H-Houdini: Scalable Invariant Learning

Sushant Dinesh
University of California, Berkeley
Berkeley, USA
sushantd@berkeley.edu

Yongye Zhu
University of California, Berkeley
Berkeley, USA
yongye.zhu@berkeley.edu

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

Abstract

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

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

We instantiate H-Houdini as VeloCT, a framework that proves hardware security properties by learning relational invariants. We benchmark VeloCT on the 'safe instruction set synthesis' problem in microarchitectural security. Here, VeloCT automatically (with no expert annotations) learns an invariant for the RISC-V Rocketchip in under 10s (2880× faster than state of the art). Further, VeloCT is the first work to scale to the RISC-V out-of-order BOOM and can (mostly-automatically) verify all BOOM variants (ranging from Small to Mega) in between 6.95 minutes to 199.1 minutes.

CCS Concepts: • **Security and privacy** → **Logic and verification**; **Side-channel analysis and countermeasures**; • **Hardware** → *Hardware validation.*

Keywords: Hardware Verification; Invariant Learning; Scalable Verification; Formal Verification; Abductive Reasoning; Microarchitectural Security; Constant-Time Programming; Incremental Invariant Learning; Hardware attacks and defenses;

ASPLOS '25, March 30–April 3, 2025, Rotterdam, Netherlands.

ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707263

ACM Reference Format:
Sushant Dinesh, Yongye Zhu, and Christopher W. Fletcher. 2025. H-Houdini: Scalable Invariant Learning. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3669940.3707263

1 Introduction

Formal verification is a well-recognized bottleneck in hardware design today. Further, one can expect this bottleneck to worsen into the future as design complexity further increases, design time cycles further decrease [35], and the set of properties to be verified further increases (e.g., due to the rise of hardware security vulnerabilities [36, 38]).

At a high level, automated verification procedures consist of an interplay between two components: (i) an algorithm that proposes invariants or partial invariants to prove a property, and (ii) off-the-shelf verification engines that check the proposed invariants. While the past several decades have seen remarkable progress in this direction (e.g., [8, 22, 28, 32, 34, 39, 40]), it still faces scalability challenges when attempting to automatically learn invariants for and verify large designs.

This paper considers scalability challenges for a popular verification paradigm called Machine Learning Inspired Invariant Synthesis (MLIS) [12, 23, 24, 27, 28, 30, 40, 53]. MLIS approaches, such as the famous Houdini [28] algorithm and Sorcar [40], frame invariant learning as an interaction between a learner and a teacher. The learner attempts to learn an invariant using examples provided by the teacher.

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

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

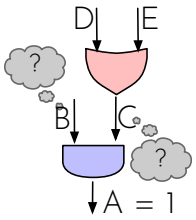
By contrast, SAT-based approaches, such as IC3/PDR [8, 22], assume the system is white box and can exploit problem structure to perform much cheaper, 'relative' checks that incrementally eliminate bad states. This results in cheaper SMT queries, but without the aforementioned other benefits of MLIS algorithms.

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

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

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

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



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

Application to secure hardware verification: VeloCT. We instantiate H-Houdini to solve an important problem in hardware security called the *safe instruction set synthesis problem (SISP)* [17, 19, 20, 26, 29]. We call the resulting analysis tool VeloCT.

Verifying the SISP enables portable and secure constant-time programming on modern processors. Constant-time programming is a ubiquitous paradigm for writing code that is safe from timing attacks [1, 7, 11, 37, 51]. The idea is to write a program using only 'safe' instructions, where a safe instruction's execution time does not depend on its operands. Then, by composition, the program's execution time is not a function of its inputs. Unfortunately, this paradigm is difficult to follow today because processors may each employ different software-invisible optimizations that turn different sets of instructions into operand timing-variable instructions. For example, a multiply instruction may be equipped with a 'skip-on-0' feature. Safe instruction synthesis addresses this issue by analyzing input RTL and determining (synthesizing) the set of safe instructions given that RTL.

Prior work [20] on automated invariant learning for the SISP is based on the MLIS algorithm Sorcar (which improves Houdini's performance by making it property directed). It thus faces the scalability issues described earlier. Specifically, the authors report that learning an inductive invariant for the open-source (in-order) RISC-V Rocketchip core took 8 hours, and were unable to scale to out-of-order cores such as RISC-V BOOM. We experimentally verified that Sorcar-style (monolithic) SMT queries did not scale to BOOM.

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

To summarize, we make the following contributions:

1. We propose H-Houdini, a scalable invariant learning algorithm that enables MLIS to use incremental, parallel SMT checks.
2. We propose VeloCT, an instance of H-Houdini that enables scalable safe instruction set synthesis given RTL as input.
3. We evaluate VeloCT on open-source processors. On the in-order Rocketchip core (10K state bits), we learn

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

VeloCT is open source, and can be found at [21].

2 Background & Motivation

2.1 Terminology

We now formally setup the invariant synthesis problem.

Definition 2.1. Transition System (*TS*): We denote a transition system *TS* by a 3-tuple $(\mathbb{S}, T, s_0)$; where $\mathbb{S}$ is the set of all states, $T : \mathbb{S} \times \mathbb{S}$ is the transition relation, and $s_0 \in \mathbb{S}$ is a special *initial* state. Let each state $s \in \mathbb{S}$ be made up of state variables with identifiers in $\mathbb{V}$.

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

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

Definition 2.2. Inductive Invariant (H): **H** is an inductive invariant for a property **P** on *TS* if the following three conditions hold. (i) *Initiation:* $\mathbf{H}(s_0)$, (ii) *Consecution:* $\forall\, s, s' \in \mathbb{S}$: $\mathbf{H}(s) \wedge T(s, s') \implies \mathbf{H}(s')$, and finally, (iii) **P** *holds:* $\forall\, s \in \mathbb{S}$: $\mathbf{H}(s) \implies \mathbf{P}(s)$.

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

Definition 2.3. Invariant Synthesis Problem: Given a transition system *TS* and a property **P**, synthesize an inductive invariant **H** to prove **P** or return None to indicate that no such **H** exists.

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

Definition 2.4. Relative Inductivity: We say **H** is relatively inductive to **G** if $\mathbf{G} \wedge \mathbf{H} \implies \mathbf{H}'$. Notice that if we set **G** to *true*, then relative inductivity reduces to the well-known monolithic inductive query. These relative inductive queries are a crucial part in making SAT-based invariant learning algorithms like IC3/AVR incremental.

Definition 2.5. Abduct (A): Consider a formula of the form $\mathbf{H} \implies \mathbf{H}'$ that does not hold initially. We define an abduct **A**, the result of an *abductive query* [42], as a formula that "fixes" the above implication, i.e., $\mathbf{A} \wedge \mathbf{H} \implies \mathbf{H}'$ holds and **A** is *non-contradictory* with **H**.

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

2.2 Machine Learning Inspired Invariant Synthesis

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

2.2.1 Houdini overview Next, we review the most popular MLIS algorithm: Houdini [28]. Houdini takes as inputs the set of all *predicates* $\mathbb{P}$, a set of positive examples $\mathcal{E}$, and a property of interest **P** and outputs either an invariant **H** if successful, or None if it fails. A predicate p is a formula in first-order logic over a subset of variables $\mathbb{V}_p \subseteq \mathbb{V}$. Given a state s, p will evaluate either evaluate to `True`, denoted by $s \models p$ or `False` denoted by $s \not\models p$.

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

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

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

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

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

Note, we can augment the original Houdini algorithm to take **P** as an input and check to see if the inductive invariant satisfies **P** (returning None if not).[1] Sorcar improves Houdini by making it *property directed* where the property is checked throughout invariant synthesis and used to further prune the search space.

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

These characteristics prevent Houdini from scaling to our problems of interest. Specifically, in our experience and as supported by prior work [17], even a single SMT query over predicates spanning the BOOM microarchitecture is prohibitively expensive and beyond the capabilities of current automated verification tools.

3 H-Houdini

We now describe our invariant learning algorithm, H-Houdini[2]. In a nutshell, H-Houdini replaces Houdini's sequential monolithic inductivity checks with a "wave" of property-directed, incremental, memoizable and parallelizable inductivity checks. These attributes result in better efficiency and scalability.

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

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

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

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

The recursion creates a wavefront of proof obligations that resembles a depth-first search (DFS) over the design's state elements (predicates). If an abduct for some p forces us to include a predicate p' that makes the invariant not inductive, we backtrack and ask for a different abduct for p (that does not include p'). If one does not exist, we backtrack further. Like a normal DFS, once we 'visit' each predicate once (either prove that including it either can or does not lead to an inductive invariant), we need not recurse through it again. Once the wavefront terminates, the hierarchy of abducts are combined to form the overall invariant. This can be done without ever performing a monolithic inductivity check, by construction of the composition of abducts.

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

We now describe the above more formally, proceeding as follows: §3.1 argues that the main premise in H-Houdini is sound; §3.2 presents the H-Houdini algorithm in detail; Appendix A provides proof sketches of soundness and completeness.

3.1 Hierarchical Decomposition is Sound

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

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

$$\mathbf{H}_1 \wedge \mathbf{H}_0 \implies \mathbf{H}_0' \tag{1}$$

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

$$\mathbf{H}_2 \wedge \mathbf{H}_1 \implies \mathbf{H}_1' \tag{2}$$

Let us assume that $\mathbf{H}_0$ and $\mathbf{H}_2$ are non-contradictory, i.e., $\exists x, x \models \mathbf{H}_0 \wedge \mathbf{H}_2$. Then, from Equation 1 and Equation 2, it

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

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

follows:

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

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

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

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

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

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

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

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

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

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

3.2 H-Houdini Algorithm

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

See algorithm 1. The algorithm has inputs similar to the original Houdini but with two differences. First, the predicate universe $\mathbb{P}$ is replaced by an oracle $O_{mine}^{\mathcal{A},\mathcal{E}}$ which is instantiated with the set of positive examples $\mathcal{E}$ and optionally expert annotations $\mathcal{A}$. Second, as H-Houdini is white box, it takes in the transition system TS. Note, p_{target} is analogous to $\mathbf{P}$ (§3). H-Houdini outputs an inductive invariant ($\mathbf{H}$) that proves p_{target}, or None if no such invariant exists.

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

First, O_{slice}^{TS} (the slicing oracle) takes as input p_{target} and outputs the set of state elements $\mathbb{V}_{slice}$ that influence the inductivity of p_{target} in one step of TS. When verifying sequential circuits (as we do in this paper), these are the state elements in the 1-step cone-of-influence (COI) for p_{target}.[3] That is, in our AND gate example (§3), if p_{target} is over the output register, $\mathbb{V}_{slice}$ is the set of input registers.

Second, $O_{mine}^{\mathcal{A},\mathcal{E}}$ (the mining oracle) translates $\mathbb{V}_{slice}$ into a set of predicates $\mathbb{P}_{\mathbb{V}}$ that will be considered when synthesizing abducts. This takes into account the user-specified predicate language, the positive examples $\mathcal{E}$ and (optionally) additional expert annotations ($\mathcal{A}$).[4]

Third, O_{abduct} (the abduction oracle) takes $\mathbb{P}_{\mathbb{V}}$ and attempts to synthesize an abduct $\mathbf{A}$ out of a subset of $\mathbb{P}_{\mathbb{V}}$ for p_{target} (line 12). If multiple abducts are found, one is returned; if O_{abduct} is called with the same p_{target} multiple times, we

Algorithm 1: The H-Houdini algorithm.

Input : $O_{mine}^{\mathcal{A},\mathcal{E}}$: The predicate mining oracle, p_{target}: Target predicate/property, TS: Transition system
Output: $\mathbf{H}$: invariant that proves p_{target} or None

```
1  ℙ_fail = ∅;
2  def H-Houdini (O^{A,E}_mine, p_target, TS) → None/H:
3      if p_target is memoized and soln ∩ ℙ_fail = ∅ then
4          return solution to p_target;
5      end
6      valid-solution = False;
7      while not valid-solution do
8          H = p_target;
9          𝕍_slice = O^{TS}_slice(p_target);
10         ℙ_𝕍 = O^{A,E}_mine(p_target, 𝕍_slice);
11         ℙ_𝕍 = ℙ_𝕍 \ ℙ_fail;
12         A = O_abduct(p_target, ℙ_𝕍);
13         set A as the memoized solution to p_target;
14         if A is None then
15             return None;
16         end
17         valid-solution = True;
18         for p in A do
19             H_sol = H-Houdini (O^{A,E}_mine, p, TS);
20             if H_sol is None then
21                 valid-solution = False;
22                 ℙ_fail = ℙ_fail ∪ {p};
23                 break;
24             end
25             H = H ∧ H_sol
26         end
27     end
28     return H;
29 end
```

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

[4] The mining oracle we describe here is deterministic. Our implementation adopts an incremental variant that returns steadily larger subsets of $\mathbb{P}_{\mathbb{V}}$ over multiple calls, to encourage smaller abducts. See §3.2.3.

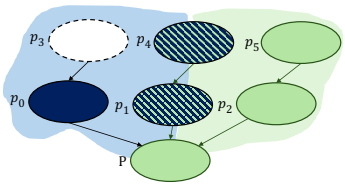


Figure 1. Dependency graph between predicates in a TS and $\mathbb{P}$. Each node is a predicate p_i, and edge $p_i \rightarrow p_j$ means p_i is in the 1-step COI of p_j. The hollowed out predicate p_3 does not hold inductively and cannot be included in an invariant. The figure shows that there are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

require a different abduct be returned each call (as in an iterator). If no abduct is found (line 15), the algorithm returns `None`, indicating that no solution exists. In our implementation, each abduct returned costs an SMT query. More details for O_{abduct} are given in §3.2.3.

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

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

Suppose O_{abduct} first returns the blue solution. Then H-Houdini fires off on subtrees p_0 and p_1. Eventually, because p_3 fails to be inductive, H-Houdini discovers that p_0 cannot hold inductively and therefore backtracks to synthesize a new solution for **P**. The next query to O_{abduct} for **P** returns the green solution as p_0 is no longer available. But the work we need to do to ratify the green solution is significantly reduced since we memoized the solution for p_1. The only additional work to do to complete the invariant is to synthesize the solution for the nodes colored green: p_2 and p_5. In this way, the algorithm only backtracks partially. H-Houdini only squashes the path of failure while the rest of the synthesized solutions may be reused.

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

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

3.2.3 Details for Abduction Oracle $O_{abduct}(p_{target}, \mathbb{P}_V)$ O_{abduct} takes a target predicate p_{target} and a set of predicates ($\mathbb{P}_V$) and outputs an abduct **A**, a conjunction over a subset of $\mathbb{P}_V$ that shows p_{target} is 1-step inductive. Formally: $\mathbf{A} \implies p'_{target}$. Such an **A** can be efficiently computed using Craig's interpolants [15, 39]. We compute the interpolants quickly using the following (to our knowledge, novel) SMT query:[5]

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

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

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

3.2.4 Parallelism The serial implementation of H-Houdini lends itself to a high degree of parallelism. For example, the loop that recursively checks each predicate in the abduct (line 18-line 25) can be performed in parallel. We parallelize this loop in our implementation and evaluation. More aggressive forms of speculative parallelism, e.g., trying different abducts in parallel, are possible. We leave more detailed investigation of these to future work.

[5] We tried other methods, e.g., cvc5's native O_{abduct} call. Our method was fastest in practice.

3.2.5 Formal requirements for soundness and completeness

H-Houdini requires any downstream analysis to satisfy two contracts for soundness and completeness. We present details for these in Appendix A.

4 VeloCT: Preliminaries

The next three sections describe VeloCT, which instantiates H-Houdini to solve the safe instruction set synthesis problem (SISP) [17, 19, 20, 29]. This section gives terminology (taken mostly from [20]); §5 describes the analysis' design in detail; Appendix C describes a worked example of the analysis on a simple microarchitecture.

In VeloCT, the transition system TS is the hardware circuit described in §2. We also introduce a set of inputs Σ to TS, input during transitions, to be either ISA instructions or the special null input ϵ indicating 'no instruction' (following [20]). Next, we define the main concepts in the SISP.

Definition 4.1. Trace (π): A trace of TS over a sequence of inputs starting at state s_0 is a sequence of states in TS obtained by applying the transition relation T ($\rightsquigarrow$): $s_0 \rightsquigarrow^{a_0} s_1 \ldots \rightsquigarrow^{a_k} s_{k+1}$ where $a_i \in \Sigma$ denotes a_i is input during $\rightsquigarrow$.

Definition 4.2. Trace Indistinguishability: Assume that an attacker can observe a subset of elements $O \subseteq \mathbb{V}$. For state s, the values of O (a projection on s) are given by $s \downharpoonright O$. We say two traces $\pi = s_0 s_1 \ldots$, $\pi' = s'_0 s'_1 \ldots$, are *trace indistinguishable* if $|\pi| = |\pi'|$ and $\forall_j\ j \in [0, |\pi|]$, $s_j \downharpoonright O = s'_j \downharpoonright O$ where $|\pi|$ denotes the length of π.

Definition 4.3. Equal-modulo-secret I-states: Let us define a subset of state elements to hold secret values: $\mathbb{V}_{sec} \subseteq \mathbb{V}$. Two states s and s' are equal-modulo-secret I-states if $s \approx^{sec} s_0 \approx^{sec} s'$ where s_0 is the init state of TS and $\approx^{sec}$ is an equality relation over the projection of states excluding secrets, i.e., over $\mathbb{V}_{pub} = \mathbb{V} \setminus \mathbb{V}_{sec}$.

Definition 4.4. Safe Set ($\widehat{\Sigma^+}$): A set of instructions form a safe set $\widehat{\Sigma^+}$ if, for every sequence of instructions x over $\widehat{\Sigma^+}$, and a pair of equal-modulo-secret I-states (s^1, s^r), the pair of traces, (π^1, π^r) of TS starting from (s^1, s^r) and given x are trace indistinguishable.

Lastly, we define the *soundness* and *precision* of **H** in verifying safe set $\widehat{\Sigma^+}$. Since **H** is a relational invariant to prove the non-interference property from above, it is defined over a pair of states. For simplicity, we construct a product state s over a product variable space $\mathbb{V} = \mathbb{V}^1 \cup \mathbb{V}^r$, where $\mathbb{V}^x$ denotes that each identifier is renamed by adding a prefix x. We say $s = (s_i^1 \cdot s_i^r)$ to denote the composition of s^1, s^r to form the product state s.

Definition 4.5. Admits Traces ($\bowtie^{\mathbf{H}}$): Consider $\pi^1 = s_0^1, s_1^1, \ldots$ and $\pi^r = s_0^r, s_1^r, \ldots$. For simplicity assume the shorter trace is padded with special null states ($\varnothing$). Next, construct product states, s_i, between corresponding states in π^1 and π^r, i.e., $\forall_{i \in |\pi^1|} s_i = (s_i^1 \cdot s_i^r)$. Lastly, define a relation $\bowtie^{\mathbf{H}}$: $\pi \times \pi$ that relates the pair (π^1, π^r) if $\forall_{i \in |\pi^1|} s_i \models \mathbf{H}$.

Definition 4.6. Soundness of H: **H** is sound if for all pairs of traces π^1, π^r that are *not* trace indistinguisible, $(\pi^1, \pi^r) \notin \bowtie^{\mathbf{H}}$.

Definition 4.7. Precision of H: **H** is precise for a safe set $\widehat{\Sigma^+}$ if for all sequences of instructions x over $\widehat{\Sigma^+}$ and starting with all pairs of equal-modulo-secret I-states, the pairs of generated traces $(\pi^1, \pi^r) \in \bowtie^{\mathbf{H}}$.

Definition 4.8. Positive Example ($e \in \mathcal{E}$): Each example is a product state $s = (s_i^1 \cdot s^r)$ of two states s^1 and s^r. A positive example, w.r.t. a property **P**, is an example which satisfies the property ($s \models \mathbf{P}$) *and* on application of the transition function T leads to either the terminating state or another positive example.

5 VeloCT

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

VeloCT implements H-Houdini for invariant learning and starts learning from **P**. For simplicity assume O consists of a single state element (call it v_o). Taking inspiration from Def. 4.2, we need to prove that the values of v_o^1 and v_o^r are always equal. This property is expressed formally in terms of an Eq-type predicate (defined in the next section) as: $\mathsf{Eq}(v_o^1, v_o^r)$.

In what follows, we first define the predicate language (§5.1.1) and the implementation of the O_{mine} (§5.1.2). Then, we discuss how VeloCT generates examples in §5.2. Lastly, we conclude the section with a proof that **H** from VeloCT is sound (Def. 4.6) and precise (Def. 4.7) in Appendix B.

5.1 Predicate Language and Mining

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

5.1.1 Predicate Language

We start with the predicate language proposed by ConjunCT [20]. We observe that for real-world processor designs we can actually simplify the predicate language. To prove a set of instructions form a safe set as defined in Def. 4.3, we do not need the expressiveness offered by Impl-type predicate introduced in ConjunCT. Rather, we can include a simpler predicate that restricts a register to hold certain safe values(s). With this intuition, VeloCT implements three *base* types of predicates:

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

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

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

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

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

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

5.1.2 O_{mine} : Predicate Mining

The predicate mining algorithm is based on a key insight: positive examples contain the *core of safety*, a skeleton that represents the *necessary* and *sufficient* conditions for an execution to be safe. The primary challenge is then to learn an invariant strong enough to prove safety and weak enough to generalize to safe executions beyond positive examples $\mathcal{E}$ (the precision definition Def. 4.7).

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

Even expert-provided predicates are validated against $\mathcal{E}$ before they are added to the final set. This validation ensures that experts can freely propose any predicates without risking unsoundness. We concretely discuss the expert predicates needed for the verification of BOOM in §6.2.

In §B.1 we show how this implementation of O_{mine} is sound and complete when using H-Houdini. Next, we discuss how to generate these positive examples for learning, and conclude by showing that it is sufficient to derive a precise invariant.

Algorithm 2: $O^{\mathcal{A},\mathcal{E}}_{mine}$: Predicate mining algorithm.

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

1 **def** $O^{\mathcal{A},\mathcal{E}}_{mine}(p_{target}, \mathbb{V}_{slice}) \rightarrow \mathbb{P}_{\mathbb{V}}$:
2 $\quad \mathbb{V}_{Eq} = \{v \mid v \in \mathbb{V}_{slice}$ and $\forall_{e \in \mathcal{E}}\ e[v^l] = e[v^r]\}$;
3 $\quad \mathbb{P}_{Eq} = \varnothing$;
4 $\quad$ **for** v *in* $\mathbb{V}_{Eq}$ **do**
5 $\quad\quad \mathbb{P}_{Eq} = \mathbb{P}_{Eq} \cup$ Eq(v^l, v^r);
6 $\quad\quad$ **if** $\exists\ c\ \forall_{e \in \mathcal{E}}\ e[v] = c$ **then**
7 $\quad\quad\quad \mathbb{P}_{Eq} = \mathbb{P}_{Eq} \cup$ EqConst(v^l, c);
8 $\quad\quad$ **end**
9 $\quad\quad p_{safe} =$ InSafeSet(v^l);
10 $\quad\quad$ **if** $\forall_{e \in \mathcal{E}}\ e \models p_{safe}$ **then**
11 $\quad\quad\quad \mathbb{P}_{Eq} = \mathbb{P}_{Eq} \cup p_{safe}$;
12 $\quad\quad$ **end**
13 $\quad$ **end**
14 $\quad \mathbb{P}^*_{expert} = \mathcal{A}_{mine}(\mathbb{V}_{Eq})$;
15 $\quad \mathbb{P}_{expert} = \{p \mid p \in \mathbb{P}^*_{expert}$ and $\forall_{e \in \mathcal{E}}\ e \models p\}$;
16 $\quad \mathbb{P}_{\mathbb{V}} = \mathbb{P}_{Eq} \cup \mathbb{P}_{expert}$;
17 $\quad$ **return** $\mathbb{P}_{\mathbb{V}}$;
18 **end**

5.2 Example Generation

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

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

Each pair of traces yields multiple positive examples, one per state in the product trace, as defined in Def. 4.8. Specifically, we extract a positive example for each pair of states where the safe instruction under analysis is in flight in the trace.

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

Constructing clean positive examples is non-trivial. To run a safe instruction on the pipeline, our infrastructure executes start-up and tear-down code containing unsafe instructions. This code may make an example *dirty* for two reasons. First, if an unsafe instruction(s) is in flight in the same cycle as

the safe instruction. Second, if an unsafe instruction (not concurrently in flight) leaves a residue in the pipeline that is still present when the safe instruction executes.

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

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

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

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

We introduce example masking to clean up such dirty traces. VeloCT uses annotations to identify valid bits in key structures, e.g., issue/reorder buffer units. Then, VeloCT pre-processes positive examples to set entries to their reset values if the entry is marked as invalid. We give a detailed description of the annotations necessary for example masking on BOOM in §6.2. Note that this was only required for complex OoO cores such as BOOM. Simple in-order cores like Rocketchip do not require example masking or annotations.

Since VeloCT is the first effort to scale invariant learning to a large OoO like BOOM, we focus on demonstrating feasibility rather than perfecting the approach. In the future, we aim to explore other alternatives: e.g., a richer predicate language that includes Impl-type predicates from ConjunCT to conditionally constrain Uop to allow only SAFE Uops only when the entry is valid.

6 Evaluation

In this section, we evaluate VeloCT on real-world processor designs. First, §6.3 evaluates analysis efficiency and scalability. Second, §6.4 evaluates security.

6.1 Implementation & Evaluation Setup

We implement VeloCT in ~ 2300 lines Python. The current implementation accepts hardware designs and assertions for safety in the btor [41] format. The current implementation implements algorithm 1 and parallelizes each iteration of the inner loop (as described in line 18).

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

Table 1. Overview of evaluated design, their complexity (in # of state bits, in simulation/pre-synthesis), and learned invariant sizes (# of predicates).

Evaluation Setup. We ran our evaluation on a machine equipped with Dual Intel®Xeon®Gold 6148 CPUs (2 sockets, 40 virtual cores/socket), 256GB of memory and Ubuntu 20.04 LTS (kernel version 5.4.0). We also evaluated on an Anyscale Ray cluster (configured with compute-optimized C6i EC2 instances) [4]. Lastly, all code was evaluated on Python 3.8 and with CVC5 v1.1.12 as our choice of SMT solver.

Descriptions of Evaluated Designs. We evaluate VeloCT on Rocketchip [5] and all four supported variants of BOOM [52]: SmallBOOM, MediumBOOM, LargeBOOM, MegaBOOM. All targets were compiled using the standard configuration with the Chipyard [3] workflow. We added additional annotations in Chisel to flatten all modules in the generated Verilog. Finally, we use yosys [50] to create a miter (product) circuit, add assertion(s), and emit the output in the btor format [41] for VeloCT to consume.

For all designs, we instantiated the top-level module to be the Core, e.g., RocketCore and BOOMCore. See Table 1. This does not include the L1 caches. Since memory instructions are known to be unsafe, and no other instructions interact with these structures, caches can be ignored. Our analysis could certainly include them if desired.

Baselines. We compare performance/scalability to the state-of-the-art in safe instruction set synthesis ConjunCT [20]. ConjunCT's analysis is based on Houdini/Sorcar and can learn an invariant (Phase II of the ConjunCT analysis) in ~ 8 hours; it was unable to evaluate on any BOOM variant due to the cost of monolithic SMT queries. We compare security / synthesized safe sets to ConjunCT and UPEC [17]. The latter hand crafts (does not learn) and checks an invariant for MediumBOOM.

6.2 Expert Annotation Efforts

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

Annotations to Augment Predicate Set. Recall from §5.1.1, our current implementation does not automatically insert/mine `EqConstSet` predicates. We augmented O_{mine} to emit these as expert annotations in two circumstances. First, BOOM decodes RISC-V instructions into micro-ops (uops). Therefore, to constrain executions to only consist of instructions from the safe set, we introduce `InSafeUop` to complement the `InSafeSet` predicate. The `InSafeUop` predicate allows state elements to only take constants that correspond to safe uops. Second, we manually constrain the ALUOpcode signal to correspond to the proposed safe set.

Annotations for Example Masking. Example masking (§5.2.1) is used to ensure that the positive examples are clean. To do this, we identify and annotate various bits in the design that semantically hold `valid` bits and pair each `valid` bit with its corresponding entries. Concretely, we identified the valid bits for the following microarchitectural structures: ALU issue buffer, CSR issue buffer, FPU issue buffer, JMP unit, and rs2 operand type bits. When the valid bit is semantically 0, we reset the value of its corresponding entries. This step took a grad student (unfamiliar with BOOM) less than a day to find all the required annotations for one design, and another day to generalize to larger BOOM variants.

6.3 VeloCT: Performance Evaluation

We now evaluate for performance and scalability.

Scalability and Parallelism. Figure 2 shows invariant search time for each design, scaling the number of available cores. We do not evaluate larger designs on smaller core counts due to time constraints and only evaluate LargeBOOM and MegaBOOM on 160 cores. The main takeaway is that doubling the core count proportionally improves analysis time up to a point, which provides an estimate of the parallel span (i.e., time given infinite cores). It is noteworthy that the span increases with design size. This indicates that larger designs will benefit from yet additional parallelism. We note that we were able to generate an invariant for Rocketchip in under 10 seconds, representing a roughly 2880× speedup compared to ConjunCT.

Figure 3 performs a deeper dive, comparing the time given a fixed 80 cores (on our local cluster) with the time given an "infinite" number of cores (on the Anyscale cluster, which provides more cores on demand). We do not evaluate Rocketchip on Anyscale as 80 cores is sufficient. As expected, time given infinite cores drops relative to the time given 80 cores proportionally with design size. Especially interesting, the time given infinite cores shows that the analysis time scales proportionally to the cubic of the design size.

Median Task and SMT Time / Task. We now explore the extent to which the above results can improve further. The fundamental cost in the analysis is the time to perform SMT queries. Thus, it is important to understand what percentage of CPU cycles are going towards SMT queries. This is shown in Figure 4, which compares SMT time to overall task time (i.e., all time spent in the H-Houdini function body). We see that non-SMT activities account for about 50% of the time, even for larger designs, indicating room to improve performance with better engineering. As expected, the time per SMT query grows with design size (which influences the complexity of the average query) and, interestingly, grows linearly with size across the larger designs. We note, the above only shows median SMT query time yet we consistently saw long tails. For example, the 95th and 99th percentile for time-per-task for MegaBOOM was 565.94s and 1000.92s respectively, while the longest task took 9310s.

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

Table 2. Safe instruction sets synthesized by VeloCT.

Total Number of SMT Queries. Orthogonally, we can improve analysis time by reducing the total number of SMT queries that need to be performed. A task performs multiple SMT queries only when backtracking occurs. Thus, we show frequency of backtracking relative to the total number of tasks in Figure 5. The main takeaway is that the number of backtracks is relatively small but non-negligible, and accounts for roughly the same % of tasks in each design. If the set of positive examples was exhaustive, the number of backtracks would be 0.

6.4 VeloCT: Security Evaluation

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

We show the set of safe instructions that we verified for each target in Table 2. To summarize, we verified that most non-memory or control-flow instructions are safe. Our result for Rocketchip matches ConjunCT, except that we found `mul`-family instructions to be unsafe. We verified this manually as correct, and occurs because we evaluated on RV64 (whereas ConjunCT evaluated on RV32). We also monolithically verified the correctness of Rocketchip invariant. Our result for BOOM matches Kunz et al. [17] for all non-FP/non-CSR type instructions (which we categorized manually as unsafe). Interestingly, `mul`-family instructions are safe on BOOM. On BOOM, we were unable to verify the safety of the `auipc` instruction. On talking with the BOOM developers, we found out that even though the latency of `auipc` is supposed to be operand-independent, the instruction indeed has variable timing behavior. We leave investigating why

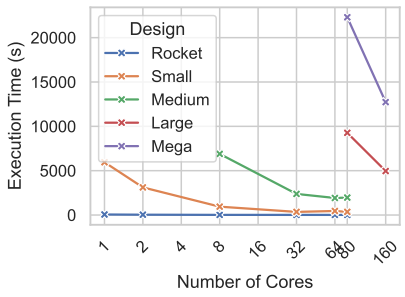


Figure 2. Execution Time of VeloCT (in s) vs. # of Parallel Cores on designs of various sizes. VeloCT's execution time consistently halves on doubling the number of cores until a saturation point is reached.

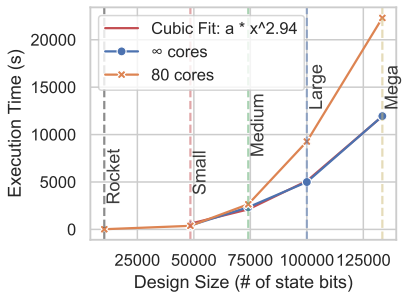


Figure 3. Execution Time (in s) of VeloCT vs. Design Size. Plot shows results for an 80-core evaluation and for ∞ cores (on an Anyscale cluster). VeloCT scaling with ∞ cores shows cubic growth.

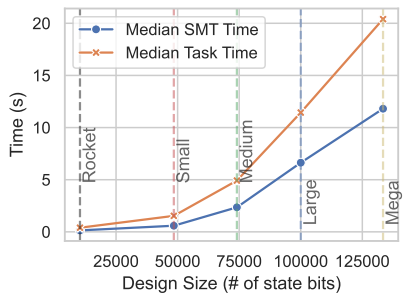


Figure 4. Median SMT Query Time and Task Time (in s) vs. Design Size (# of state bits). The time per SMT query scales linearly with the design size.

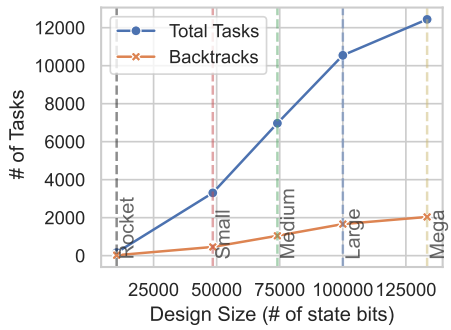


Figure 5. # of Tasks and # of Instances of Backtracking vs. Design Size (# of bits). Number of tasks grows linearly with design size. Number of tasks backtracked is relatively low indicating good coverage from positive examples.

this is, and whether an alternate implementation of `auipc` could be made safe, to future work.

7 Related Work

SAT-based Invariant Learning & IC3. SAT-based methods for learning invariants have been explored in depth over the years, including abstract interpretation [14], interpolants [39], predicate abstractions [6], abductive reasoning [18], constraint solving [33], incremental learning [8, 22] and its variants [34]. More recently, IC3 implementations at the word-level [13, 32] have also been studied. H-Houdini and IC3 are similar in that they both use relative induction to incrementally learn invariants, but their approaches differ significantly. IC3 relies on negative examples to generalize counterexamples and strengthen invariants, which is challenging in practice. In contrast, H-Houdini combines mining positive examples with abductive reasoning to build invariants incrementally. Additionally, H-Houdini is fully parallel at the predicate level, while IC3 uses parallelism to explore multiple inductive strengthenings, requiring synchronization and communication between tasks.

MLIS and ICE Learning. MLIS techniques [12, 23, 24, 27, 28, 30, 40, 53] have a long history in invariant learning, starting with DAIKON [23] to find likely invariants from traces and Houdini [28] which is an algorithm to find inductive invariants for unannotated programs. The MLIS setting has strong connections to synthesis in other domains [2, 43, 44]. Garg et al. [30] first introduced ICE-learning which was later generalized and successfully applied to learn invariants in various settings [12, 24]. The main drawback with ICE-learning techniques is the need for monolithic inductive queries, which do not scale well with a large predicate language and to large systems under verification like BOOM. H-Houdini solves this using relative inductive queries, making it a RICE-learner (*Relative* ICE) as introduced by Vizel et al. [48]. To the best of our knowledge, H-Houdini is the first MLIS-based incremental invariant learning algorithm developed in the RICE framework.

Abductive Reasoning in Program Analysis. Abductive reasoning, introduced by Pierce [42], has been applied in program analysis to infer missing preconditions [31] and to scale shape analysis for large software through bi-abduction [9, 10]. Dillig et al. [18] used it to synthesize inductive invariants for loops, but their approach lacks completeness guarantees, cannot exploit hierarchy or parallelism, and requires extensive backtracking. In contrast, H-Houdini offers strong completeness guarantees, utilizes hierarchical and parallel processing and reduces backtracking by leveraging positive examples.

Hardware Constant-Time Verification. Several recent projects focus on verifying constant-time hardware [16, 17, 25, 26, 46, 47] and constant-time contracts [20, 45, 49]. But, none of these existing works scale to large OoO cores such as BOOM—which is a major contribution of our work.

8 Discussion & Conclusion

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

While we expect H-Houdini to generalize beyond the SISP, more research is needed to make it completely "push button" and capable of verifying arbitrary properties. We see two synergistic categories of future work. First, work that further improves automation. That is, while H-Houdini can efficiently and automatically search for an invariant, it requires the user to specify a predicate abstraction, implement O_{mine} and specify a means to create positive examples. Nominally, these steps should also be automated. Second, work that demonstrates H-Houdini on verification tasks beyond the SISP. We are confident that H-Houdini will apply largely "out of the box" to other 2-safety properties, such as other forms of traditional non-interference. Beyond 2-safety, developing extensions to N-safety properties (common in security verification), liveness properties, LTL-style properties and functional correctness checking are all exciting directions for future study.

9 Acknowledgments

We thank the anonymous reviewers and our shepherd for their excellent feedback. This work was partially funded by NSF grants CNS-1954521, CNS-1942888, CNS-2154183 and CCF-8191902; as well as by gifts from Intel and Apple.

References

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

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD'13*.

[3] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, 2020.

[4] Anyscale. Anyscale homepage. https://www.anyscale.com/. Accessed: 2024-06-23.

[5] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016.

[6] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI'01*.

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

[8] Aaron R Bradley. SAT-based model checking without unrolling. In *VMCAI'11*.

[9] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL'09*.

[10] Cristiano Calcagno, Dino Distefano, and Viktor Vafeiadis. Bi-abductive resource invariant synthesis. In *APLAS'09*.

[11] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. FaCT: A DSL for Timing-Sensitive Computation. In *PLDI'19*.

[12] Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. Ice-based refinement type discovery for higher-order functional programs. *Journal of Automated Reasoning*, 2020.

[13] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Infinite-state invariant checking with IC3 and predicate abstraction. *Formal Methods Syst. Des.*, 2016.

[14] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*.

[15] William Craig. Linear reasoning. A new form of the herbrand-gentzen theorem. *J. Symb. Log.*, 1957.

[16] Lucas Deutschmann, Johannes Müller, Mohammad R. Fadiheh, Dominik Stoffel, and Wolfgang Kunz. Towards a Formally Verified Hardware Root-of-Trust for Data-Oblivious Computing. In *DAC'22*.

[17] Lucas Deutschmann, Johannes Müller, Mohammad R Fadiheh, Dominik Stoffel, and Wolfgang Kunz. A scalable formal verification methodology for data-oblivious hardware. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[18] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. Inductive invariant generation via abductive inference. In *OOPSLA'13*.

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

[20] Sushant Dinesh, Madhusudan Parthasarathy, and Christopher Fletcher. CONJUNCT: Learning inductive invariants to prove unbounded instruction safety against microarchitectural timing attacks. In *S&P'24*.

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

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

[23] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE'00*.

[24] P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P. Madhusudan. Horn-ICE learning for synthesizing invariants and contracts. In *OOPSLA'18*.

[25] Mohammad Rahmani Fadiheh, Dominik Stoffel, Clark W. Barrett, Subhasish Mitra, and Wolfgang Kunz. Processor hardware security vulnerabilities and their detection by unique program execution checking. In *DATE'19*.

[26] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Müller, Jörg Bormann, Sayak Ray, Jason M. Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. An exhaustive approach to detecting transient execution side channels in RTL designs of processors. *IEEE Transactions on Computers*, 2023.

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

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

[29] Michael Flanders, Reshabh K. Sharma, Alexandra E. Michael, Dan Grossman, and David Kohlbrenner. Avoiding instruction-centric microarchitectural timing channels via binary-code transformations. In

ASPLOS'14.
[30] Pranav Garg, Christof Löding, Parthasarathy Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *CAV'14*.
[31] Roberto Giacobazzi. Abductive analysis of modular logic programs. In *ILPS'94*.
[32] Aman Goel and Karem A. Sakallah. AVR: abstractly verifying reachability. In *TACAS'20*.
[33] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI'08*.
[34] Alexander Ivrii and Arie Gurfinkel. Pushing to the top. In *FMCAD'15*.
[35] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *ISCA '17*.
[36] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P'19*.
[37] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO'96*.
[38] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security'18*.
[39] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *CAV'03*.
[40] Daniel Neider, Shambwaditya Saha, Pranav Garg, and P. Madhusudan. SORCAR: Property-Driven Algorithms for Learning Conjunctive Invariants. In *SAS'19*.
[41] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2, btormc and boolector 3.0. In *CAV'18*.
[42] Charles Sanders Peirce. *Collected papers of charles sanders peirce*, volume 5. Harvard University Press, 1974.
[43] Armando Solar-Lezama. *Program synthesis by sketching*. PhD thesis, USA, 2008.
[44] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS'06*.
[45] Qinhan Tan, Yuheng Yang, Thomas Bourgeat, Sharad Malik, and Mengjia Yan. RTL verification for secure speculation using contract shadow logic. In *Arxiv'24*.
[46] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. IODINE: Verifying Constant-Time Execution of Hardware. In *USENIX Security'19*.
[47] Klaus V. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. Solver-aided constant-time hardware verification. In *CCS'21*.
[48] Yakir Vizel, Arie Gurfinkel, Sharon Shoham, and Sharad Malik. IC3-flipping the E in ICE. In *VMCAI'17*.
[49] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. Specification and verification of side-channel security for open-source processors via leakage contracts. In *CCS'23*.
[50] Claire Wolf. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/. Accessed: 2024-12-14.
[51] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *NDSS'19*.
[52] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. May 2020.
[53] He Zhu, Stephen Magill, and Suresh Jagannathan. A data-driven CHC solver. In *PLDI'18*.

A H-Houdini Analysis

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

A.1 Contracts

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

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

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

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

A.2 Soundness

We argue that H-Houdini is a concrete instance of the incremental learner from §3.1. The initial property we aim to prove, $\mathbf{H}_0$, serves as the starting point. Each incremental step $\mathbf{H}_i$, synthesized as an abduct, is formed by a conjunction of a subset of predicates from $\mathbb{P}_{\mathbb{V}}$. That is, let $\mathbf{H}_i = \bigwedge_k p_k$. Let $\mathbf{A}^k_i$ be the result of O_{abduct} on p_k. Then, by Contract 1, $\mathbf{H}_{i+1} = \bigwedge_k \mathbf{A}^k_i$, formed by taking conjunctions over abducts for p_k. Finally, by Contract 2, H-Houdini satisfies the premise for soundness (P-S).

A.3 Completeness

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

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

B Soundness & Precision of VeloCT

B.1 O_{mine} satisfies H-Houdini Contracts

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

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

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

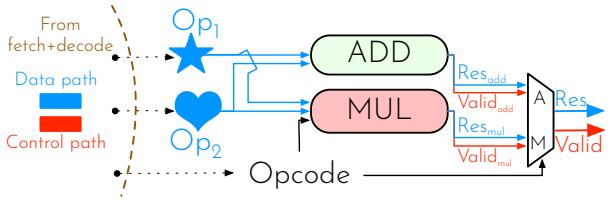
Finally, we show the precision (Def. 4.7) and soundness (Def. 4.6) of $\mathbf{H}$ obtained from VeloCT.

B.1.1 H is Sound This is trivially true: $\mathbf{P} = \texttt{Eq}(v^{\texttt{l}}_o, v^{\texttt{r}}_o)$ is a part of $\mathbf{H}$ and is inductive. Therefore, any state that eventually leads to violating $\mathbf{P}$ will not be allowed by $\mathbf{H}$. This proves $\mathbf{H}$ is sound (Def. 4.6).

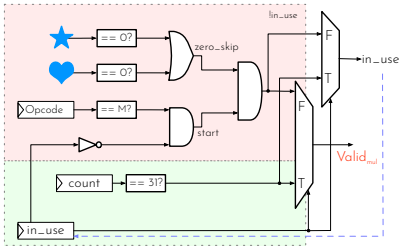
B.1.2 H is Precise We break this proof up into three parts.
(I): For any instruction allowed by $\mathbf{H}$, $\mathbf{H}$ *allows all its executions, i.e., with any operand value.* Every $e \in \mathcal{E}$ forces the secret values in the register file to be unequal on l and r. Therefore, we cannot add an Eq, or further, EqConst or EqConstSet, on instruction operand values. Therefore, we cannot restrict an instruction's operand values. (I) follows.
(II): $\mathbf{H}$ *allows all executions of every safe instruction.* We have at least one example per safe instruction. Therefore, $\mathbf{H}$ needs to allow at least one execution of every safe instruction to be consistent with $\mathcal{E}$. Further, from (I), $\mathbf{H}$ will allow all executions from every safe instruction. (II) follows.
(III): $\mathbf{H}$ *allows compositions of safe instructions.* From (II), as $\mathbf{H} = \bigwedge_i p_i$, every p_i also satisfies (II). Consider p_0, p_1, p_2 over state elements v_0, v_1, v_2 respectively. Further, let the topology be such that v_0 is some composition of v_1, v_2. Now, as p_1, p_2 allow all executions of all safe instructions, v_1 and v_2 are allowed to independently take values that occur in 2 different safe executions. Consequently, for p_0 to be inductive, p_0 should allow v_0 to take any value that occurs in the composition of allowed safe executions. Since p_0 was an arbitrary choice, this argument holds for all p_i. Since every p_i allows all compositions of all safe instructions, (III) follows. Since (III) is equivalent to Def. 4.7, $\mathbf{H}$ is precise.

C H-Houdini Running Example

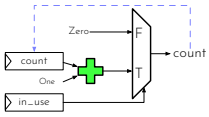
In this section, we will demonstrate how H-Houdini works for the VeloCT setting using the example circuit shown in Figure 6 (a). The figure shows a simplified version of an execute stage in a CPU shown in Figure 6. This execute stage consists of 2 functional units, an ADD and a MUL, operating on 2 32-bit operands $\texttt{Op}_1$ and $\texttt{Op}_2$. The final output from the execute stage is the result in Res and a control signal, Valid, to indicate when the result is valid. Similarly, each FU outputs a result, $\texttt{Res}_{add/mul}$, and a valid



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



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



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

Figure 6. High-level diagrams to illustrate H-Houdini. (a) shows the high-level circuit diagram for the example. A slice of the implementation of the MUL FU (described in Figure 7) is shown in (b) (in_use and $\texttt{Valid}_{mul}$) and (c) (count). The red region in (b) contains the update expression in the case in_use is false, and the green region contains the update expression when in_use is true. The dashed purple line indicates the next state update.

signal, $\texttt{Valid}_{add/mul}$, which is then forwarded to the final result/valid through a MUX.

Description of FU. The operational implementation of the ADD/MUL units are shown in verilog-like pseudo code in Figure 7. At a high-level, the ADD FU computes the result and valid bits in a single cycle. The MUL FU implements a classic iterative multiplier that takes 32 cycles to output the result, but, implements a *zero-skip* optimization: if one of the operands is a zero, the multiplier immediately outputs the result (0) in a single cycle. To simplify later discussion, the slice of the circuit used to generate $\texttt{Valid}_{mul}$ is shown visually as a circuit diagram in Figure 6 (b-c).

Property to Prove. For this example we will prove the 2-safety property from VeloCT using H-Houdini. We start by considering two identical copies of the circuit shown

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

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

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

Figure 7. Verilog-like description of the ADD and MUL FU. ADD computes on its two operands and outputs the results and valid in a single cycle. The MUL FU implements a classic 32-bit iterative multiplication algorithm with a zero-skip optimization: non-zero operand values take a slow path (32 cycles) to output the result and valid. Otherwise, if either of the operands are zero, the result and valid bits are computed in a single cycle.

in Figure 6.[6] We will add a superscript of L and R to all the state elements and wires, e.g., $\texttt{Res}^L$ vs. $\texttt{Res}^R$, to differentiate between the copies. The two copies of the circuit are non-interacting and execute independently of each other. Concretely, we will learn an inductive invariant to prove that the two copies of Valid are always equal at every cycle, represented as $\texttt{Eq(Valid)} \iff \texttt{Valid}^L = \texttt{Valid}^R$.

To make the explanation more intuitive, we will present the algorithm as an interactive exercise between the authors and readers. Along the way we will construct the solution graph Figure 8, where each node in the graph represents a property and an edge from node i to node j means that property j requires property i to hold for j to be inductive. The graph is also annotated with callouts (of the form "*i:X*") to indicate the steps. We will refer to these in our explanation below.

Our objective is to prove the property Eq(Valid). For simplicity, we first consider how to ensure this property holds after simulating the circuit for a single cycle. Specifically, if

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

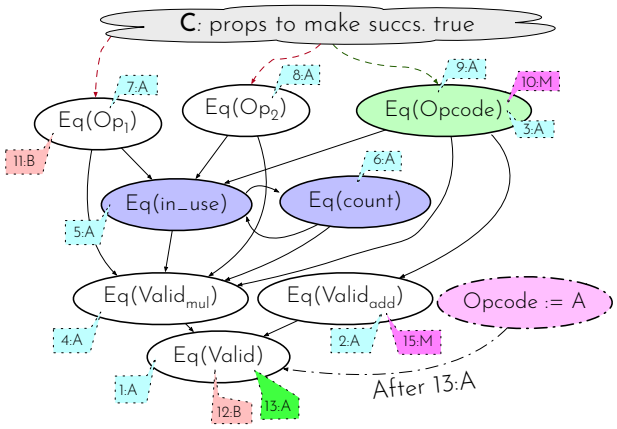


Figure 8. Graphical representation of H-Houdini invariant learning for example circuit in Figure 6. Each node in this graph representation is a predicate, and an edge from predicate i to j denotes that i is required for 1-step inductivity of j. The graph is labeled with callouts of the form *"i-X"*, where i denotes the step number and the X denotes the action: A for abduct, B for backtrack, M for memoization. Hence, the graph is meant to be read in order of the i's.

we start from a state where `Eq(Valid)` holds and take one step, can we guarantee that `Eq(Valid)` will also hold in the subsequent state? From the circuit, we observe that the value of `Valid` in the next cycle is determined by the values of `Opcode`, $\texttt{Valid}_{add}$, and $\texttt{Valid}_{mul}$, which constitute the state elements in the 1-step cone-of-influence (COI) of `Valid`.

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

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

This pattern of abductive reasoning is repeated across queries `2-9A`, allowing us to construct a solution graph (Figure 8). We make two remarks. First, `Eq(in_use)` depends on `Eq(count)`, and vice versa. However, this circular dependency does not cause issues because each property is processed only once. The solution for `Eq(count)` derived using `Eq(in_use)` does not require us to rethink the solution for `Eq(in_use)`, as its solution is already known and remains unchanged. Second, more generally, no property needs to be reasoned about twice (aside from backtracking, discussed next) as no solution changes once it is found. For instance, abductive reasoning in `9:A` does not introduce new queries, allowing us to reuse previously memoized results (shown as `10:M`).

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

Backtracking. What if the property $\texttt{Eq(Op}_1\texttt{)}$ failed to be inductive? This would invalidate the solution we have $\texttt{Eq(Valid}_{mul}\texttt{)}$, so we would need to step back, i.e., backtrack, and find other ways to make $\texttt{Eq(Valid}_{mul}\texttt{)}$ hold (`11-B`). Unfortunately, without a way to restrict values of $\texttt{Eq(Op}_1\texttt{)}$, we cannot make $\texttt{Eq(Valid}_{mul}\texttt{)}$ hold. Thus, we need to backtrack all the way back to `Eq(Valid)` (`12-B`). Can we find a different way to make `Eq(Valid)` hold in the next step? (`13-A`). Yes! Another solution is to have $\texttt{Eq(Valid}_{add}\texttt{)}$ AND `Opcode := ADD` hold. We can continue to repeat the same abductive inference process (`14-A`) reusing memoized solutions from before (`15-M`) to reduce work.

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

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