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

Sushant Dinesh
University of Illinois Urbana-Champaign
sdinesh2@illinois.edu

Madhusudan Parthasarathy
University of Illinois Urbana-Champaign
madhu@illinois.edu

Christopher W. Fletcher
University of Illinois Urbana-Champaign
cwfletch@illinois.edu

***Abstract*—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 paper 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.

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 [1]. To comprehensively mitigate these attacks, a multitude of software and hardware defenses (some commercialized [2], [3], [4]) 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) [5], [6], [7], [8], [9], [10], [11], [12], [13].* 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 [5], [6], [8], [14], [15].

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 [16], [17], [18], [19].

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 automati-

cally constructs relational invariants using a combination of symbolic analysis and invariant learning, inspired from techniques in program synthesis [20], [21], [22], [23], [24]. 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 [23] (which enhances the famous Houdini algorithm [25]), 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 design. Second, we show how CONJUNCT enables microarchitects to reason about the safety of proposed microarchitectural optimizations. Specifically, we implement two computation reuse schemes [26], an exemplar advanced microarchitectural optimization, on top of the Ibex core. We use CONJUNCT to confirm a hypothesis made in Vicarte et. al. [16]: 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.

In summary, we make 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.

The open-source release of CONJUNCT can be found here: https://github.com/FPSG-UIUC/conjunct.

2. Background and Motivation

There is a rich literature on how programs interacting with hardware resources (e.g., the cache [27], [28], [29], TLB [30], branch predictor [31], [32], functional unit execution ports [33], [34], [35], complex arithmetic instructions [36], [37], [38], speculative execution [39] and others [40]) 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 [28], [30], [41], [42], [43]. Beyond memory instructions, several other instruction types (namely branch instructions [5], [44] and specific complex arithmetic operations [36]) are well-known transmitters. This has led to a line of hardware and software defenses [5], [6], [7], [8], [44], [45], [46], [47], [48], [49] (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 [5] but

also the more recent speculative side channels [7], [9], [39], [45]. For example, Spectre attacks [39] 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 [11], [12], [45], [50].

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 [16], [18] to optimize their common case behavior. Vicarte and Deng et al. [16], [18] describe several families of optimizations that fit this mold:

- Computation simplification / elimination optimizations (e.g., [51], [52], [53]) 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., [26], [54], [55]) memoize computation when the same instruction(s) are executed twice with the same operands.
- Value prediction (e.g., [56], [57], [58]) saves cycles when an instruction returns a predictable result.
- Significance compression (e.g., [59], [60], [61]) impacts performance depending on the position of the high-order on bit in each program word.
- Silent stores (e.g., [62], [63], [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. [16] describes how:

- The above optimizations are seldom implemented in the processor's 'Execute' stage/ALUs. For example, even computation simplification [51], 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 [59] 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 [62] 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. §3-§5 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, §3 defines the problem. As summarized in §1, the analysis itself is broken into two phases. First, we perform a bounded analysis over a fixed number of cycles (§4) 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 (§5) and prove safety of instructions for an unbounded number of cycles.

3. 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 3.1. 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 3.2. 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 $(opcode, \overrightarrow{pv})$ 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 $\rightsquigarrow$: $\mathbb{S} \times \Sigma \rightarrow \mathbb{S}$ 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 $a \in \Sigma$.

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

Definition 3.3. 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 $s \restriction O$ denote the projection of the state s onto O.

Definition 3.4. 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 $j \in [0, n]$, $s_j \restriction O \neq s'_j \restriction O$.

Definition 3.5. 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 3.6. 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 (§4) 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 §5, 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. 3.6. 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 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 §5 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.

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	$\mathbf{S} \restriction O$	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 1: A summary of all definitions used in CONJUNCT.

4. Phase 1: Bounded Analysis

In this section, we build on the terminology defined in §3 to develop our symbolic execution-based analysis on hardware designs described, e.g., in Verilog. For convenience, terms defined are summarized in Table 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 $P = a_{IUT} \parallel P_s$, where P_s is any possible suffix program.[2]

We augment the definition of state s from Def. 3.1 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.

2. Although we write P as having a suffix but not prefix program, the application of P in §4.1 will be equivalent to considering P with an arbitrary prefix program.

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 [65], [66] used for checking two-safety properties such as non-interference and is equivalent to a miter circuit used in the architecture community [67]. 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. 3.6 is formalized in §6.1.) We will refer to the (symbolic unconstrained) secret data released by the register file as D_{secret}.

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}.

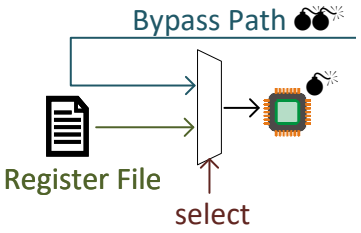

Figure 1: Multiple sources for operand values. Operand values for instructions may either arrive from the register file or along the bypass path and is selected by the control variable to the MUX, `select`. Each 💣 represents a potential source of unsafety.

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

This does not prevent a_{IUT} from forwarding a function of D_{secret} to later instructions. We address this in §4.1.1.

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 1)? Can this miss cases where *multiple* a_{IUT} need to receive secret data and interact for unsafety to manifest? We prove in §6.1 that our complete analysis is sound and handles these cases, while requiring only the above specified annotation burden.

4.1. Symbolic Execution

Algorithm 1: Symbolic Execution-based analysis.

```
   Data: O, K, a_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;
```

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 (§5).

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, a_{IUT}: the $a_{IUT} \in \Sigma$ representing the instruction-under-test (IUT), and P_s: the suffix program to execute after the IUT, and returns either Safe indicating that a_{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 $\mathbf{S}$ are *symbolic and constrained to be equal* on the left and right executions of the design. See §4 "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 a_{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.

4.1.1. 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 a_{IUT} through architectural data dependencies with a_{IUT}, microarchitectural state set by a_{IUT}, etc. This creates a blame attribution problem: if a safety violation occurs due to the execution of a_{IUT} followed by some instruction a', should a_{IUT} be deemed unsafe, or should a'? Suppose a' *is actually* unsafe. In that case, we risk incorrectly blaming a_{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 a_{IUT} (as the only source of unconstrained data is from the register read on behalf of a_{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 `nops` 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 a_{IUT} and one or more instructions a' in P_s. In this case, we make a conservative assumption and treat both a_{IUT} and a' (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 a_{IUT} and a *specific* a' is unsafe, but leave this for future work. The implications of this in the precision of the overall analysis is discussed in §6.2.

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. 3.6, but only for the bounded K cycles.

4.2. 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.

5. Phase 2: Invariant Learning

The bounded analysis (§4) 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 $\mathbf{H}$,

$$\mathbf{S} \models \mathbf{H} \implies Safe(\mathbf{S}) \tag{1}$$

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

$$(\mathbf{S} \models \mathbf{H}) \wedge (\mathbf{S} \rightsquigarrow^{a} \mathbf{S}') \implies \mathbf{S}' \models \mathbf{H} \quad \forall a \in \widehat{\Sigma^{+}_{(ii)}} \qquad (2)$$

An $\mathbf{H}$-state is any state $\mathbf{S}$ that satisfies $\mathbf{H}$. Therefore, any $\mathbf{H}$-state is *Safe*. Together, Equation 1 and Equation 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 1 and Equation 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 [23]—and as we later show in §7.4 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 [23]. SORCAR describes a theoretical framework for learning conjunctive invariants from examples, which is an improvement over the well-known Houdini algorithm [25], 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 §6.2). We also need to specify/generate the positive/negative samples to bootstrap the learning algorithm (see §4) and we need to find mechanisms to recover from failure when an invariant is not found (see §5.5).

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 $\mathbf{H}$. 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 §5.1, we describe an invariant grammar that is sufficiently rich to enable analysis of several recent microarchitectures (e.g., the pipelined RISC-V RocketChip [68]). We then describe in §5.3 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.

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

$\langle p \rangle$ ::= 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: 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 $\mathbb{V}$.

Tolerating unsafe instructions. Proving that a set of instructions is safe using an invariant $\mathbf{H}$ has further benefits. In particular, we can allow *unsafe* instructions to execute in states satisfying $\mathbf{H}$ provided that results in states that remain in $\mathbf{H}$ (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 §5.4.

5.1. Language for H

We use the DSL shown in Figure 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 [69]. 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 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 $\mathsf{Impl}(v, vs)$ 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 $\mathbb{V}$.

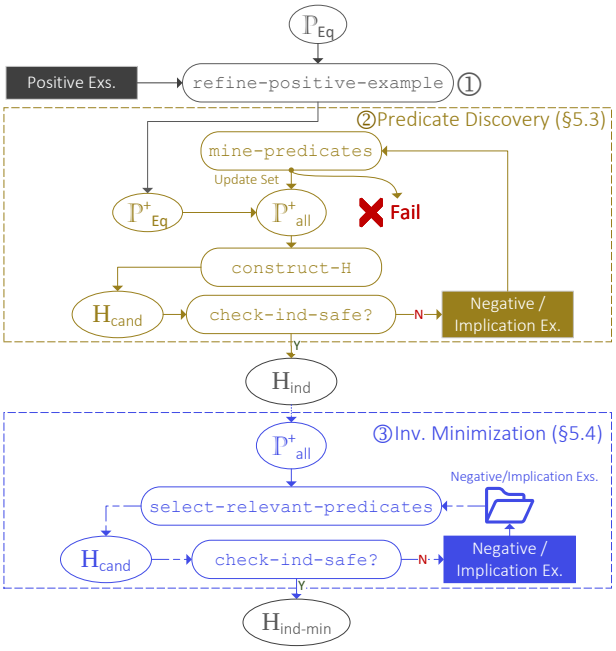

Figure 3: Workflow diagram showing learning of the inductive invariant $\mathbf{H}$. `construct-H` is a procedure that generates an invariant $\mathbf{H}_{cand}$ by taking a conjunction over $\mathbb{P}^{+}_{all}$.

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, §5.3 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 [70], [71], [72] (including the microarchitectures we studied) and also admit an efficient analysis.

5.2. 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 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 1) and inductivity (Equation 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 (§4). Initially, the set of implication examples is 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 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 §5.5 and present the details of predicate discovery next in §5.3.

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 §5.4.

4. As discussed in §5, 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.

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.

5.3. 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.

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
```

5.4. 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 instruction may also be safe. Second, it helps experts analyze and understand the root cause of unsafety in their designs, as we show in §7.4. 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 (§5.3). 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 5.1. 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 [73]: 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 [74] 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.

5.5. 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 §7 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.

6. Proof Sketches

In this section we will provide proof sketches for CONJUNCT. First, we will refine Def. 3.6 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 §6.1). Finally, we show that the invariant $\mathbf{H}_{ind}$ synthesized by CONJUNCT is precise (in §6.2).

6.1. CONJUNCT Proof of Soundness

First, we will instantiate Def. 3.6 for the constant-time programming setting. Recall, Def. 3.6 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 6.1. 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]

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.

It should be clear that by the definition of SISP that CT-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 $\widehat{\Sigma^+}$. Appendix A provides a proof that $\widehat{\Sigma^+}$ is valid in CT-SISP(O) for the specified O.

6.2. 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. 3.2), we define a product machine that operates over a pair of states.

Definition 6.2. 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 6.3. 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 §4 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 §4 may be imprecise when attributing blame to an instruction (§4.1.1) 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 §4.1.1, and so $\widehat{\Sigma^+} = \Sigma^+$, and the derived invariant is indeed precise.

7. 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.

7.1. 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 [22], 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 [75] 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 [76] as the SMT solver in all our experiments. We obtained the open-source designs from their official repositories and processed them through yosys [77], 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) 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.

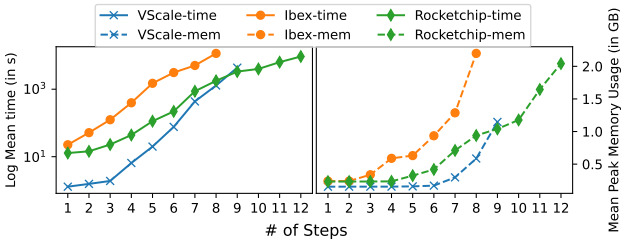

Figure 4: Performance of Bounded Analysis (§4). Steps correspond to clock cycles.

All three pipelines (*V-Scale* [78], *Ibex* [79], *RocketChip* [68]) 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.

7.2. Quantitative Evaluation

	# Pipeline			# Annotations				
	Stages	# Regs	# Wires	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: 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.

	Predicate Discovery				Greedy			
	Eq	Impl	Total	Time	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 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.

Performance of ConjunCT. We now evaluate ConjunCT in terms of analysis time and memory usage, as a function of the design complexity. Figure 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 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 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. 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 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 3. For all three designs, we invoked predicate discovery (§5.3) 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 (§5.4).

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 (§5.4), 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.

7.3. Security Properties / Security Evaluation

Next, we analyze the set of safe and unsafe instructions as identified by ConjunCT. Table 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 (§8), 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]

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 (§7.1), so there are no cache-based attacks (§2) 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.

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

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

Figure 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.

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).

7.4. 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.

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 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 (§5.1). 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.

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

	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 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.

- (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 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 (§5.3) 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 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 (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.

7.5. Case-Study: Computation Reuse

Lastly, we show how ConjunCT can help microarchitects evaluate the security impacts of their proposed optimizations (echoing §2). 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 [26]. This is an interesting optimization to study as it is typically implemented as a part of the pipeline's instruction decode (ID) logic [16], [26]. Hence, it illustrates the need to audit the entire pipeline rather than just the execution units.

Sodani et al. [26] 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. [26] find that both of the above schemes improve performance. Even more interestingly, as noted by Vicarte et al. [16], 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 [79], which consists of instruction fetch (IF) and in-

struction 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 (§4) 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 (§5.2) 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.

8. 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** (§5) and **R2** (§6.2). Predicate discovery (§5.3) and our approach using synthesis (§4-§5) empirically enables analysis with very few expert guidance/annotations (§7). 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 [80], 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 §4: 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 [81]). 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 [82] (and its follow-on Xenon [69]) 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 [83] 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 (§4) can be viewed as inferring *likely contracts*, i.e., the set of safe/unsafe instructions. Our Phase 2 (§5.2) then proves that the inferred likely contracts are actual contracts. This, when the problem at hand is determining the safe instruction set (Def. 3.6), 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 [84], [85], [86], [87], fuzzing [88], [89], [90], [91], 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.

9. 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).

Acknowledgments. We thank the anonymous reviewers for their valuable feedback. This research was partially funded by NSF grants 1954521, 1942888 and 2154183, as well as by Intel through the RARE center.

References

[1] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware," *IACR'16*.

[2] "Arm Architecture Registers Armv8, for Armv8-A architecture profile," https://developer.arm.com/docs/ddi0595/latest/aarch32-system-registers/cpsr.

[3] "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.

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

[5] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *CRYPTO'96*.

[6] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing," in *NDSS'19*.

[7] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data," in *MICRO'19*.

[8] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan, "FaCT: A DSL for Timing-Sensitive Computation," in *PLDI'19*.

[9] J. Fustos, F. Farshchi, and H. Yun, "SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks," *DAC'19*.

[10] S. Dinesh, G. Garrett-Grossman, and C. W. Fletcher, "SynthCT: Towards Portable Constant-Time Code," in *NDSS'22*.

[11] M. Patrignani and M. Guarnieri, "Exorcising Spectres with Secure Compilers," in *CCS'21*.

[12] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom, "Ultimate SLH: Taking Speculative Load Hardening to the Next Level," in *USENIX Security'23*.

[13] N. Mosier, H. Lachnitt, H. Nemati, and C. Trippel, "Axiomatic Hardware-Software Contracts for Security," in *ISCA'22*.

[14] D. J. Bernstein, "djbsort," https://sorting.cr.yp.to/.

[15] "ChaCha20 (BearSSL)," https://bearssl.org/gitweb/.

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

[17] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, "Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest," in *S&P'22*.

[18] S. Deng, N. Matyunin, W. Xiong, S. Katzenbeisser, and J. Szefer, "Evaluation of Cache Attacks on Arm Processors and Secure Caches," *IEEE Transactions on Computers*, vol. 71, pp. 2248–2262, 2022.

[19] T. Downs, "Hardware Store Elimination," https://travisdowns.github.io/blog/2020/05/13/intel-zero-opt.html, 2020, accessed on 17.06.2020.

[20] R. Alur, R. Bodík, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madhusudan, M. M. K. Martin, M. Raghothaman, S. Saha, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-Guided Synthesis," in *Dependable Software Systems Engineering*, 2015, vol. 40, pp. 1–25.

[21] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama, "Search-based program synthesis," *Commun. ACM*, vol. 61, pp. 84–93, 2018.

[22] E. Torlak and R. Bodik, "A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages," in *PLDI'14*.

[23] D. Neider, S. Saha, P. Garg, and P. Madhusudan, "Sorcar: Property-Driven Algorithms for Learning Conjunctive Invariants," in *SAS*, ser. Lecture Notes in Computer Science, vol. 11822, 2019, pp. 323–346.

[24] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A robust framework for learning invariants," in *CAV'14*.

[25] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *FME'01*.

[26] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse," in *ISCA'97*.

[27] Y. Yarom, D. Genkin, and N. Heninger, "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA," *Journal of Cryptographic Engineering*, vol. 7, pp. 99–112, 2017.

[28] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *CT-RSA'06*.

[29] Y. Yarom and K. Falkner, "Flush+Reload: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security'14*.

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

[31] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *MICRO*, 2016.

[32] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Understanding and Mitigating Covert Channels Through Branch Predictors," *TACO'16*.

[33] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port Contention for Fun and Profit," in *S&P*, 2019.

[34] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting Speculative Execution through Port Contention," in *CCS'19*.

[35] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures," in *NDSS'20*.

[36] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On Subnormal Floating Point and Abnormal Timing," in *S&P'15*.

[37] J. Großschädl, E. Oswald, D. Page, and M. Tunstall, "Side-Channel Analysis of Cryptographic Software via Early-Terminating Multiplications," in *ICISC'09*.

[38] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors," in *S&P'09*.

[39] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P'19*.

[40] D. Evtyushkin and D. Ponomarev, "Covert Channels through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations," in *CCS'16*.

[41] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Binoculars: Contention-Based Side-Channel Attacks Exploiting the Page Walker," in *USENIX Security'22*.

[42] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, "MemJam: A False Dependency Attack Against Constant-Time Crypto Implementations," *International Journal of Parallel Programming*, vol. 47, pp. 538–570, 2019.

[43] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU," in *NDSS'17*.

[44] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks," *IACR'05*.

[45] M. Vassena, C. Disselkoen, K. v. Gleissenthall, S. Cauligi, R. G. Kıcı, R. Jhala, D. Tullsen, and D. Stefan, "Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade," *POPL'21*.

[46] D. J. Bernstein, "The Poly1305-AES Message-Authentication Code," in *FSE'05*.

[47] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing Digital Side-Channels through Obfuscated Execution," in *USENIX Security'15*.

[48] B. A. Fisch, D. Vinayagamurthy, D. Boneh, and S. Gorbunov, "Iron: Functional Encryption using Intel SGX," in *CCS'17*.

[49] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An Oblivious and Encrypted Distributed Analytics Platform," in *NSDI'17*.

[50] R. Choudhary, A. Wang, Z. N. Zhao, A. Morrison, and C. Fletcher, "DECLASSIFLOW: A Static Analysis for Modeling Non-Speculative Knowledge to Relax Speculative Execution Security Measures," in *CCS'23*.

[51] S. E. Richardson, "Exploiting trivial and redundant computation," in *ARITH'93*.

[52] J. J. Yi and D. J. Lilja, "Improving processor performance by simplifying and bypassing trivial computations," in *ICCD'02*.

[53] E. Atoofian and A. Baniasadi, "Improving energy-efficiency by bypassing trivial computations," in *IPDPS'05*.

[54] A. Sodani and G. Sohi, "Understanding the Differences between Value Prediction and Instruction Reuse," in *MICRO'98*.

[55] C. Molina, A. González, and J. Tubella, "Dynamic Removal of Redundant Computations," in *ICS'99*.

[56] S. Mittal, "A survey of value prediction techniques for leveraging value locality," *CCPE'17*, vol. 29, no. 21, p. e4250, 2017.

[57] C. Sakhuja, A. Subramanian, P. Joshi, A. Jain, and C. Lin, "Combining Branch History and Value History For Improved Value Prediction," *CVP-Championship Value Prediction*, 2019.

[58] A. Seznec, "Exploring value prediction with the eves predictor," in *CVP-Championship Value Prediction*, 2018.

[59] D. Brooks and M. Martonosi, "Dynamically exploiting narrow width operands to improve processor power and performance," in *HPCA'99*.

[60] R. Canal, A. González, and J. E. Smith, "Very low power pipelines using significance compression," in *MICRO'00*.

[61] S. Wang, J. Hu, S. G. Ziavras, and S. W. Chung, "Exploiting narrow-width values for thermal-aware register file designs," in *DATE'09*.

[62] K. Lepak and M. Lipasti, "Silent Stores for Free," in *MICRO'00*.

[63] I. Kim and M. H. Lipasti, "Implementing Optimizations at Decode Time," in *ISCA'02*.

[64] T. Downs, "Hardware Store Elimination," https://travisdowns.github.io/blog/2020/05/13/intel-zero-opt.html, 2020.

[65] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying Constant-Time Implementations," in *USENIX Security'16*.

[66] G. Barthe, P. R. D'argenio, and T. Rezk, "Secure information flow by self-composition," *Mathematical Structures in Computer Science*, vol. 21, no. 6, pp. 1207–1252, 2011.

[67] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz, "A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors," in *DAC'20*.

[68] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, 2016.

[69] K. v. Gleissenthall, R. G. Kıcı, D. Stefan, and R. Jhala, "Solver-Aided Constant-Time Hardware Verification," in *CCS'21*.

[70] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "GPUVerify: a verifier for GPU kernels," in *OOPSLA'12*.

[71] A. Lal and S. Qadeer, "Powering the static driver verifier using corral," in *FSE'14*.

[72] K. L. McMillan, "Interpolation and SAT-based model checking," in *CAV'03*.

[73] R. M. Karp, "Reducibility Among Combinatorial Problems," in *Proceedings of a symposium on the Complexity of Computer Computations*, ser. The IBM Research Symposia Series, R. E. Miller and J. W. Thatcher, Eds., 1972, pp. 85–103.

[74] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of operations research*, vol. 4, no. 3, pp. 233–235, 1979.

[75] riscv, "riscv-opcodes repository," https://github.com/riscv/riscv-opcodes, 2023, [Online; accessed 18-Apr-2023].

[76] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A Versatile and Industrial-Strength SMT Solver," in *TACAS*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243, 2022, pp. 415–442.

[77] C. Wolf, "Yosys Open SYnthesis Suite," https://yosyshq.net/yosys/.

[78] ucb bar, "V-Scale," https://github.com/LGTMCU/vscale, 2023, [Online; accessed 18-Apr-2023].

[79] lowRISC, "Ibex," https://github.com/lowRISC/ibex, 2023, [Online; accessed 18-Apr-2023].

[80] L. Deutschmann, J. Müller, M. R. Fadiheh, D. Stoffel, and W. Kunz, "Towards a Formally Verified Hardware Root-of-Trust for Data-Oblivious Computing," in *DAC'22*.

[81] L. Deutschmann, J. Mueller, M. R. Fadiheh, D. Stoffel, and W. Kunz, "A scalable formal verification methodology for data-oblivious hardware," *arXiv'23*.

[82] K. v. Gleissenthall, R. G. Kıcı, D. Stefan, and R. Jhala, "IODINE: Verifying Constant-Time Execution of Hardware," in *USENIX Security'19*.

[83] Z. Wang, G. Mohr, K. von Gleissenthall, J. Reineke, and M. Guarnieri, "Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts," in *CCS'23*.

[84] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[85] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *PLDI'05*.

[86] R. Majumdar and K. Sen, "Hybrid concolic testing," in *ICSE'07*.

[87] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI'08*.

[88] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *CCS'17*.

[89] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *PLDI'08*.

[90] M. Zalewski, "AFL: American Fuzzy Lop," https://lcamtuf.coredump.cx/afl/, 2023, [Online; accessed 18-Apr-2023].

[91] P. Godefroid, "Fuzzing: Hack, Art, and Science," *Commun. ACM*, vol. 63, p. 70–76, 2020.

Appendix A.
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. 6.1).

For the purposes of the proof, consider CONJUNCT to be composed of two black boxes, corresponding to Phase 1 (§4) and Phase 2 (§5) 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 A.1. The set of safe instructions output as $\widehat{\Sigma^+}$ by CONJUNCT satisfies CT-SISP(O) (Def. 6.1).

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. 6.1.

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 1), out-of-order/speculative execution, etc. In all cases, the annotation burden needed to specify R to satisfy Def. 6.1 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 A.1 holds for all Phase 1 implementations.

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

The overarching goal of this paper is to identify instructions in an ISA that are safe in an unbounded setting with respect to microarchitectural timing attacks. To prove safety in the unbounded setting, the paper proposes ConjunCT for learning inductive invariants from RTL code with minimal developer annotations. Using ConjunCT, the work identified inductive invariants for three systems, which helped demonstrate the safety of certain instructions and ISA optimizations of three implementations of RISC-V architecture in a reasonable amount of time.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) The paper presents a highly automated and principled approach for identifying safe and unsafe instructions in an ISA represented in RTL with minimal developer annotations.
2) The paper presents a Domain-Specific Language (DSL) for expressing invariants along with an automated inductive invariant learning approach called CONJUNCT, which runs on top of SORCAR/Houdini and is crucial in formally proving instruction safety in an unbounded setting.
3) The effectiveness of CONJUNCT has been demonstrated in three implementations of the RISC-V architecture in which it was able to learn the necessary inductive invariants necessary to discharge the proof of unbounded safety of different instructions.