 2018.

[7] Alejandro Cabrera Aldaya et al. "Port Contention for Fun and Profit". In: *S&P*. 2019.

[8] Jose Bacelar Almeida et al. "Verifying Constant-Time Implementations". In: *USENIX Security'16*.

[9] Rajeev Alur et al. "Search-based program synthesis". In: *Commun. ACM* 61 (2018), pp. 84–93.

[10] Rajeev Alur et al. "Syntax-Guided Synthesis". In: *Dependable Software Systems Engineering*. Vol. 40. 2015, pp. 1–25. DOI: 10.3233/978-1-61499-495-4-1.

[11] Rajeev Alur et al. "Syntax-guided Synthesis". In: *FMCAD'13*.

[12] Alon Amid et al. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs". In: *IEEE Micro* 40.4 (2020), pp. 10–21. ISSN: 1937-4143. DOI: 10.1109/MM.2020.2996616.

[13] Glenn Ammons, Rastislav Bodík, and James R Larus. "Mining specifications". In: *ACM Sigplan Notices* 37.1 (2002), pp. 4–16.

[14] Marc Andrysco et al. "On Subnormal Floating Point and Abnormal Timing". In: *S&P'15*.

[15] Anyscale. *Anyscale Homepage*. https://www.anyscale.com/. Accessed: 2024-06-23.

[16] ARM. *ARM Data Indepdent Timing Specification*. [Online; accessed 3-Jun-2021]. 2021.

[17] *Arm Architecture Registers Armv8*. https://developer.arm.com/docs/ddi0595/latest/aarch32-system-registers/cpsr.

[18] Krste Asanović et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, 2016.

[19] Ehsan Atoofian and Amirali Baniasadi. "Improving energy-efficiency by bypassing trivial computations". In: *IPDPS'05*.

[20] Haniel Barbosa et al. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *TACAS*. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24.

[21] Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. "Secure information flow by self-composition". In: *Mathematical Structures in Computer Science* 21.6 (2011), pp. 1207–1252.

[22] Daniel J. Bernstein. "Curve25519: New Diffie-Hellman Speed Records". In: *PKC'06*.

[23] Daniel J. Bernstein. *djbsort*. https://sorting.cr.yp.to/. Accessed: 2024-12-14.

[24] Daniel J. Bernstein. "The Poly1305-AES Message-Authentication Code". In: *FSE'05*.

[25] Adam Betts et al. "GPUVerify: a verifier for GPU kernels". In: *OOPSLA'12*.

[26] Atri Bhattacharyya et al. "SMoTherSpectre: Exploiting Speculative Execution through Port Contention". In: *CCS'19*.

[27] Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. "Counterexample to Induction-Guided Abstraction-Refinement (CTIGAR)". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 831–848. DOI: 10.1007/978-3-319-08867-9_55. URL: https://doi.org/10.1007/978-3-319-08867-9%5C_55.

[28] Marcel Böhme et al. "Directed greybox fuzzing". In: *CCS'17*.

[29] Aaron R Bradley. "SAT-based Model Checking Without Unrolling". In: *VMCAI'11*.

[30] David Brooks and Margaret Martonosi. "Dynamically exploiting narrow width operands to improve processor power and performance". In: *HPCA'99*.

[31] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." In: *OSDI'08*.

[32] Cristiano Calcagno, Dino Distefano, and Viktor Vafeiadis. "Bi-abductive Resource Invariant Synthesis". In: *APLAS'09*.

[33] Cristiano Calcagno et al. "Compositional shape analysis by means of bi-abduction". In: *POPL'09*.

[34] Ramon Canal, Antonio González, and James E Smith. "Very low power pipelines using significance compression". In: *MICRO'00*.

[35] Claudio Canella et al. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *USENIX Security'19*.

[36] Claudio Canella et al. "Fallout: Leaking Data on Meltdown-Resistant CPUs". In: *CCS*. 2019.

[37] Sunjay Cauligi et al. "FaCT: A DSL for timing-sensitive computation". In: *PLDI'19*.

[38] Sunjay Cauligi et al. "Towards Constant-Time Foundations for the New Spectre Era". In: *PLDI'20*.

[39] *ChaCha20 (BearSSL)*. https://bearssl.org/gitweb/. Accessed: 2024-12-14.

[40] Adrien Champion et al. "ICE-based refinement type discovery for higher-order functional programs". In: *Journal of Automated Reasoning* (2020).

[41] Kevin Cheang et al. "A Formal Approach to Secure Speculation". In: *Proceedings of the Computer Security Foundations Symposium (CSF)*. 2019.

[42] Kevin Cheang et al. "A Formal Approach to Secure Speculation". In: *CSF'19*. 2019.

[43] Boru Chen et al. "GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers". In: *SEC'24*.

[44] Rutvik Choudhary et al. "DECLASSIFLOW: A Static Analysis for Modeling Non-Speculative Knowledge to Relax Speculative Execution Security Measures". In: *CCS'23*.

[45] Rutvik Choudhary et al. "Speculative Privacy Tracking (SPT): Leaking Information From Speculative Execution Without Compromising Privacy". In: *MICRO'21*.

[46] Vasek Chvatal. "A greedy heuristic for the set-covering problem". In: *Mathematics of operations research* 4.3 (1979), pp. 233–235.

[47] Alessandro Cimatti et al. "Infinite-state invariant checking with IC3 and predicate abstraction". In: *Formal Methods Syst. Des.* (2016).

[48] Edmund M. Clarke et al. "Counterexample-Guided Abstraction Refinement". In: *CAV*. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 154–169.

[49] Bart Coppens et al. "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors". In: *S&P'09*.

[50] William Craig. "Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem". In: *J. Symb. Log.* (1957).

[51] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. "DySy: dynamic symbolic execution for invariant inference". In: *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. Ed. by Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn. ACM, 2008, pp. 281–290. DOI: 10.1145/1368088.1368127. URL: https://doi.org/10.1145/1368088.1368127.

[52] Eva Darulova and Viktor Kuncak. "Sound Compilation of Reals". In: *POPL'14*.

[53] Sandeep Dasgupta et al. "A Complete Formal Semantics of x86-64 User-level Instruction Set Architecture". In: *PLDI'19*. PLDI 2019. Phoenix, AZ, USA, 2019, pp. 1133–1148. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314601.

[54] *Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance*. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html.

[55] Shuwen Deng et al. "Evaluation of Cache Attacks on Arm Processors and Secure Caches". In: *IEEE Transactions on Computers* 71 (2022), pp. 2248–2262.

[56] Calvin Deutschbein and Cynthia Sturton. "Mining Security Critical Linear Temporal Logic Specifications for Processors". In: *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. 2018.

[57] Lucas Deutschmann et al. "A scalable formal verification methodology for data-oblivious hardware". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).

[58] Lucas Deutschmann et al. "Towards a Formally Verified Hardware Root-of-Trust for Data-Oblivious Computing". In: *DAC'22*.

[59] Isil Dillig et al. "Inductive invariant generation via abductive inference". In: *OOPSLA'13*.

[60] Sushant Dinesh, Grant Garrett-Grossman, and Christopher W. Fletcher. "SynthCT: Towards Portable Constant-Time Code". In: *NDSS'22*.

[61] Sushant Dinesh, Madhusudan Parthasarathy, and Christopher Fletcher. "CONJUNCT: Learning Inductive Invariants to Prove Unbounded Instruction Safety Against Microarchitectural Timing Attacks". In: *S&P'24*.

[62] Sushant Dinesh, Yongye Zhu, and Christopher W. Fletcher. "H-Houdini: Scalable Invariant Learning". In: *ASPLOS '25*.

[63] Sushant Dinesh, Yongye Zhu, and Christopher W. Fletcher. *VeloCT*. https://github.com/FPSG-UIUC/veloct. Accessed: 2024-12-15.

[64] Travis Downs. *Hardware Store Elimination*. Accessed on 06/17/2020. URL: https://travisdowns.github.io/g/2020/05/13/intel-zero-opt.html.

[65] Rohit Dureja et al. "IC3 with Internal Signals". In: *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*. IEEE, 2021, pp. 63–71. DOI: 10.34727/2021/ISBN.978-3-85448-046-4_14. URL: https://doi.org/10.34727/2021/isbn.978-3-85448-046-4%5C_14.

[66] Niklas Eén, Alan Mishchenko, and Robert Brayton. "Efficient implementation of property directed reachability". In: *FMCAD'11*.

[67] Michael D. Ernst et al. "Quickly detecting relevant program invariants". In: *ICSE'00*.

[68] Saba Eskandarian and Matei Zaharia. "ObliDB: Oblivious Query Processing for Secure Databases". In: *Proceedings of the VLDB Endowment* ().

[69] Dmitry Evtyushkin and Dmitry Ponomarev. "Covert Channels through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations". In: *CCS'16*.

[70] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Jump over ASLR: Attacking branch predictors to bypass ASLR". In: *MICRO*. 2016.

[71] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Understanding and Mitigating Covert Channels Through Branch Predictors". In: *TACO'16* ().

[72] Dmitry Evtyushkin et al. "BranchScope: A New Side-Channel Attack on Directional Branch Predictor". In: *ASPLOS'18*. 2018.

[73] P. Ezudheen et al. "Horn-ICE learning for synthesizing invariants and contracts". In: *OOPSLA'18*.

[74] Mohammad Rahmani Fadiheh et al. "A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors". In: *DAC'20*.

[75] Mohammad Rahmani Fadiheh et al. "An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors". In: *IEEE Transactions on Computers* (2023).

[76] Mohammad Rahmani Fadiheh et al. "Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking". In: *CoRR* abs/1812.04975 (2018). arXiv: 1812.04975. URL: http://arxiv.org/abs/1812.04975.

[77] Grigory Fedyukovich, Samuel J Kaufman, and Rastislav Bodík. "Sampling invariants from frequency distributions". In: *FMCAD'17*.

[78] Grigory Fedyukovich et al. "Quantified Invariants via Syntax-Guided Synthesis". In: *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11561. Lecture Notes in Computer Science. Springer, 2019, pp. 259–277. DOI: 10.1007/978-3-030-25540-4_14. URL: https://doi.org/10.1007/978-3-030-25540-4%5C_14.

[79] Grigory Fedyukovich et al. "Solving Constrained Horn Clauses Using Syntax and Data". In: *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*. Ed. by Nikolaj S. Bjørner and Arie Gurfinkel. IEEE, 2018, pp. 1–9. DOI: 10.23919/FMCAD.2018.8603011. URL: https://doi.org/10.23919/FMCAD.2018.8603011.

[80] Susanne Felsen et al. "Secure and private function evaluation with Intel SGX". In: *SIGSAC'19*.

[81] Ben A. Fisch et al. "Iron: Functional Encryption using Intel SGX". In: *CCS'17*.

[82] Cormac Flanagan and K Rustan M Leino. "Houdini, an annotation assistant for ESC/Java". In: *FME'01*.

[83] Michael Flanders et al. "Avoiding Instruction-Centric Microarchitectural Timing Channels Via Binary-Code Transformations". In: *ASPLOS'24*.

[84] Jacob Fustos, Farzad Farshchi, and Heechul Yun. "SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks". In: *2019 56th ACM/IEEE Design Automation Conference (DAC)* (2019), pp. 1–6.

[85] Mark Gabel and Zhendong Su. "Javert: fully automatic mining of general temporal properties from dynamic traces". In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*. Ed. by Mary Jean Harrold and Gail C. Murphy. ACM, 2008, pp. 339–349. DOI: 10.1145/1453101.1453150. URL: https://doi.org/10.1145/1453101.1453150.

[86] Zibo Gao et al. "I Know What You Said: Unveiling Hardware Cache Side-Channels in Local Large Language Model Inference". In: *arXiv preprint arXiv:2505.06738* (2025).

[87] Pranav Garg et al. "ICE: A robust framework for learning invariants". In: *CAV'14*.

[88] Pranav Garg et al. "Learning invariants using decision trees and implication counterexamples". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 499–512. DOI: 10.1145/2837614.2837664. URL: https://doi.org/10.1145/2837614.2837664.

[89] Pierre-Loıc Garoche, Temesghen Kahsai, and Cesare Tinelli. "Incremental Invariant Generation Using Logic-Based Automatic Abstract Transformers". In: *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*. Ed. by Guillaume Brat, Neha Rungta, and Arnaud Venet. Vol. 7871. Lecture Notes in Computer Science. Springer, 2013, pp. 139–154. DOI: 10.1007/978-3-642-38088-4_10. URL: https://doi.org/10.1007/978-3-642-38088-4%5C_10.

[90] Qian Ge et al. *A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware*. Cryptology ePrint Archive, Report 2016/613. https://eprint.iacr.org/2016/613. 2016.

[91] Roberto Giacobazzi. "Abductive Analysis of Modular Logic Programs." In: *ILPS'94*.

[92] Github. *Zkt "Constant Time" Instruction List*. https://github.com/rvkrypto/riscv-zkt-list. 2023.

[93] Klaus v. Gleissenthall et al. "IODINE: Verifying Constant-Time Execution of Hardware". In: *USENIX Security'19*.

[94] Klaus v. Gleissenthall et al. "Solver-Aided Constant-Time Hardware Verification". In: *CCS'21*.

[95] Patrice Godefroid. "Fuzzing: Hack, Art, and Science". In: *Commun. ACM* 63 (2020), pp. 70–76. ISSN: 0001-0782. DOI: 10.1145/3363824.

[96] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. "Grammar-based whitebox fuzzing". In: *PLDI'08*.

[97] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed automated random testing". In: *PLDI'05*.

[98] Aman Goel and Karem A. Sakallah. "AVR: Abstractly Verifying Reachability". In: *TACAS'20*.

[99] Aman Goel and Karem A. Sakallah. "Model Checking of Verilog RTL Using IC3 with Syntax-Guided Abstraction". In: *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*. Ed. by Julia M. Badger and Kristin Yvonne Rozier. Vol. 11460. Lecture Notes in Computer Science. Springer, 2019, pp. 166–185. DOI: 10.1007/978-3-030-20652-9_11. URL: https://doi.org/10.1007/978-3-030-20652-9%5C_11.

[100] Susanne Graf and Hassen Saıdi. "Construction of Abstract State Graphs with PVS". In: *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*. Ed. by Orna Grumberg. Vol. 1254. Lecture Notes in Computer Science. Springer, 1997, pp. 72–83. DOI: 10.1007/3-540-63166-6_10. URL: https://doi.org/10.1007/3-540-63166-6%5C_10.

[101] Ben Gras et al. "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures". In: *NDSS'20*.

[102] Ben Gras et al. "ASLR on the Line: Practical Cache Attacks on the MMU". In: *NDSS'17*.

[103] Ben Gras et al. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks". In: *USENIX Security'18*.

[104] Johann Großschädl et al. "Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications". In: *ICISC'09*.

[105] Marco Guarnieri and Marco Patrignani. *Contract-Aware Secure Compilation*.

[106] Marco Guarnieri et al. "Hardware-Software Contracts for Secure Speculation". In: *IEEE S&P 2021*.

[107] Marco Guarnieri et al. "Spectector: Principled Detection of Speculative Information Flows". In: *S&P'20*.

[108] Travis Hance et al. "Finding Invariants of Distributed Systems: It's a Small (Enough) World After All". In: *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*. Ed. by James Mickens and Renata Teixeira. USENIX Association, 2021, pp. 115–131. URL: https://www.usenix.org/conference/nsdi21/presentation/hance.

[109] Sudheendra Hangal and Monica S. Lam. "Tracking down software bugs using automatic anomaly detection". In: *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*. Ed. by Will Tracz, Michal Young, and Jeff Magee. ACM, 2002, pp. 291–301. DOI: 10.1145/581339.581377. URL: https://doi.org/10.1145/581339.581377.

[110] Sudheendra Hangal et al. "IODINE: a tool to automatically infer dynamic invariants for hardware designs". In: *Proceedings of the 42nd Design Automation Conference, DAC 2005, San Diego, CA, USA, June 13-17, 2005*. Ed. by William H. Joyner Jr., Grant Martin, and Andrew B. Kahng. ACM, 2005, pp. 775–778. DOI: 10.1145/1065579.1065786. URL: https://doi.org/10.1145/1065579.1065786.

[111] Zyad Hassan, Aaron R. Bradley, and Fabio Somenzi. "Better generalization in IC3". In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013, pp. 157–164. URL: https://ieeexplore.ieee.org/document/6679405/.

[112] Thomas A. Henzinger et al. "Abstractions from proofs". In: *ACM SIGPLAN Notices* 49.4S (2014), pp. 79–91. DOI: 10.1145/2641638.2641655. URL: https://doi.org/10.1145/2641638.2641655.

[113] Thomas A. Henzinger et al. "Software Verification with BLAST". In: *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*. Ed. by Thomas Ball and Sriram K. Rajamani. Vol. 2648. Lecture Notes in Computer Science. Springer, 2003, pp. 235–239. DOI: 10.1007/3-540-44829-2_17. URL: https://doi.org/10.1007/3-540-44829-2%5C_17.

[114] Kryštof Hoder and Nikolaj Bjørner. "Generalized property directed reachability". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2012, pp. 157–171.

[115] HWMCC. *HWMCC'24 Results*. https://hwmcc.github.io/2024/. Accessed: 2025-05-214.

[116] Alexander Ivrii and Arie Gurfinkel. "Pushing To the Top". In: *FMCAD'15*.

[117] Susmit Jha and Sanjit A Seshia. "A theory of formal synthesis via inductive learning". In: *Acta Informatica* 54 (2017), pp. 693–726.

[118] Ranjit Jhala and Kenneth L. McMillan. "A Practical and Complete Approach to Predicate Refinement". In: *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*. Ed. by Holger Hermanns and Jens Palsberg. Vol. 3920. Lecture Notes in Computer Science. Springer, 2006, pp. 459–473. DOI: 10.1007/11691372_33. URL: https://doi.org/10.1007/11691372%5C_33.

[119] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *ISCA '17*.

[120] Adharsh Kamath et al. "Finding Inductive Loop Invariants using Large Language Models". In: *CoRR* abs/2311.07948 (2023). DOI: 10.48550/ARXIV.2311.07948. arXiv: 2311.07948. URL: https://doi.org/10.48550/arXiv.2311.07948.

[121] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Proceedings of a symposium on the Complexity of Computer Computations*. Ed. by Raymond E. Miller and James W. Thatcher. The IBM Research Symposia Series. 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9.

[122] Ilhyun Kim and Mikko H. Lipasti. "Implementing Optimizations at Decode Time". In: *ISCA'02*.

[123] Jason Kim et al. "iLeakage: Browser-based Timerless Speculative Execution Attacks on Apple Devices". In: CCS '23. Copenhagen, Denmark, 2023.

[124] Vladimir Kiriansky et al. "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors". In: *MICRO'18*. 2018.

[125] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *S&P'19*.

[126] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *CRYPTO'96*.

[127] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis". In: *CRYPTO'99*. 1999.

[128] David Kohlbrenner and Hovav Shacham. "On the effectiveness of mitigations against floating-point timing channels". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 69–81. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/kohlbrenner.

[129] Siddharth Krishna, Christian Puhrsch, and Thomas Wies. "Learning invariants using decision trees". In: *arXiv preprint arXiv:1501.04725* (2015).

[130] Akash Lal and Shaz Qadeer. "Powering the static driver verifier using corral". In: *FSE'14*.

[131] Hyun Bin Lee et al. "DOVE: A Data-Oblivious Virtual Environment". In: *NDSS'21*.

[132] Suho Lee and Karem A. Sakallah. "Unbounded Scalable Verification Based on Approximate Property-Directed Reachability and Datapath Abstraction". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 849–865. DOI: 10.1007/978-3-319-08867-9_56. URL: https://doi.org/10.1007/978-3-319-08867-9%5C_56.

[133] K.M. Lepak and M.H. Lipasti. "Silent Stores for Free". In: *MICRO'00*.

[134] Wenchao Li. "Specification Mining: New Formalisms, Algorithms and Applications". PhD thesis. EECS Department, University of California, Berkeley, Mar. 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-20.html.

[135] Zhenmin Li and Yuanyuan Zhou. "PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code". In: *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*. Ed. by Michel Wermelinger and Harald C. Gall. ACM, 2005, pp. 306–315. DOI: 10.1145/1081706.1081755. URL: https://doi.org/10.1145/1081706.1081755.

[136] Ben Liblit et al. "Bug isolation via remote program sampling". In: *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*. Ed. by Ron Cytron and Rajiv Gupta. ACM, 2003, pp. 141–154. DOI: 10.1145/781131.781148. URL: https://doi.org/10.1145/781131.781148.

[137] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *USENIX Security'18*. 2018.

[138] Moritz Lipp et al. "PLATYPUS: Software-based Power Side-Channel Attacks on x86". In: (2020).

[139] Chang Liu et al. "ObliVM: A Programming Framework for Secure Computation". In: S&P '15.

[140] V. Benjamin Livshits et al. "Merlin: specification inference for explicit information flow problems". In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. Ed. by Michael Hind and Amer Diwan. ACM, 2009, pp. 75–86. DOI: 10.1145/1542476.1542485. URL: https://doi.org/10.1145/1542476.1542485.

[141] lowRISC. *Ibex*. https://github.com/lowRISC/ibex. [Online; accessed 18-Apr-2023]. 2023.

[142] Haojun Ma et al. "I4: incremental inference of inductive invariants for verification of distributed protocols". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. Ed. by Tim Brecht and Carey Williamson. ACM, 2019, pp. 370–384. DOI: 10.1145/3341301.3359651. URL: https://doi.org/10.1145/3341301.3359651.

[143] Rupak Majumdar and Koushik Sen. "Hybrid concolic testing". In: *ICSE'07*.

[144] Makai Mann et al. "Pono: a flexible and extensible SMT-based model checker". In: *International Conference on Computer Aided Verification*. Springer. 2021, pp. 461–474.

[145] Kenneth L. McMillan. "Applications of Craig Interpolants in Model Checking". In: *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. Lecture Notes in Computer Science. Springer, 2005, pp. 1–12. DOI: 10.1007/978-3-540-31980-1_1. URL: https://doi.org/10.1007/978-3-540-31980-1%5C_1.

[146] Kenneth L. McMillan. "Interpolation and SAT-based model checking". In: *CAV'03*.

[147] Tomás Mikolov et al. "Distributed Representations of Words and Phrases and their Compositionality". In: *NIPS'13*.

[148] P. Mishra et al. "Oblix: An Efficient Oblivious Search Index". In: *S&P'18*.

[149] Sparsh Mittal. "A survey of value prediction techniques for leveraging value locality". In: *CCPE'17* 29.21 (2017), e4250.

[150] Ahmad Moghimi et al. "MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations". In: *International Journal of Parallel Programming* 47 (2019), pp. 538–570.

[151] Carlos Molina, Antonio González, and Jordi Tubella. "Dynamic Removal of Redundant Computations". In: *ICS'99*.

[152] David Molnar et al. "The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks". In: *IACR'05* ().

[153] Nicholas Mosier et al. "Axiomatic Hardware-Software Contracts for Security". In: *ISCA'22*.

[154] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *TACAS'08*.

[155] Annamalai Narayanan et al. "graph2vec: Learning Distributed Representations of Graphs". In: *CoRR* abs/1707.05005 (2017).

[156] Alireza Nazari et al. "EDDIE: EM-Based Detection of Deviations in Program Execution". In: *ISCA'17*.

[157] Daniel Neider et al. "SORCAR: Property-Driven Algorithms for Learning Conjunctive Invariants". In: *SAS'19*.

[158] Aina Niemetz et al. "Btor2, btormc and boolector 3.0". In: *CAV'18*.

[159] Olga Ohrimenko et al. "Oblivious Multi-Party Machine Learning on Trusted Processors". In: *USENIX Security'16*.

[160] Oleksii Oleksenko et al. "SpecFuzz: Bringing Spectre-type vulnerabilities to the surface". In: *USENIX Security'20*.

[161] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *CT-RSA'06*.

[162] Our World in Data. *Compute used to train notable AI systems, across domains*. https://ourworldindata.org/grapher/artificial-intelligence-training-computation. 2024.

[163] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. "Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical". In: *USENIX Security'21*.

[164] Marco Patrignani and Marco Guarnieri. "Exorcising Spectres with Secure Compilers". In: *CCS'21*.

[165] Kexin Pei et al. "Can Large Language Models Reason about Program Invariants?" In: *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*. Ed. by Andreas Krause et al. Vol. 202. Proceedings of Machine Learning Research. PMLR, 2023, pp. 27496–27520. URL: https://proceedings.mlr.press/v202/pei23a.html.

[166] Charles Sanders Peirce. *Collected papers of charles sanders peirce*. Vol. 5. Harvard University Press, 1974.

[167] Elizabeth Polgreen, Andrew Reynolds, and Sanjit A. Seshia. "Satisfiability and Synthesis Modulo Oracles". In: *VMCAI'22*. 2022.

[168] Elizabeth Polgreen et al. "UCLID5: Multi-modal Formal Modeling, Verification, and Synthesis". In: *CAV'22*. Springer.

[169] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. "Path Sensitive Inference of Function Precedence Protocols". In: *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 2007, pp. 240–250. DOI: 10.1109/ICSE.2007.63. URL: https://doi.org/10.1109/ICSE.2007.63.

[170] Ashay Rane, Calvin Lin, and Mohit Tiwari. "Raccoon: Closing Digital Side-Channels through Obfuscated Execution". In: *USENIX Security'15*.

[171] Ashay Rane, Calvin Lin, and Mohit Tiwari. "Secure, Precise, and Fast Floating-Point Operations on x86 Processors". In: *USENIX Security'16*. 2016.

[172] Stephen E Richardson. "Exploiting trivial and redundant computation". In: *ARITH'93*.

[173] riscv. *riscv-opcodes repository*. https://github.com/riscv/riscv-opcodes. [Online; accessed 18-Apr-2023]. 2023.

[174] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. "Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs". In: *CIKM '20*. ACM.

[175] Chirag Sakhuja et al. "Combining Branch History and Value History For Improved Value Prediction". In: *CVP-Championship Value Prediction* (2019).

[176] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. "ZeroTrace : Oblivious Memory Primitives from Intel SGX". In: *NDSS'18*.

[177] Stephan van Schaik et al. "RIDL: Rogue In-flight Data Load". In: *S&P*. May 2019.

[178] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic superoptimization". In: *ASPLOS'13*.

[179] Eric M. Schulte et al. "Post-compiler software optimization for reducing energy". In: *ASPLOS'14*.

[180] Michael Schwarz et al. "ZombieLoad: Cross-Privilege-Boundary Data Sampling". In: *CCS*. 2019.

[181] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A concolic unit testing engine for C". In: *ACM SIGSOFT Software Engineering Notes* 30.5 (2005), pp. 263–272.

[182] Sanjit A. Seshia. "Combining Induction, Deduction, and Structure for Verification and Synthesis". In: *Proceedings of the IEEE* 103.11 (2015), pp. 2036–2051.

[183] Sanjit A. Seshia. "Sciduction: Combining Induction, Deduction, and Structure for Verification and Synthesis". In: *Proceedings of the Design Automation Conference (DAC)*. June 2012, pp. 356–365.

[184] Sanjit A. Seshia and Pramod Subramanyan. "UCLID5: Integrating Modeling, Verification, Synthesis, and Learning". In: *Proceedings of the 15th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*. Oct. 2018.

[185] André Seznec. "Exploring value prediction with the eves predictor". In: *CVP*. 2018.

[186] Fahad Shaon et al. "SGX-BigMatrix: A Practical Encrypted Data Analytic Framework With Trusted Processors". In: *CCS'17*.

[187] Rahul Sharma, Aditya V Nori, and Alex Aiken. "Interpolants as classifiers". In: *International Conference on Computer Aided Verification*. Springer. 2012, pp. 71–87.

[188] Rahul Sharma et al. "Verification as learning geometric concepts". In: *Static Analysis: 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings 20*. Springer. 2013, pp. 388–411.

[189] Sharon Shoham et al. "Static Specification Mining Using Automata-Based Abstractions". In: *IEEE Trans. Software Eng.* 34.5 (2008), pp. 651–666. DOI: 10.1109/TSE.2008.63. URL: https://doi.org/10.1109/TSE.2008.63.

[190] Xujie Si et al. "Code2Inv: A Deep Learning Framework for Program Verification". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12225. Lecture Notes in Computer Science. Springer, 2020, pp. 151–164. DOI: 10.1007/978-3-030-53291-8_9. URL: https://doi.org/10.1007/978-3-030-53291-8%5C_9.

[191] A. Sodani and G.S. Sohi. "Understanding the Differences between Value Prediction and Instruction Reuse". In: *MICRO'98*.

[192] Avinash Sodani and Gurindar S. Sohi. "Dynamic Instruction Reuse". In: *ISCA'97*.

[193] Armando Solar-Lezama. "Program synthesis by sketching". PhD thesis. USA, 2008. ISBN: 9781109097450.

[194] Armando Solar-Lezama et al. "Combinatorial sketching for finite programs". In: *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 2006, pp. 404–415.

[195] Yuheng Su et al. "Extended CTG Generalization and Dynamic Adjustment of Generalization Strategies in IC3". In: *CoRR* abs/2501.02480 (2025). DOI: 10.48550/ARXIV.2501.02480. arXiv: 2501.02480. URL: https://doi.org/10.48550/arXiv.2501.02480.

[196] Yuheng Su et al. "The rIC3 Hardware Model Checker". In: *CoRR* abs/2502.13605 (2025). DOI: 10.48550/ARXIV.2502.13605. arXiv: 2502.13605. URL: https://doi.org/10.48550/arXiv.2502.13605.

[197] Chuyue Sun et al. "ClassInvGen: Class Invariant Synthesis using Large Language Models". In: *CoRR* abs/2502.18917 (2025). DOI: 10.48550/ARXIV.2502.18917. arXiv: 2502.18917. URL: https://doi.org/10.48550/arXiv.2502.18917.

[198] Qinhan Tan et al. "RTL Verification for Secure Speculation Using Contract Shadow Logic". In: *Arxiv'24*.

[199] Robert Endre Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM J. Comput.* (1972).

[200] Mohit Tiwari et al. "Complete Information Flow Tracking from the Gates Up". In: *ASPLOS'09*. 2009.

[201] Shruti Tople and Prateek Saxena. "On the Trade-Offs in Oblivious Execution Techniques". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Michalis Polychronakis and Michael Meier.

[202] Emina Torlak and Rastislav Bodik. "A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages". In: *PLDI'14*.

[203] Po-An Tsai et al. "Safecracker: Leaking Secrets through Compressed Caches". In: *ASPLOS'20*.

[204] ucb-bar. *V-Scale*. https://github.com/LGTMCU/vscale. [Online; accessed 18-Apr-2023]. 2023.

[205] Klaus V. Gleissenthall et al. "Solver-aided constant-time hardware verification". In: *CCS'21*.

[206] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Proceedings of the 2Nd Workshop on System Software for Trusted Execution*. SysTEX'17. 2017.

[207] Marco Vassena et al. "Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade". In: *POPL'21* ().

[208] Shobha Vasudevan et al. "GoldMine: Automatic assertion generation using data mining and static analysis". In: *Design, Automation and Test in Europe, DATE 2010, Dresden, Germany, March 8-12, 2010*. Ed. by Giovanni De Micheli et al. IEEE Computer Society, 2010, pp. 626–629. DOI: 10.1109/DATE.2010.5457129. URL: https://doi.org/10.1109/DATE.2010.5457129.

[209] Jose Vicarte et al. "Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data". In: *ISCA'21*.

[210] Jose Rodrigo Sanchez Vicarte et al. "Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest". In: *S&P'22*.

[211] Yakir Vizel et al. "IC3-flipping the E in ICE". In: *VMCAI'17*.

[212] Guanhua Wang et al. "oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis". In: ().

[213] S. Wang et al. "Exploiting narrow-width values for thermal-aware register file designs". In: *DATE'09*.

[214] Wenhao Wang et al. "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX". In: *CCS '17*.

[215] Yingchen Wang et al. "DVFS frequently leaks secrets: Hertzbleed attacks beyond SIKE, cryptography, and CPU-only data". In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2023, pp. 2306–2320.

[216] Yingchen Wang et al. "Gpu. zip: On the side-channel implications of hardware-based graphical data compression". In: *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2024, pp. 3716–3734.

[217] Yingchen Wang et al. "Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86". In: *USENIX Security'22*.

[218] Zilong Wang et al. "Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts". In: *CCS '23*.

[219] Zilong Wang et al. "Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts". In: *CCS'23*.

[220] Westley Weimer and George C. Necula. "Mining Temporal Specifications for Error Detection". In: *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Vol. 3440. Lecture Notes in Computer Science. Springer, 2005, pp. 461–476. DOI: 10.1007/978-3-540-31980-1_30. URL: https://doi.org/10.1007/978-3-540-31980-1%5C_30.

[221] Cheng Wen et al. "Enchanting Program Specification Synthesis by Large Language Models Using Static Analysis and Program Verification". In: *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part II*. Ed. by Arie Gurfinkel and Vijay Ganesh. Vol. 14682. Lecture Notes in Computer Science. Springer, 2024, pp. 302–328. DOI: 10.1007/978-3-031-65630-9_16. URL: https://doi.org/10.1007/978-3-031-65630-9%5C_16.

[222] John Whaley, Michael C. Martin, and Monica S. Lam. "Automatic extraction of object-oriented component interfaces". In: *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*. Ed. by Phyllis G. Frankl. ACM, 2002, pp. 218–228. DOI: 10.1145/566172.566212. URL: https://doi.org/10.1145/566172.566212.

[223] Claire Wolf. *Yosys Open SYnthesis Suite*. https://yosyshq.net/yosys/. Accessed: 2024-12-14.

[224] Guangyuan Wu et al. "LLM Meets Bounded Model Checking: Neuro-symbolic Loop Invariant Inference". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*. Ed. by Vladimir Filkov, Baishakhi Ray, and Minghui Zhou. ACM, 2024, pp. 406–417. DOI: 10.1145/3691620.3695014. URL: https://doi.org/10.1145/3691620.3695014.

[225] Haoze Wu, Clark W. Barrett, and Nina Narodytska. "Lemur: Integrating Large Language Models in Automated Program Verification". In: *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL: https://openreview.net/forum?id=Q3YaCghZNt.

[226] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems". In: *S&P'15*. 2015.

[227] Jinlin Yang et al. "Perracotta: mining temporal API rules from imperfect traces". In: *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. Ed. by Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa. ACM, 2006, pp. 282–291. DOI: 10.1145/1134285.1134325. URL: https://doi.org/10.1145/1134285.1134325.

[228] Jianan Yao et al. "DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols". In: *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 2021, pp. 405–421. ISBN: 978-1-939133-22-9.

[229] Jianan Yao et al. "DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols". In: *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*. Ed. by Marcos K. Aguilera and Hakim Weatherspoon. USENIX Association, 2022, pp. 485–501. URL: https://www.usenix.org/conference/osdi22/presentation/yao.

[230] Jianan Yao et al. "Learning nonlinear loop invariants with gated continuous logic networks". In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 106–120. DOI: 10.1145/3385412.3385986. URL: https://doi.org/10.1145/3385412.3385986.

[231] Yuval Yarom and Katrina Falkner. "Flush+Reload: A high resolution, low noise, L3 cache side-channel attack". In: *USENIX Security'14*.

[232] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA". In: *Journal of Cryptographic Engineering* 7 (2017), pp. 99–112.

[233] Joshua J Yi and David J Lilja. "Improving processor performance by simplifying and bypassing trivial computations". In: *ICCD'02*.

[234] Jiyong Yu et al. "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing". In: *NDSS'19*.

[235] Jiyong Yu et al. "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data". In: *MICRO'19*.

[236] Drew Zagieboylo et al. "Specverilog: Adapting information flow control for secure speculation". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 2068–2082.

[237] Michał Zalewski. *AFL: American Fuzzy Lop*. https://lcamtuf.coredump.cx/afl/. [Online; accessed 18-Apr-2023]. 2023.

[238] Danfeng Zhang et al. "A Hardware Design Language for Timing-Sensitive Information-Flow Security". In: *SIGPLAN Not.* 50.4 (Mar. 2015), pp. 503–516. ISSN: 0362-1340. DOI: 10.1145/2775054.2694372. URL: http://doi.acm.org/10.1145/2775054.2694372.

[239] Zhiyuan Zhang et al. “Ultimate SLH: Taking Speculative Load Hardening to the Next Level”. In: *USENIX Security’23*.

[240] Jerry Zhao et al. “SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine”. In: (2020).

[241] Zirui Neil Zhao et al. “Binoculars: Contention-Based Side-Channel Attacks Exploiting the Page Walker”. In: *USENIX Security’22*.

[242] Wenting Zheng et al. “Opaque: An Oblivious and Encrypted Distributed Analytics Platform”. In: *NSDI’17*.

[243] He Zhu, Stephen Magill, and Suresh Jagannathan. “A data-driven CHC solver”. In: *PLDI’18*.