INSIGHT: Automatic Generation of Explanations for Efficient Identification of Hardware Bugs and Underspecifications

Vincent Quentin Ulitzsch
MIT CSAIL
viniul@mit.edu

Alessandro Bertani
Politecnico di Milano
MIT CSAIL
alessandro.bertani@polimi.it

Peter W. Deutsch
MIT CSAIL
pwd@mit.edu

David Langus Rodriguez
MIT Lincoln Laboratory
Cornell University
david.langus@ll.mit.edu

Kelly Xu
MIT CSAIL
krxu@mit.edu

Aarti Gupta
Princeton University
aartig@cs.princeton.edu

Sharad Malik
Princeton University
sharad@princeton.edu

Mengjia Yan
MIT CSAIL
mengjiay@mit.edu

***Abstract*—Hardware verification is a critical task in processor development. Functional correctness bugs that escape verification can manifest as serious security vulnerabilities, allowing attackers to violate control and dataflow integrity. A common verification strategy to eliminate functional correctness bugs during design time is to compare the architectural behavior of a device under test, i.e., the hardware implementation, against a golden reference implementation, using model checking or fuzzing to expose mismatches. In practice, a single bug often manifests in many mismatch-triggering programs. Moreover, due to ISA underspecification, not every mismatch corresponds to a bug. Without additional guidance, model checkers repeatedly rediscover programs that trigger the same bug or ISA underspecification, while fuzzers generate large numbers of triggering programs that stem from the same underlying cause, causing substantial manual overhead.**

This paper presents INSIGHT, a framework that automatically synthesizes generalized, machine-readable explanations of mismatches between a processor implementation and its reference model. INSIGHT (i) guides model checkers towards exposing distinct bugs and underspecifications and (ii) clusters large program corpora by shared explanations, enabling automatic deduplication.

INSIGHT translates explanation synthesis into a combinatorial optimization problem and solves it efficiently using Integer Linear Programming solvers. We implement and evaluate INSIGHT on an in-order core and the out-of-order BOOM RISC-V core. We show that with INSIGHT, model checkers can discover more distinct bugs in the same timeframe. INSIGHT can also reduce a large corpus of fuzzing testcases into a small number of clusters.

1. Introduction

Hardware verification at the register-transfer level (RTL) is a critical task in processor development, consuming a significant portion of resources. The complexity of today's processors necessitates rigorous verification to ensure functional correctness and security. Reported ratios of up to 5 verification engineers per 1 design engineer highlight the importance of and the challenges involved in verification [1]. Still, bugs escape this validation process and have devastating effects on correctness and security, as evidenced by a series of recently uncovered security vulnerabilities [2], [3], [4]. As CPU designs only grow in complexity, there is an urgent need to increase the efficiency of processor verification methodologies, both in terms of their capability to detect bugs, as well as in reducing the amount of manual overhead involved.

One promising approach to processor verification is to compare the behavior of a device under test (DUT) with that of a golden reference implementation. This comparison can be done by verifying that the two implementations have the same ISA-level behavior. We refer to differences in architectural state between the DUT and the reference implementation as *mismatches*. Mismatches in the ISA-level behavior of the DUT can pose security vulnerabilities, as they could allow an attacker to violate data and control-flow integrity, bypass program analysis techniques, or leak information across privilege boundaries [5], [6], [7], [8].

Techniques to automatically unearth mismatches between the DUT and the reference implementation can roughly be divided into two methodologies: (i) model checking, which formulates the problem of checking equivalence as temporal logic formulas, and then utilizes SAT solving to either prove the DUT's adherence to the reference implementation or identify a program that illustrates a difference to the reference implementation [9], [10], and (ii) fuzz testing and constrained random test generation, which executes the DUT and the reference implementation with (semi-)randomly generated programs and compares their architectural state [5], [6], [7], [11], [12], [13], [14], [15], [16].

When using methods to uncover mismatches between a reference implementation and a DUT, verification engineers

face the following complications.

First, each underlying cause can correspond to multiple mismatch-triggering programs. This stems from the fact that a single bug can be triggered via many different instruction sequences. For example, if a DUT incorrectly forwards values between pipeline stages during a Write-After-Write (WaW) hazard, both Program 1a and Program 1b will independently trigger a mismatch.

Second, there are typically multiple distinct mismatch causes between the DUT and the reference implementation. A design might suffer from multiple distinct bugs, resulting in multiple mismatches of different natures. Further exacerbating this issue is the fact that ISA specifications are often underspecified [17]. A single reference model cannot perfectly capture all valid hardware implementations implementing the same ISA. Each difference in interpretation of the ISA specification between the reference model and the DUT can result in further mismatch causes. For instance, consider the case where one RISC-V processor allows misaligned loads while the other does not. Then both Program 2a and Program 2b, shown in Figure 1, might cause a mismatch between the reference model and the DUT. But since the RISC-V specification does not dictate whether misaligned loads should be allowed [18], these do not point to an actual bug in the DUT.

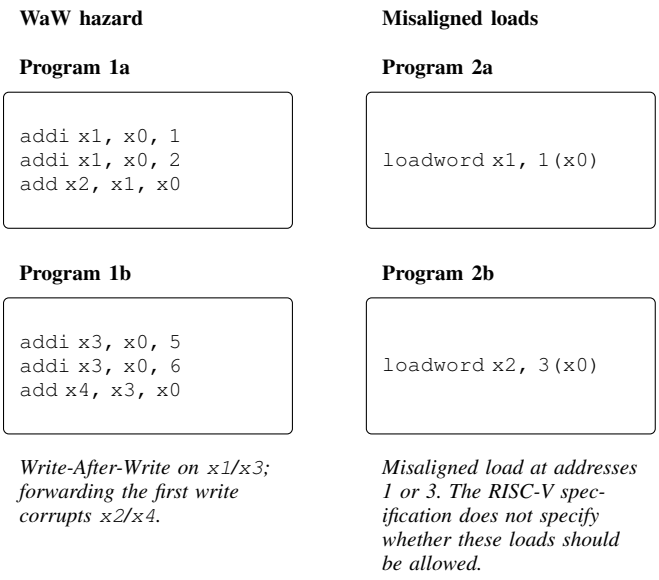


Figure 1. Triggering programs and their explanations for an example RISC-V processor implementation.

Manual overhead through the lack of explanations. The aforementioned complications create manual overhead when using model checking or fuzz testing. A main driver of this manual overhead is that existing tools only provide mismatch-triggering programs, but not a corresponding explanation of why the mismatch occurs.

A standard model checking query yields only a single counterexample (i.e., one mismatch-triggering program) to an assertion. However, verification engineers would like to find all potential mismatches between the reference core and the DUT. Naïvely repeating the same queries is not viable, as this will often rediscover the exact same mismatch-triggering program, offering no new insights to the verification engineer. Even if the original mismatch-triggering program is explicitly excluded from the search space via assumptions, the model checker may return a different program that fails for the exact same reason (e.g., returning Program 2b after Program 2a is excluded).

Given the high computational cost of each query, verification engineers must actively steer the model checker toward mismatches with distinct mismatch causes. Doing so requires either fixing the DUT (in case of a bug) or adjusting the reference model to match the DUT's ISA interpretation (in case of an underspecification issue). However, as noted in a seminal work by ARM [9], the compartmentalization of design and verification teams creates a long feedback loop, making it impractical to wait for hardware fixes. Alternatively, verification engineers can manually formulate constraints to exclude all mismatch-triggering programs that trigger the same mismatch cause from the model checker's search space. To this end, they have to find an explanation that we call *generalized*: a condition that holds for all programs surfacing the same behavior as the original mismatch-triggering program, but not for any other programs.

Fuzzers also suffer from a lack of explanations, although they do not necessarily get stuck. Fuzzers report a plethora of mismatch-triggering programs, potentially sharing the same explanation [19]. The user has to manually deduplicate these reports, which can pose an overwhelming amount of effort. Prior works have proposed methods to work around ISA underspecifications and known bugs, but these either require extensive manual work, rely on simple, error-prone heuristics, or restrict the fuzzer's search space and thereby impede performance [7], [17], [19].

Automatic synthesis of explanations would therefore alleviate a manual overhead that bottlenecks verification workflows. Instead of manually parsing long waveforms to determine the cause of a mismatch, verification engineers could integrate the synthesized explanations into their workflow, reducing debugging effort.

This paper. This paper introduces **INSIGHT**: an algorithm that, given a mismatch-triggering program, finds a machine-readable, generalized explanation of the mismatch, with applications to model checking and fuzz testing. Starting from a single mismatch-triggering program P, INSIGHT generates small mutations of P and partitions them into mismatching and matching sets. INSIGHT then learns an explanation for P by synthesizing a formula that distinguishes mutated programs that still mismatch from those that do not.

A key challenge INSIGHT faces is generalization: akin to learning algorithms, the synthesized explanation must avoid both underfitting and overfitting. An underfitting explanation is too broad, capturing benign programs or programs that do not share the same root cause as the mismatch-triggering program P. In contrast, an overfitting explanation is too narrow, failing to cover programs that manifest the same

bug or ISA underspecification as P.

By avoiding under- and overfitting, INSIGHT produces explanations that extend to unseen mismatch-triggering programs while excluding unrelated behavior. We refer to such an explanation as a *generalized explanation*. A key contribution of INSIGHT is to find generalized explanations by translating the problem of explanation synthesis into a *constrained covering optimization problem*. To the best of our knowledge, INSIGHT is the first approach that provides generalized explanations and formulates explanation synthesis as an optimal covering problem. This reformulation allows INSIGHT to use the power of Integer Linear Programming (ILP) solvers.

INSIGHT's explanations can be used to (i) steer model checkers away from rediscovering the same bug/underspecification and towards distinct failures, and (ii) cluster large fuzzing corpora by shared explanations, yielding automated deduplication.

Contributions. The contributions of this paper are:

- **INSIGHT**, a framework that learns generalized, machine-readable explanations for DUT–reference mismatches, facilitating more efficient hardware verification. INSIGHT formulates explanation synthesis as a constrained covering optimization problem, enabling the use of ILP solvers for efficient search (Section 3 & 4).
- We show how to use INSIGHT's explanations to (i) steer model checkers toward discovering distinct bugs and underspecifications, and (ii) automatically deduplicate large fuzzing corpora by clustering programs that share an explanation (Section 5).
- We evaluate INSIGHT on an in-order core (Kronos) and the out-of-order BOOM RISC-V core. INSIGHT-guided model checking discovers more distinct mismatches than an unguided baseline, and INSIGHT reduces thousands of fuzzer-generated programs into a small number of low-noise clusters (Section 6).

INSIGHT is open-source and available at https://github.com/MATCHA-MIT/insight.

2. Background

This section discusses existing techniques relevant to INSIGHT. We cover model checking, a standard hardware verification method that is one of INSIGHT's target use cases; invariant synthesis, from which INSIGHT draws key algorithmic ideas; and Integer Linear Programming, which INSIGHT leverages in its explanation synthesis to solve a constrained covering optimization problem.

2.1. Model Checking For RTL Verification

Model checking is a standard method for verifying RTL hardware designs, e.g., processors and accelerators [9], [10], [20], [21]. Model checkers analyze annotated RTL code and verify properties specified using SystemVerilog Assertions (SVA). SVA also supports defining assumptions on the system using the `assume` keyword. These assumptions define constraints on the search space (e.g., on the inputs). On a high level, model checkers map the problem of verifying a design's adherence to a user-specified assertion to temporal logic formulas. Through SAT solving, model checkers exhaustively examine all possible input sequences and reachable states to determine whether any of them violate a user-specified assertion. If a violation is found, the model checker generates a *counterexample (CEX) trace* demonstrating the failure. If no violation is found, the model checker guarantees correctness either within a bounded number of steps (bounded model checking) or for all reachable states (unbounded model checking).

2.2. Invariant Synthesis

An invariant is a property that holds at every reachable state of a system. Invariant synthesis is the task of automatically generating such properties for a given hardware or software design. Several families of techniques have been developed for invariant synthesis. The ICE learning framework [22], [23] formulates invariant synthesis as an interaction between a *learner*, which proposes candidate invariants, and a *teacher*, which verifies whether those candidates are correct. The Houdini algorithm [24] can be viewed as a specific instantiation of this framework. Houdini starts from a pool of candidate predicates $\mathcal{P}$ over the program state. Each candidate predicate is a formula over the state variables of the system, such as an equality between two signals or a comparison against a constant value. Houdini iteratively removes predicates from $\mathcal{P}$ that do not satisfy a given property or are not inductive, yielding the maximal inductive subset $\mathcal{P}^\star \subseteq \mathcal{P}$. The conjunction $\bigwedge_{p \in \mathcal{P}^\star} p$ forms the strongest inductive invariant expressible as a conjunction over $\mathcal{P}$.

A complementary line of work is Syntax-Guided Synthesis (SyGuS) [25], where synthesis is guided by a user-provided grammar, and candidates over the synthesis search space are validated using a verifier (such as an SMT solver).

INSIGHT draws on ideas from both lines of work. Like Houdini, INSIGHT operates over conjunctions of predicates, although we do not require inductive subsets in our application. Like SyGuS, INSIGHT uses a structured search space to guide synthesis.

2.3. Integer Linear Programming (ILP)

The goal of ILP is to optimize a linear objective subject to linear constraints with variables restricted to integers. Given the objective and constraints

$$\max\ c^\top x \quad \text{s.t.} \quad Ax \leq b,$$

where $c \in \mathbb{R}^n, A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$, ILP solvers find a valid integer assignment to $x \in \mathbb{Z}^n$ that maximizes the objective. ILPs can be solved using state-of-the-art solvers such as Gurobi [26], CPLEX [27], or lp_solve [28], but the problem is NP-hard in general [29].

3. INSIGHT

This paper presents INSIGHT, a framework that automatically synthesizes generalized, machine-readable explanations of mismatches between a processor design and a reference model.

INSIGHT's Goal. INSIGHT is designed to support two primary use cases. First, it can **guide model checking towards finding mismatch-triggering programs that exhibit distinct behaviors**, ensuring that each discovered program provides new and non-redundant information to the verification engineer. Second, it can **enable the automated deduplication of large sets of mismatch-triggering programs**, such as those produced by fuzzers, by grouping programs that share the same underlying explanation.

To facilitate these use cases, the explanations synthesized by INSIGHT aim to hold true for all programs that require the same bug fix or arise from the same type of ISA underspecification, but not for any other programs. INSIGHT learns these explanations by generalizing from examples.

INSIGHT's Approach. INSIGHT takes as input a single mismatch-triggering program P, and outputs a synthesized explanation. Figure 2 depicts INSIGHT's workflow. INSIGHT first applies (semi-)random, small mutations to the initial mismatch-triggering program P, and then learns an explanation from observations about the behavior of the resulting mutations. In doing so, INSIGHT leverages the following architectural observation. If a small mutation to a mismatch-triggering program still results in another mismatch-triggering program, then both of these programs likely trigger the same underlying bug or ISA underspecification. Conversely, if a mutation no longer produces a mismatch, that change identifies behavior that the explanation should not describe.

INSIGHT's mutations thus result in two sets: a set of other mismatch-triggering programs (called *counterexample (CEX) programs*, referred to as CEX-set), and a set of programs that do not cause a mismatch (called *benign (BEX) programs*, referred to as BEX-set). INSIGHT then learns an explanation for P by learning a formula that separates these two sets. This formula, exemplified in Figure 2, acts as the synthesized explanation.

INSIGHT's challenge is to synthesize an explanation that generalizes from the generated programs, as depicted in Figure 3. Akin to learning algorithms, INSIGHT's synthesis algorithm needs to avoid explanations that are *underfitting* (describing programs that do not trigger the same bug or ISA underspecification as P, or even unrelated benign examples) or *overfitting* (there are some programs that trigger the same bug or underspecification as P which are not described by the explanation). By synthesizing an explanation that avoids under- and overfitting, INSIGHT finds explanations that extrapolate even to mismatch-triggering programs that INSIGHT has not seen, without holding true for unrelated programs. We call such an explanation a *generalized explanation*. Generalized explanations are important for both INSIGHT's use in model checking as well as deduplication. In the case of model checking, overfitting explanations could lead to an increase in (potentially redundant) model checking queries, while underfitting explanations might result in missed mismatches. In the case of deduplication, overfitted explanations will increase the number of clusters, thereby increasing manual effort, while underfitted explanations will cluster unrelated mismatch-triggering programs together.

Core challenges solved by INSIGHT. INSIGHT's workflow solves three core challenges that arise when attempting to synthesize generalized explanations.

Challenge C1: Choosing the right representation. The first challenge lies in identifying the right representation for INSIGHT's explanations. The representation needs to be expressive enough to capture a wide range of mismatch behaviors, yet constrained enough to permit efficient search. INSIGHT chooses quantifier-free first-order logic formulas over internal signals as a representation (Section 3.1).

Challenge C2: Defining a metric for the quality of explanations. The second challenge lies in developing a method to automatically determine whether a candidate explanation is generalized (not over- or underfitting). This is not only a matter of generating a diverse set of program mutations to the original program P: as we explain in Section 3.2, not every explanation that perfectly separates the generated CEX from the generated BEX programs is also generalized, forcing us to adjust our criterion. INSIGHT solves this challenge by introducing a *formula score* that balances multiple objectives (Section 3.2).

Challenge C3: An efficient search algorithm. INSIGHT's third challenge is to devise an efficient search algorithm capable of identifying explanations that optimize the desired evaluation metric. INSIGHT translates the problem of finding the formula with the best formula score into a constrained covering optimization problem, which allows it to harness ILP solvers (Section 3.3).

3.1. Generalized Explanation Representations

We now determine the appropriate representation for INSIGHT's generalized explanations. Three requirements arise from INSIGHT's use-cases. First, the representation must allow explanations to be machine-readable and directly translatable into search space restrictions for model checkers, and allow for automatic evaluation (*machine-readability*). Second, the representation must be expressive enough to capture a wide range of microarchitecture conditions (*expressiveness*). Third, the representation must be restricted enough to make searching through the representation space computationally feasible (*restrictiveness*).

Quantifier-free first-order logic formulas (with theories supported typically by SMT Solvers, such as equality and arithmetic) present themselves as a natural choice for representations of explanations, as they fulfill the *machine-readability* requirement.

Naïvely, one could pick logic formulas describing only instruction sequences themselves as a representation. For

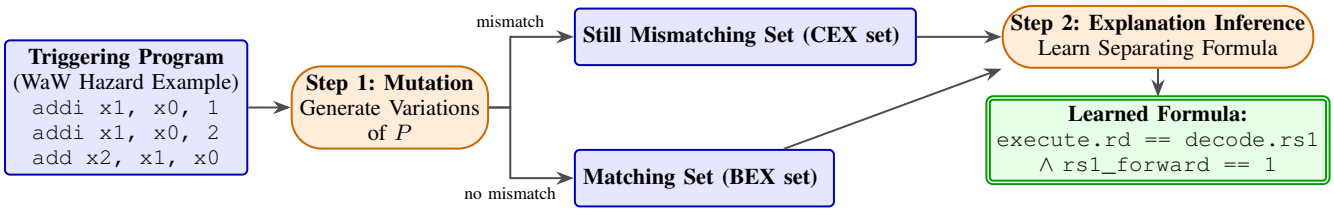


Figure 2. High-level overview of INSIGHT's explanation synthesis. Mutations of a triggering program produce both mismatching (CEX) and benign (BEX) cases. We use ILP to find a formula that best separates CEX from BEX programs.

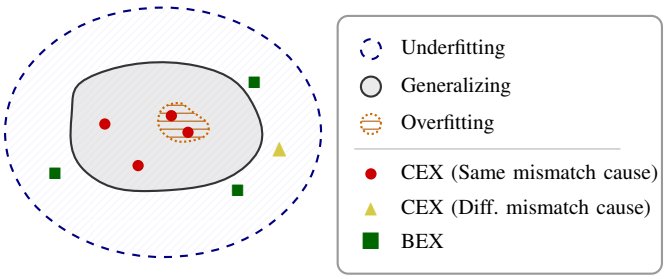


Figure 3. The relationship between underfitting, generalizing, and overfitting. Each boundary represents a potential explanation.

instance, one could attempt to explain mismatches caused by misaligned loads as all load instructions with an unaligned offset. However, this representation does not lead to concise explanations: the formula must also capture the program's data and control flow to determine the effects of the instructions. For example, the target address of a load instruction can be derived from a register. Therefore, deciding whether a load is misaligned requires tracking register values, and handling mismatch-relevant control flow requires modeling which paths execute. Doing so would require the logic formulas to encode an ISA machine, increasing their complexity to a point where this choice cannot describe programs of arbitrary length. As such, this representation lacks both expressiveness and restrictiveness.

INSIGHT's explanation format. To overcome this challenge, INSIGHT's representation for explanations is logic formulas with the following properties:

- they are formulated over core internal signals rather than instruction sequences,
- they describe constraints on a single-cycle state, and
- they are composed of conjunctions of basic predicates.

We call these formulas *state formulas*. This representation fulfills all three requirements for INSIGHT's representation.

By formulating logic formulas over the design's internal signals, rather than only instruction sequences, INSIGHT implicitly captures control and data–flow decisions without re-encoding program semantics or branching behavior. For instance, the Sodor core (a single cycle RISC-V core) [30] has an internal `misaligned` Boolean signal, describing whether a load is misaligned. The formula `sodor.misaligned == 1` is sufficient to characterize program executions in which a misaligned load occurs. INSIGHT therefore leverages the design's internal signals, such as control registers or pipeline status bits, to obtain an expressive representation that is simpler than logic formulas over instruction sequences.

To also achieve restrictiveness, we constrain the formulas to statements over a single-cycle state. One might naturally assume that such a formulation is too restrictive to describe complex bugs triggered over multiple cycles. However, a crucial architectural observation of INSIGHT is that this representation still retains expressiveness. This is because hardware history is inherently encoded within the core's internal signals. For instance, a single-cycle snapshot of a Load-Store-Queue inherently reflects a long history of prior memory accesses. Consequently, mismatches can be described as an incorrect next-state transition at a particular micro-architectural state, regardless of the specific instruction sequence used to reach that state. INSIGHT captures this pivotal state, thereby leveraging the implicit encoding of history in the core's internal signals. We discuss the limitations of this single-cycle state formula representation in Section 7.

Lastly, INSIGHT restricts state formulas to a conjunction of basic predicates. This assumption aligns with traditional invariant synthesis work and enables more efficient search [24], [31]. Restricting INSIGHT's search space to conjunctions of basic predicates further facilitates the discovery of commonalities between the CEX programs. We elaborate on possible use of disjunctions in Section 5 and Section 7.

3.2. Metric to Evaluate Generalized Explanations

The next challenge lies in identifying a criterion to automatically determine whether a candidate state formula is a good generalized explanation.

To develop this criterion, we first define the concept of *covering*, which helps quantify the relationship between a candidate explanation and the programs generated through mutations. Executing a program on a given design produces a sequence of states (one for each cycle), referred to as its execution trace. A formula ϕ is said to cover a state s if $s \models \phi$ (i.e., ϕ holds true for s). A formula covers a trace τ if it covers at least one state within that trace (if $\vee_{s \in \tau} s \models \phi$). Intuitively, if a formula ϕ covers a trace, it explains that trace and, by extension, the corresponding program. Throughout this paper, we will also refer to a formula ϕ covering a program if it covers the corresponding execution trace.

Recall that INSIGHT tries to learn an explanation from the generated CEX and BEX programs. In defining a criterion for whether this is generalized, INSIGHT needs to consider three factors:

(1) Generated counterexamples are not guaranteed to share an explanation with P. It may be the case that a mutated program still triggers a mismatch, but should not be grouped together with P, as the mismatch is caused by a different underlying bug or ISA underspecification. For instance, consider a core that does not support misaligned loads and further implements incorrect store-to-load forwarding. Now assume INSIGHT receives as input a program P that triggers a misaligned load. If INSIGHT's mutations cause the mutated program P' to perform an aligned load, but to an address that has been previously written to, P' is still mismatch-triggering, but should not be grouped together with P (see Figure 3, ▲).

(2) Mutations might result in masked counterexamples. Mutations of the original mismatch-triggering program P might yield cases that exercise the same microarchitectural conditions as P but do not ultimately manifest a mismatch. This masking effect can be the result of, e.g., coincidental value alignment or arithmetic masking. For instance, consider Program 3, depicted in Figure 4, exhibiting a Write-After-Write (WaW) hazard. Under correct architectural behavior, the processor must forward the result of $I2$ to $I3$, as $I2$ computes the latest value for register `x1`. Now, consider a buggy processor that mishandles WaW hazards by incorrectly forwarding the stale result from $I1$ to $I3$. Despite this bug, because both $I1$ and $I2$ happen to compute the same value (7), the final state of `x4` remains architecturally correct. We refer to programs that exhibit this underlying behavior without causing a detectable mismatch as masked counterexamples (masked CEXs).

Because masked CEXs exercise the bug but appear benign, they are placed in the BEX set. If INSIGHT were strictly required to exclude these seemingly benign traces from the explanation, it would be forced to find complex predicates that distinguish P from the masked CEX based on coincidental data values rather than the actual explanation itself. Requiring the synthesized formula ϕ to exclude masked CEXs could thus lead to complex and overfitted explanations that fail to generalize.

Program 3 (WaW Hazard)

```
# Setup: x2 := 3; x3 := 4
I1: x1 := x2 + 4  # x1 is 7
I2: x1 := x3 + 3  # x1 is 7
I3: x4 := x1 + 2
```

Figure 4. Masked counterexample for a WaW hazard

(3) Long formulas can overfit. Finally, note that formulas with a large number of predicates can easily overfit. For instance, a trivial solution that covers all CEX traces, but no BEX traces, would be to simply conjunct a predicate $instruction \neq bex$ for each BEX-trace $bex \in BEX$. However, this solution would not offer any explanation for the underlying mismatch cause.

INSIGHT's *Formula Score*. INSIGHT's key idea to address the above considerations is to formulate the problem of finding a generalized explanation as a *constrained covering optimization problem* instead of enforcing a strict separation requirement. To this end, INSIGHT finds a formula ϕ that maximizes a *formula score* while fulfilling a set of constraints. The formula score balances multiple objectives, namely 1) maximizing the number of covered CEX traces, 2) minimizing the number of covered BEX traces, and 3) minimizing the number of predicates used in ϕ.

The conflict between these three objectives enforces generalization, by promoting a middle ground between maximizing covered CEX traces, minimizing covered BEX traces, and minimizing the formula's complexity. We formally describe the formula score in Section 4.1.

3.3. Efficiently Synthesizing Explanations

The final challenge is to find a single-cycle state formula that yields the maximum formula score.

To this end, INSIGHT employs a two-phase strategy inspired by the Houdini algorithm [24] (which is discussed in Section 2.2). INSIGHT first generates (or *mines*) a set of candidate predicates, and then second, selects the optimal subset of these predicates.

Mining candidate predicates. INSIGHT automatically mines a set of candidate predicates $\mathcal{P}$ through analysis of the traces belonging to the CEX and BEX programs. For instance, when a signal `signal` holds value v in a counterexample trace of the initial mismatch-triggering program, but differs in benign runs, INSIGHT adds the candidate predicate `signal` $==$ v to $\mathcal{P}$. This ensures that $\mathcal{P}$ contains only those predicates that could potentially distinguish mismatch triggering from benign executions. We give more details on this selection process in Section 4.2.

Efficient predicate selection. After generating the set of candidate predicates $\mathcal{P}$, INSIGHT is faced with the challenge of finding a subset of $\mathcal{P}$ whose conjunction yields the highest formula score. A simple enumerate-and-score approach is computationally intractable due to the exponential number of possible predicate subsets. Furthermore, directly applying the Houdini algorithm is not suitable as it aims to find the maximal inductive subset of predicates satisfying a property, whereas INSIGHT faces a multi-objective optimization problem.

Instead, INSIGHT exploits a key structural property of its representation: for a set of states S, the subset of states covered by a conjunction of predicates is precisely the intersection of the states covered by the individual predicates. Formally, if $cov(\phi)$ denotes the set of states covered by formula ϕ, then $cov(\phi_1 \wedge \phi_2) = cov(\phi_1) \cap cov(\phi_2)$.

Recall that the formula score of a conjunction $\phi = p_1 \wedge p_2 \cdots \wedge p_k$ depends on which CEX and BEX traces (and therefore, states) it covers. Therefore, finding a formula ϕ with the highest formula score corresponds to finding a

selection of predicates whose intersection of their covered states maximizes the number of covered CEX traces, while minimizing the number of covered BEX traces and the formula's length. This observation enables us to develop an efficient search algorithm. INSIGHT only needs to determine once, for each predicate, which states it covers. Subsequently, finding the covered states for any conjunction becomes a computationally inexpensive matter of computing set intersections. This reformulation converts the original problem into a combinatorial optimization problem, allowing INSIGHT to harness well-established ILP solvers.

To this end, INSIGHT formulates an Integer Linear Program, set up to find a selection of predicates with maximum formula score. Thereby, INSIGHT obtains an efficient-in-practice predicate selection algorithm.

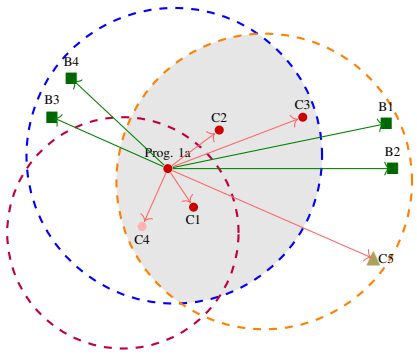
Illustration of INSIGHT. We illustrate INSIGHT's approach in Figure 5 using the WaW hazard bug from Program 1a in Figure 1. Recall that this bug causes the processor to forward an incorrect result to an affected instruction. Assume that INSIGHT's mutations yield some BEX- and the following CEX programs: C1, which reproduces the same WaW hazard as Program 1a but with different register operands; C2 and C3, which trigger the same WaW forwarding bug but using a `sub` instruction instead of `add` as the consuming instruction; and C5, which triggers a mismatch due to an unrelated bug. Assume that the mutations also produce a masked counterexample C4, which exercises the WaW hazard but does not cause a visible mismatch because both writes happen to produce the same value.

Now suppose INSIGHT mines three candidate predicates from the execution traces: p_1: `rs1_forward == 1` (the forwarding path for `rs1` is active), p_2: `execute.rd == dec.rs1` (the destination register in the execute stage matches the source register being decoded), and p_3: `dec.op == add` (the decoded instruction is an `add`). Each predicate covers a subset of the observed traces, which we can visualize as a circle in Figure 5. INSIGHT's goal is to select predicates such that the intersection of their corresponding circles maximizes the number of covered CEX traces, minimizes the number of covered BEX traces, and minimizes the number of selected predicates.

In this example, selecting predicates p_1 and p_2 yields the maximum formula score. Their intersection covers C1, . . . , C4 while leaving C5 uncovered, correctly treating it as an outlier from a different mismatch cause. Importantly, the solution does not select p_3: adding `dec.op == add` would exclude C2 and C3 (which use `sub`), overfitting the explanation to the specific opcode of Program 1a rather than capturing the underlying hazard condition. Note also that $p_1 \wedge p_2$ covers the masked counterexample C4 in the BEX set, since C4 exercises the same forwarding condition. Intuitively, the resulting formula describes that forwarding is active when it should not be.

4. INSIGHT Search Algorithm in Detail

We now describe INSIGHT's search algorithm in more detail, using the notation described in Table 1. Recall that



Points: ● CEX (Same cause) ▲ CEX (Diff. cause) ■ BEX ● Masked counterexample

Predicates: p_1: rs1_fwd == 1 p_2: exe.rd == dec.rs1 p_3: dec.op == add $p_1 \cap p_2$ (selected)

Figure 5. INSIGHT as a constrained covering problem. We want the intersection of all predicates to maximize the number of covered CEX traces, while minimizing the covered BEX traces and the number of selected predicates.

TABLE 1. NOTATION.

Symbol	Meaning
P	Input triggering program.
P_{CEX}	Set of mismatching (CEX) programs generated from P.
P_{BEX}	Set of matching (BEX) programs generated from P.
$\mathcal{T}_{\text{CEX}}$	Execution traces corresponding to P_{CEX}.
$\mathcal{T}_{\text{BEX}}$	Execution traces corresponding to P_{BEX}.
S	Set of all states appearing in at least one trace.
S_{BEX}	States observed in at least one BEX trace.
S_{CEX}	States observed in at least one CEX trace.
$\tau_{origcex}$	Trace corresponding to the original triggering program P.
$\mathcal{P}$	Set of candidate predicates mined from $\mathcal{T}_{\text{CEX}}$ and $\mathcal{T}_{\text{BEX}}$.
$\phi \subseteq \mathcal{P}$	Subset of predicates forming a potential explanation.
ϕ (as formula)	Conjunction of all predicates in ϕ.
$cov(p)$	Subset of states in s covered by a predicate/formula p.
$covCEX\phi$	Subset of CEX traces covered by a formula ϕ.
$covBEX\phi$	Subset of BEX traces covered by a formula ϕ.

INSIGHT's search algorithm takes as input the two sets of mutated programs P_{CEX} (denoting the CEX-set) and P_{BEX} (denoting the BEX-set), the associated execution traces $\mathcal{T}_{\text{CEX}}$ and $\mathcal{T}_{\text{BEX}}$, and the original program's trace $\tau_{origcex}$. The search algorithm itself is independent from the exact way the two sets of mutated programs are produced – we provide details on our mutation implementation in Section 6.

Given the two sets P_{CEX}, P_{BEX} and their execution traces as input, INSIGHT finds a formula ϕ with maximum formula score (as will be defined in Section 4.1) as follows. INSIGHT first mines a set of candidate predicates $\mathcal{P}$ via dynamic analysis of the $\tau_{origcex}$ and BEX traces $\mathcal{T}_{\text{BEX}}$ (Section 4.2). It then constructs and solves an ILP to identify an optimal selection of predicates (Section 4.3).

4.1. Formula Score

Recall that INSIGHT represents candidate explanations as a conjunction of simple, "basic" predicates (for example, `signal == value`, `signal1 == signal2`, or the Boolean outcome of a conditional). Concretely, a candidate formula ϕ is a set of predicates $\{p_1, p_2, \ldots, p_k\}$ and the corresponding logical formula is the conjunction $\phi := p_1 \wedge p_2 \wedge \cdots \wedge p_k$.

As explained in Section 3.2, INSIGHT solves a constrained optimization problem to guide the selection of predicates towards a generalized explanation. Recall from Section 3.2 that the formula score must account for three factors: (1) generated CEXs are not guaranteed to share an explanation with P, (2) masked CEXs may appear in the BEX set, and (3) long formulas can overfit. To this end, INSIGHT identifies a subset of predicates whose conjunction constitutes a formula ϕ that optimizes a combination of three objectives, each addressing one of the factors:

1) **Maximize the number of covered CEX traces** $|\boldsymbol{covCEX(\phi)}|$. Maximizing enforces generalization, as INSIGHT must find an explanation shared by many CEX traces, preventing overfitted formulas. By maximizing rather than requiring that all CEX traces are covered, we account for the fact that not all generated CEXs necessarily share an explanation with P (addressing factor 1).
2) **Minimize the number of covered BEX traces** $|\boldsymbol{covBEX(\phi)}|$. This prevents INSIGHT from finding underfitting formulas. By minimizing rather than requiring that no BEX traces are covered, we account for the possibility that some BEX programs are masked counterexamples (addressing factor 2).
3) **Minimize the length of the formula** $\boldsymbol{\phi}$, measured through the number of predicates $|\phi|$. This prevents overfitted solutions that conjunct many predicates to memorize the generated examples (addressing factor 3).

INSIGHT optimizes this combination subject to the constraint that the CEX trace $\tau_{origcex}$ belonging to the input program P must be covered. Note that there is a trade-off between INSIGHT's three key objectives. For instance, it might be that one formula ϕ_1 covers more CEX than another formula ϕ_2, but only at the cost of also covering more BEXs than ϕ_2. Which formula to pick in this trade-off space is a matter of prioritizing either over- or underfitting. INSIGHT allows the user to encode their preferences through adjustable weights, controlling the priorities in finding these formulas. This leads to the following *formula score*, introducing adjustable weights λ and μ to control the priorities in finding these formulas.

$$\sigma(\phi) = |covCEX(\phi)| - \lambda \cdot |covBEX(\phi)| - \mu \cdot |\phi| \quad (1)$$

. Here, λ, the *BEX penalty*, controls the cost of covering benign traces, and μ, the *predicate cost*, controls the penalty for formula complexity. INSIGHT finds a formula ϕ that maximizes this formula score.

4.2. Predicate Mining

INSIGHT mines candidate predicates from signal mismatches between the input trace $\tau_{origcex}$ of program P and benign traces $\mathcal{T}_{\text{BEX}}$, using the grammar in Figure 6. In summary, INSIGHT generates different predicate candidates depending on a signal's type and observed values during execution.

INSIGHT's predicate mining strategy is guided by the observation that mismatches can be explained by a combination of control-flow decisions upon encountering certain values or hazards/resource contentions. Each of these factors can be represented by predicates that relate an internal signal to a constant or to other internal signals. To avoid overfitting and increase scalability, INSIGHT considers a signal's type when mining predicates. For instance, INSIGHT does not mine predicates for counter signals, since those are rarely useful in explaining a mismatch.

In detail, INSIGHT mines predicates as follows. First, INSIGHT extracts, through static analysis of the RTL code, each RTL conditional (branching/mux) present in the code. Second, INSIGHT assigns each internal signal in the RTL code one or multiple types, which INSIGHT derives from a user-provided function. For BOOM and Kronos, a simple classification function based on the signal's bitwidth and ≈ 10 lines of regular expressions were sufficient for this classification (for instance, signals of bitwidth 1 are assigned the type control signal). In our implementation, we classify the signals into the following types.

Control Signal Signals which influence control-flow decisions.

Immediate Signal Signals which carry immediate values, for instance, arguments to an ALU.

Register-Index Signal Signals which carry physical or logical register indices.

INSIGHT then takes this classification, and the list of RTL conditionals, and generates predicate candidates using the concrete values observed during CEX/BEX execution. Concretely, INSIGHT generates the following signal types.

EqPred For any control signal, add `signal` $== v$ for values v seen in τ_{origcex}.

NeqPred For any control or register-index signal taking value v in a BEX state, add `signal` $\neq v$.

CrossEq Add $\texttt{signal}_1 == \texttt{signal}_2$ if two register-index/immediate signals coincide in some τ_{origcex} state but differ in some BEX state.

CondPred For each RTL conditional, add the Boolean-outcome predicate (e.g., $(a \vee b) == v$, $v \in \{0, 1\}$).

We highlight that INSIGHT can also handle alternative mining strategies, e.g., based on static analyses or expert-provided predicates. In addition, the user-provided signal-type function can be heuristic and does not need to be completely accurate. A wrongly typed signal can result in over-/underfitting or increased solving time, but does not fundamentally block the algorithm.

$$\langle \text{Formula} \rangle ::= \langle \text{Pred} \rangle \ \big(\wedge \ \langle \text{Pred} \rangle \big)^*$$

$$\langle \text{Pred} \rangle ::= \underbrace{\texttt{signal} = \langle \text{Val} \rangle}_{\langle \text{EqPred} \rangle} \mid \underbrace{\texttt{signal} \neq \langle \text{Val} \rangle}_{\langle \text{NeqPred} \rangle} \mid$$

$$\underbrace{\texttt{signal}_1 = \texttt{signal}_2}_{\langle \text{CrossEq} \rangle} \mid \underbrace{(\langle \text{Cond} \rangle) = \langle \text{Bool} \rangle}_{\langle \text{CondPred} \rangle}$$

Figure 6. Predicate grammar used by INSIGHT.

4.3. Search Algorithm for Predicate Subset Selection

Given the set of candidate predicates $\mathcal{P}$, the goal of INSIGHT is to identify a subset of predicates $\phi \subseteq \mathcal{P}$ that maximizes the overall formula score $\sigma(\phi)$. As discussed in Section 3.3, the search space of all possible subsets of $\mathcal{P}$ is exponentially large. Exhaustively enumerating and scoring every possible combination of predicates is thus computationally infeasible.

To overcome this challenge, INSIGHT reformulates the search for an optimal subset as a *combinatorial optimization problem*. This reformulation enables INSIGHT to utilize well-established ILP solvers and employ multiple optimization strategies to further prune the search space.

Recall that the set of states covered by a conjunction of predicates is precisely the intersection of the states covered by each predicate. Formally,

$$\text{cov}(p_1 \wedge p_2) = \text{cov}(p_1) \cap \text{cov}(p_2). \tag{2}$$

INSIGHT uses this observation to cast the problem of finding an optimal selection of predicates as a *constrained covering optimization problem*.

INSIGHT only needs to identify the set of covered states of each predicate p, denoted by $\text{cov}(p)$, once. It can then compute the set of covered CEX and BEX traces of a conjunction of predicates simply through their intersection, without needing to do further logical formula evaluation. INSIGHT exploits this observation to construct an ILP instance that identifies the optimal selection of predicates from $\mathcal{P}$.

To formulate the ILP, INSIGHT introduces a binary variable for each predicate $p \in \mathcal{P}$, each collected trace $\tau \in (\mathcal{T}_{\text{CEX}} \cup \mathcal{T}_{\text{BEX}})$, and each state $s \in S$ in those traces (where S denotes the set of all states appearing in at least one trace) as follows:

- $x_p \in \{0, 1\}$, indicating whether predicate p is selected.
- $x_\tau \in \{0, 1\}$, indicating whether trace τ is covered.
- $x_s \in \{0, 1\}$, indicating whether state s is covered.

We use the constraints in Figure 7 to ensure that x_s is set to 1 if and only if all selected predicates cover s, and x_τ is set to 1 if and only if at least one of its contained states is covered. The constraints formulate the relationship between each predicate and its covered states, and the relationship between each trace and its comprising states. We can then formulate the ILP objective to maximize the selection's formula score through Equation 3.

C1: If a state is covered, then no selected predicates excludes it.

$$\forall s \in S\colon \ \ x_s = 1 \ \Rightarrow \sum_{\{p \mid s \notin cov(p)\}} x_p \leq 0 \tag{C1}$$

C2: If a state is not covered, at least one selected predicate excludes it

$$\forall s \in S\colon \ \ x_s = 0 \ \Rightarrow \sum_{\{p \mid s \notin cov(p)\}} x_p \geq 1 \tag{C2}$$

C3: A CEX trace is covered iff at least one of its states is covered.

$$\forall \tau_{cex} \in \mathcal{T}_{\text{CEX}}\colon \ \ x_{cex} = 1 \ \Leftrightarrow \sum_{s \in \tau_{cex}} x_s \geq 1 \tag{C3}$$

C4: A BEX trace is covered iff at least one of its states is covered.

$$\forall \tau_{bex} \in \mathcal{T}_{\text{BEX}}\colon \ \ x_{bex} = 1 \ \Leftrightarrow \sum_{s \in \tau_{bex}} x_s \geq 1 \tag{C4}$$

C5: Rule out the solution that selects no predicates.

$$\sum_{p \in \mathcal{P}} x_p \geq 1 \tag{C5}$$

C6: Force the original input trace to be covered by the synthesized formula.

$$x_{\tau_{\text{origcex}}} = 1 \tag{C6}$$

Figure 7. Constraints to enforce selection of predicates that maximize the resulting formula score, using Equation 2.

$$\max \quad \left[\sum_{\tau \in \mathcal{T}_{\text{CEX}}} x_\tau\right] - \lambda \left[\sum_{\tau \in \mathcal{T}_{\text{BEX}}} x_\tau\right] - \mu \left[\sum_{p \in \mathcal{P}} x_p\right], \tag{3}$$

where λ and μ are the tunable weight factors of the formula score, as described in Section 4.1. Through constraint C6, we ensure that the final predicate selection covers $\tau_{origcex}$. Note that for a given selection of predicates $\phi = \{p \mid x_p = 1\}$, the ILP objective given in Equation 3 exactly matches ϕ's formula score given by Equation 1.

4.4. Optimizations

INSIGHT implements multiple optimizations for better generalization and scalability.

Further reducing overfitting through architectural constraints. Formulating synthesis as a constrained covering optimization problem allows easy incorporation of architectural knowledge into INSIGHT. For example, a predicate that fixes the value of the `csr_addr` field of the currently decoded instruction rarely aids in deriving a generalized explanation, unless it is embedded in a CSR-instruction related formula. We can encode this preference directly in the ILP by conditioning the selection of these predicates on the selection of predicates that describe the currently decoded instruction as CSR-related, thereby further reducing the risk of overfitting.

Synthesizing for both cores. INSIGHT always synthesizes two state formulas: one using only signals of the reference implementation, and one using signals of the DUT. INSIGHT then returns the state formula with a higher formula score as the explanation. This helps in synthesizing generalized explanations, as explaining mismatches through a simple, single-cycle reference implementation can yield simpler explanations (for instance, in the case of misaligned instructions, see Section 6).

Scalability optimizations. INSIGHT applies preprocessing to shrink the constructed ILP instance. It discards *dominated predicates* (those covering a subset of CEX states but a superset of BEX states of another predicate) and prunes states that do not affect predicate selection. See Appendix 10.1 for details.

In addition, INSIGHT ranks candidates by individual formula score and constructs the ILP out of only the n highest scoring candidates. This optimization is not guaranteed to preserve the optimality, but it does not seem to harm generalizability in practice.

5. Usecases for INSIGHT

We now describe how to use INSIGHT in its two intended use cases: guiding model checkers and deduplicating a set of mismatch-triggering programs.

Guiding Model Checkers With INSIGHT. INSIGHT's explanations can steer model checkers toward distinct bugs and underspecifications. Algorithm 1 describes an iterative refinement workflow for this purpose. The algorithm maintains a set of learned explanations Φ (line 1). Each iteration begins by querying the model checker under assumptions $\bigwedge_{\phi\in\Phi}\neg\phi$ (line 3). These assumptions exclude traces already covered by the learned explanations Φ. If no counterexample is returned (line 4), the loop terminates (line 5), as no further mismatches can be found within the exploration bound. Otherwise, we extract the mismatch-triggering program P (line 7). We then synthesize an explanation ϕ using INSIGHT and add it to Φ (lines 9–11). We repeat this synthesis (lines 9–13) until no unexplained generated counterexamples remain. This optimization reduces the number of required model checking queries. It achieves this by synthesizing explanations for counterexamples that were found through INSIGHT's mutations, but left uncovered by the explanation synthesized in the corresponding INSIGHT run.

In subsequent queries, each $\neg\phi$ is inserted as an `assume` constraint. This excludes executions satisfying previously discovered explanations. As a result, the model checker is directed toward distinct mismatches. Notice that this algorithm synthesizes disjunctions implicitly. If two explanations ϕ_1 and ϕ_2 are derived, we add `assume` $(\neg(\phi_1 \wedge \phi_2))$. This is equivalent to `assume` $(\neg\phi_1 \vee \neg\phi_2)$. We evaluate this workflow in Section 6.3.

Testcase Deduplication. INSIGHT enables automatic clustering of a corpus of mismatch-triggering programs F_{CEX}. Algorithm 2 demonstrates this workflow. While the corpus to deduplicate F_{CEX} is not empty, we pick a random program P from the remaining set (line 3). We then mutate P and synthesize an explanation ϕ for it (lines 4–5). All programs in the corpus covered by ϕ are grouped into the same cluster (line 6), and the covered programs are then removed from the remaining corpus (line 7). The process iterates until the entire corpus has been assigned to a cluster (lines 8–10). We note that, should a bug require multiple disjuncted state formulas to be fully explained, Algorithm 2 will generate each of the required state formulas, but present them as different clusters. We evaluate this workflow in Section 6.4.

Algorithm 1 Guided model checking with INSIGHT

Input: DUT, Reference
1: $\Phi \leftarrow \emptyset$ ▷ set of learned explanations
2: **while** True **do**
3: $CEX \leftarrow$ MODELCHECK(DUT, Reference, `assume` $\bigwedge_{\phi\in\Phi}\neg\phi$)
4: **if** $CEX = \perp$ **then**
5: **break** ▷ no further mismatches
6: **end if**
7: $P \leftarrow$ EXTRACTPROGRAM(CEX)
8: **repeat**
9: $P_{CEX}, P_{BEX} \leftarrow$ INSIGHT$Mutate(P)$
10: $\phi \leftarrow$ INSIGHT$Synthesize(P_{CEX}, P_{BEX})$
11: $\Phi \leftarrow \Phi \cup \{\phi\}$
12: $P_{CEX} \leftarrow \{p_{CEX} \in P_{CEX} \mid p_{CEX} \not\models \Phi\}$ ▷ remove already explained samples
13: **if** $P_{CEX} \neq \emptyset$ **then**
14: $P \leftarrow$ any $p \in P_{CEX}$ ▷ Pick a new unexplained sample for next mutation
15: **end if**
16: **until** $P_{CEX} = \emptyset$
17: **end while**
18: **return** Φ ▷ distinct mismatch/underspec explanations

Algorithm 2 Testcase deduplication using INSIGHT

1: **Input:** F_{CEX} ← counterexamples (e.g., found by a fuzzer)
2: $\Phi \leftarrow \emptyset$
3: **while** $F_{CEX} \neq \emptyset$ **do**
4: $P \leftarrow$ random CEX chosen from F_{CEX}
5: $P_{CEX}, P_{BEX} \leftarrow$ INSIGHT$Mutate(P)$
6: $\phi \leftarrow$ INSIGHT$Synthesize(P_{CEX}, P_{BEX})$
7: $F_{CEX} \leftarrow \{p_{CEX} \in F_{CEX} \mid p_{CEX} \not\models \phi\}$ ▷ remove already explained samples
8: $\Phi \leftarrow \Phi \cup \{\phi\}$
9: **end while**
10: **return** Φ ▷ Φ defines a clustering of F_{CEX}

Debugging. INSIGHT's explanations point to a pivotal state that will lead to a mismatch between a DUT and a reference core. These explanations do not necessarily point to an immediate bug fix, but can aid the debugging process in practice. Section 6.2 demonstrates that INSIGHT's explana-

tions are short (≈ 3 predicates on average for BOOM) and provide useful information in practice.

Integration into agentic code generation. Beyond manual debugging, we envision that INSIGHT can also facilitate a self-correcting agentic AI workflow. In an agentic code generation workflow, an LLM generates RTL code, which is then verified through established techniques such as model checking, and then correct the RTL code based on the results [32]. Typically, this correction step is done by feeding long counterexample traces back to the LLM. We imagine that feeding back INSIGHT's explanations, whether as formulas over internal signals or natural-language hints, could help steer the LLM towards better code fixes, improving this agentic code generation workflow. We leave the evaluation of this agentic workflow for future work.

6. Evaluation

Our evaluation aims to answer three questions:

RQ1: How well can INSIGHT synthesize generalized explanations across different bug types and cores?
RQ2: Is INSIGHT effective in unblocking model checking?
RQ3: Is INSIGHT effective in deduplicating fuzzing results?

6.1. Implementation and Dataset

Implementation. We implement INSIGHT's mutation engine, predicate mining, and ILP construction and solving as follows.

Our Rust-based, parallelized mutation engine implements a set of structure-aware mutations geared towards providing useful information to INSIGHT, such as changing a random operand, opcode, or immediate, inserting random instructions, or replacing all occurrences of a register or an immediate value. Mutated programs are classified via Verilator co-simulation using per-core testbenches that emit an architectural commit log (PC, retired instruction, destination register, and register values). A program triggers a mismatch if and only if the commit logs diverge between the reference core and the DUT. We also extend Verilator [33] to instrument the RTL to emit a Boolean signal for every conditional (`if`/`case`) statement and log its outcome per cycle, enabling extraction of `CondPred` candidates. We parse the resulting waveforms and mine the predicates using a parallelized Rust implementation, which then constructs and solves the resulting ILPs using Gurobi [26].

Reference and tested cores. The reference core used was an extended version of the **Sodor** core [30], a simple single-cycle core, implementing the RV32I and the Zicsr (for CSRs) instruction set. We extended the Sodor core to also support the RV64I base instruction set and the M (multiply/divide) RISC-V instruction set. We used Sodor to find mismatches with the **Kronos** core [34], a 32-bit, three-stage, in-order core, tested in a recent fuzzing work [5], and the **BOOM** core [35], a 10-stage out-of-order processor. We downsized the BOOM core (PMP, FPU, and BPU disabled, and an ROB and issue queue size of 2) for model checking scalability. Porting INSIGHT to a different core requires a standard verification testbench, a signal-type mapping regex (see Section 4.2), and, optionally, a selection of relevant modules.

INSIGHT configuration. We classify the cores' signal types using simple regular expressions, and then use the mining strategy described in Section 4.2. We construct the ILP using the top 150 predicates, sorted by their individual formula score.

Studied Mismatches. Our analysis is centered around the comparison of the Sodor core with both Kronos and BOOM. Table 2 summarizes the studied underspecifications and bugs, where $US2$ only differs between Boom and Sodor. We study three real-world bugs in the Kronos core ($K1$-$K3$), and five bugs in the Boom core ($B1 - B5$). We limit our analysis of BOOM to bugs in the RISC-V extension sets shared by the Sodor core. To still include bugs that require more complex microarchitectural conditions to trigger in our analysis, we injected artificial bugs into the BOOM core. A case-study on Bug B3 in Appendix 10.3 elaborates on the potential security impact of functional correctness bugs. For Kronos, the security impact of these bugs is also discussed in [5].

Establishing Ground Truth. For both the model checking as well as the deduplication use case evaluation, we rely on an oracle to determine whether a certain CEX triggered a specific bug. To obtain this oracle, we generate variants of the same DUT: one with all known bugs fixed, and one per bug with only that bug left unfixed. We label a program as triggering "Bug X" if it causes a mismatch on the "Bug X-only" variant. If a CEX triggers a mismatch on the fully patched DUT variant, we manually investigate the cause to confirm it is not an unknown bug and to classify the exact type of ISA underspecification responsible.

6.2. Evaluating INSIGHT's Explanation

We first qualitatively evaluate INSIGHT's explanations along three types of mismatches: those caused by 1) ISA underspecifications, and 2) by bugs, broken down into 2a) simpler single-instruction bugs, and 2b) more complex microarchitectural bugs.

To evaluate the quality of INSIGHT's explanations and their sensitivity to the weight parameters λ and μ, we use INSIGHT to generate explanations for the studied bugs ($K1$–$K3$ and $B1$–$B5$) and underspecifications ($US1$–$US4$). To this end, we provide a manually crafted mismatch-triggering program to INSIGHT. For the BEX penalty λ, we sweep across values equivalent to 1%, 25%, 50%, 75%, and 100% of the total number of generated BEX traces. Similarly, for the predicate cost μ, we sweep across values representing 0.1%, 0.25%, 0.5%, 0.75%, and 1% of the total number of CEX traces. Much higher values for λ and μ result in degenerate explanations and were therefore excluded from this case-study. We summarize the results below. Table 4 in the Appendix depicts INSIGHT's verbatim output for example explanations $E1 \dots E5$. For readability,

TABLE 2. INJECTED MICROARCHITECTURAL BUGS IN KRONOS AND BOOM CORE

ID	Description	Source
	Kronos Bugs	
K1	A hazard-detection bug causes incorrect forwarding in a WaW Hazard.	Real-world [5]
K2	Writes to the `minstret` CSR, which tracks retired instructions, cause miscounting.	Real-world [5]
K3	Incorrect decoding of `FENCE` and `FENCE.I` instructions.	Real-world [5]
	BOOM Bugs	
B1	Retired instruction miscounting. (same as K2)	Real-world [5]
B2	Incorrect register file bypass implementation causes some instructions to read incorrect data.	Real-world [36]
B3	Off-by-one error in the ALU causes incorrect results during signed arithmetic.	Injected, see Appendix (Section 10.2)
B4	Incorrect forwarding logic for variable-latency instruction.	Injected, see Appendix (Section 10.2)
B5	Store-to-load forwarding selects the oldest matching store instead of the youngest.	Injected, see Appendix (Section 10.2)
	ISA Underspecifications vs. Sodor	
US1	Reset values of CSRs differ	ISA underspec.
US2	Behaviour upon executing misaligned branches differ.	ISA underspec.
US3	Reads to previously written to CSRs return different values	ISA underspec.
US4	Sodor core does not trap on `wfi` instruction	ISA underspec

we replaced magic numbers with their semantic meaning and abbreviated module names.

Influence of parameter choices. We first summarize the impact of parameter choices for λ and μ. Explanations for mismatches that do not experience masked CEX are resilient to parameter choices. For other explanations, the BEX penalty λ parameter can be used to steer INSIGHT towards over- or underfitting (depending on the user's preference), but a wide range of acceptable parameter choices exists. The predicate cost μ has a less pronounced effect, with $\mu \in \{0.25\%, 0.5\%, 0.75\%\}$ leading to similar state formulas. This effect can be explained by the fact that generalized explanations themselves tend to consist of only a limited number of predicates. Including more predicates risks covering less CEX, which naturally leads to short solutions. Without any predicate cost, INSIGHT can include predicates that overfit to the generated examples by using `NeqPred` predicates to exclude BEX traces from the set of covered traces without excluding any CEX. Discouraging this behavior (and not overpenalizing it) is sufficient. Beyond that, the predicate cost μ has limited practical impact in our experiments.

Observation (1): The BEX penalty λ and the predicate cost μ parameters can steer INSIGHT toward over- or underfitting. Moderate BEX penalty $\lambda \in \{25\%, 50\%, 75\%\}$ and predicate cost $\mu \in \{0.25\%, 0.5\%\}$ give intuitive explanations and are recommended for effective bug-finding.

Handling of ISA underspecification. We observe that INSIGHT describes underspecifications by leveraging either signals from the reference core or the DUT's CSR file. For instance, INSIGHT describes $US1$ through a formula stating that Sodor is currently executing a CSR-type instruction, and the target CSR address matches one of the CSRs that behave differently across cores, and the write-back destination is a valid register. This captures precisely the CSR operations where the two cores diverge (E1). INSIGHT explains mismatch $US2$ by leveraging Sodor's internal `inst_misaligned` signal, producing a single-predicate formula that asserts this signal equals one (E2).

Single-instruction bugs. Similar to underspecifications, mismatches caused by single instructions can be captured either through signals of the single-cycle reference core or through the conditions in the DUT that occur upon their execution.

For instance, INSIGHT describes $K3$ through Sodor's internal signals: It occurs when the opcode and funct3 point to a fence instruction, but the `rs1` field is unequal to 0, which triggers a difference in Kronos' and Sodor's decoding (E3).

Complex bugs. INSIGHT describes more complicated pipelining bugs by describing the microarchitectural conditions under which they occur.

For instance, INSIGHT describes $K1$ by conjoining three predicates over Kronos's pipeline signals: the decoder's `rs1` forwarding path is active, the register pending in the hazard control unit matches the register being read in the fetch stage, and the fetch stage is in its `FETCH` state. This captures the scenario where a read-after-write-after-write hazard occurs and the pipeline attempts to forward a value for rs1 while a prior write to the same register is still pending (E4).

As another example, for bug $B5$, INSIGHT synthesizes an explanation that conjuncts the following predicates (E5): there is a store to load forward match, both the first and second entry in the LSU match, and the second is being forwarded, and the register types of the storing and loading instructions match, the instruction is not squashed, and can execute. Multiple such explanations can be disjuncted to fully describe the LSU bug (see Section 7).

Results (RQ 1): INSIGHT can synthesize explanations for a wide variety of bugs and ISA underspecifications.

6.3. Effectiveness in Model Checking Unblocking

To assess whether INSIGHT helps model checking uncover more distinct bugs, we implemented the workflow of Algorithm 1 in Section 5 using JasperGold [37]. We use JasperGold to generate counterexamples to the assertion that, for any instruction sequence of bounded length, the DUT and the reference core produce identical commit logs.

We execute Algorithm 1 (INSIGHT-guided model checking) multiple times in an incremental fashion, each time increasing the timeout and the length of symbolic instructions. Once the added assumptions cause JasperGold to timeout, we increase the number of symbolic instructions and the timeout until we exhaust a budget of 12 hours of total runtime (counting INSIGHT's runtime itself, too). We implement the following optimization for BOOM when using INSIGHT: we first restrict JasperGold to search all combinations of CSR-type instructions (of maximum length 2) that lead to a mismatch until timeout. Then, we lift that restriction to search over all instruction types. Note that this only uncovers mismatch-triggering programs faster without harming generalizability.

We compare INSIGHT-guided model checking to a baseline (running for 24h for a conservative estimate) that merely excludes the exact retrieved counterexample (CEX) by adding an assumption. Then, the baseline reruns the query, with the received CEX excluded from the model checker's search space. For INSIGHT-guided model checking, we compare two moderate BEX penalties ($\lambda \in \{25\%, 50\%\}$), while fixing the predicate cost to $\mu = 0.5\%$, given that we did not observe significant changes to other moderate μ choices in our qualitative case study. Figures 8 and 9 plot the cumulative number of distinct underspecifications and bugs over time (classified per Table 2).

On both BOOM and Kronos, the baseline quickly converges to rediscovering the same issues. It discovers $US1$ on Kronos, and $US1$ and $B3$ on BOOM. After that, JasperGold reports different manifestations of the identified underspecification and mismatches until the time budget is exhausted. In contrast, INSIGHT-guided runs surface more distinct underspecifications and bugs in substantially less time. For Kronos, both the $\lambda = 25\%$ and $\lambda = 50\%$ configurations identify all bugs and underspecifications. For BOOM, both configurations find all mismatches except $B5$. However, we ensured that this is not due to INSIGHT synthesizing underfitting explanations: no synthesized explanation excludes a $B5$ counterexample, pointing to the model checker's scalability limits as the cause of these misses.

Results (RQ 2): INSIGHT can unblock model checkers that are otherwise stuck on rediscovering manifestations of the same mismatch. This leads to more bugs and ISA underspecifications being discovered in less time.

Table 3 reports per-run statistics. INSIGHT's runtime cost is amortized through more uncovered distinct mismatches. In addition, runs with lower BEX penalty use fewer JasperGold queries (since they overfit less).

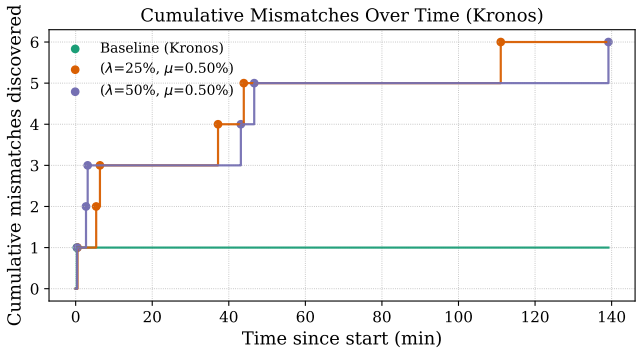


Figure 8. Number of underspecifications and bugs found vs. time for Kronos.

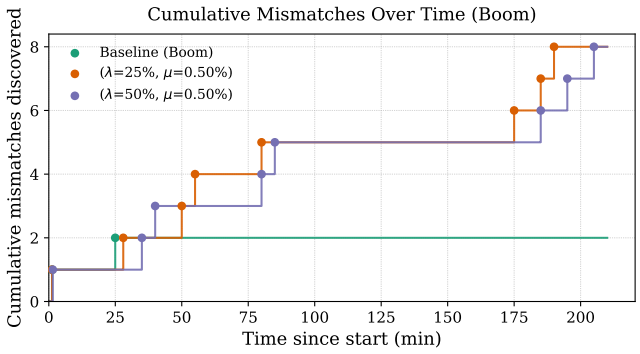


Figure 9. Number of underspecifications and bugs found vs. time for BOOM.

6.4. Using INSIGHT for Deduplication

We evaluate INSIGHT's deduplication by clustering mismatch-triggering programs generated by the Cascade fuzzer [5] for the Kronos and BOOM cores using Algorithm 2. We cluster a total of 8,165 mismatch-triggering programs triggering bugs $K1 - K3$ on Kronos and 930 mismatch-triggering programs triggering bugs $B1 - B5$ on BOOM. Each mismatch-triggering program consists of up to several thousand instructions.

We compare INSIGHT's clusters to a ground truth clustering, obtained using the ground truth oracle described Section 6.1. If two programs trigger the same mismatch according to the ground truth oracle, we cluster them together in the ground truth clustering. Note that a single program may trigger a mismatch on multiple versions of the same core: a program that contains both a triple-hazard bug ($K1$) and an incorrectly decoded `FENCE` instruction ($K3$) will trigger a mismatch on the designs injected with bugs $K1$ and $K3$, respectively.

To obtain INSIGHT's clustering, we execute Algorithm 2 to compute INSIGHT's explanations and their covered counterexamples. We then evaluate the quality of INSIGHT's clustering by measuring 1) to what extent INSIGHT reduces the overhead related to manual counterexample classification, and 2) how often it groups together counterexamples that

TABLE 3. INSIGHT-GUIDED RUNS VS. BASELINE COMPARISION. THE BEX PENALTY λ AND PREDICATE COST μ ARE STATED IN PERCENT OF TRACES GENERATED.

Config	#Queries/ mismatch	Runtime (% INSIGHT)	Discovered mismatches
Baseline			
baseline(Kronos)	4932	24h (100% JG)	US1
baseline(Boom)	697.5	24h (100% JG)	B3; US1
Kronos			
Kronos(λ=25%, $\mu = 0.5\%$)	4.33	12h (4%)	K1-K3; US1; US3; US4;
Kronos(λ=50%, $\mu = 0.5\%$)	4.83	12h (5%)	K1-K3; US1;US3; US4;
Boom			
Boom($\lambda = 25\%$, $\mu = 0.5\%$)	3.1	12h (9%)	B1-B4; US1-US4
Boom($\lambda = 50\%$, $\mu = 0.5\%$)	3.4	12h (11%)	B1-B4; US1-US4

do not appear to trigger the same bug.

We count programs that are covered by the same explanation (and therefore, are clustered together by INSIGHT) as a True Positive (TP) if they also trigger the same bug according to the ground truth oracle, and as a False Positive (FP) otherwise. We define the False Positive Rate (FPR) as $\text{FPR} = \frac{FP}{FP+TP}$, i.e., the fraction of mismatch-triggering programs wrongly grouped together out of those grouped together by the same explanation. A lower FPR indicates that INSIGHT's explanations more precisely group programs that trigger the same bug in the ground truth. We sweep the BEX penalty $\lambda \in \{1\%, 25\%, 50\%, 75\%, 100\%\}$ while keeping predicate cost fixed to $\mu = 0.5\%$ and, for each run, we compute the value of the FPR metric. For each run, we also state the number of synthesized state formulas, which is the number of clusters a verification engineer would need to investigate.

Figure 10 and Figure 11 show the value of the FPR metric and the number of explanations synthesized by INSIGHT for each value of λ for bugs $K1-K3$ on the Kronos core and bugs $B1-B5$ on the BOOM core, respectively. On both cores, the number of explanations synthesized by INSIGHT increases with higher BEX penalty, suggesting that higher BEX penalties produce a finer-grained clustering. The intermediate penalty values offer a balanced configuration, with an FPR of 18.6% and 10 clusters for $\lambda = 75\%$ on Kronos, and an FPR of 12% and 23 clusters for $\lambda = 50\%$ on BOOM. We observe that the number of synthesized explanations with these optimal values for λ is higher than the number of classes appearing in the ground truth. This behavior can be explained by the fact that some bugs require multiple (disjunct) state formulas to be fully described.

Results (RQ 3): INSIGHT can effectively deduplicate mismatch-triggering programs generated by a fuzzer, reducing a large set of counterexamples into a small number of clusters and a low FPR value.

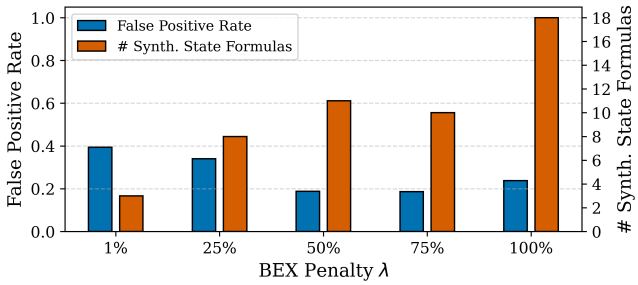


Figure 10. Results of the de-duplication algorithm applied to testcases generated by Cascade for bugs $K1-K3$.

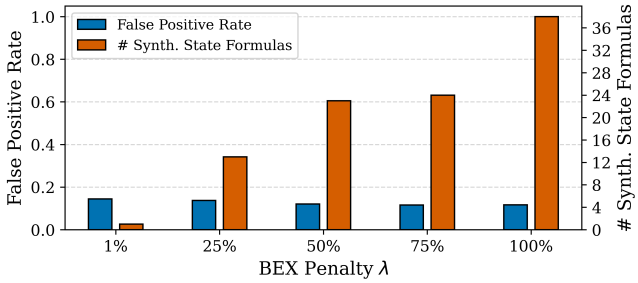


Figure 11. Results of the de-duplication algorithm applied to testcases generated by Cascade for bugs B1–B5.

7. Discussion

We now discuss some of INSIGHT's limitations and point to potential future directions.

Synthesizing Disjunctions. INSIGHT assumes that generalized explanations are conjunctions of basic predicates. Some mismatch causes, however, require multiple disjuncted state formulas to be explained. For instance, consider the WaW-Hazard bug from Section 3. This WaW hazard might manifest if a hazard exists in either the first source register (rs1) or the second source register (rs2) of a receiving instruction. Explaining this bug completely would therefore require a disjunction of two state formulas: one for the `rs1` hazard and one for the `rs2` hazard.

INSIGHT can address these disjunctive cases depending on the use case, though this remains a notable limitation.

In the model checking use case, disjunctions are created implicitly. A bug that requires disjunctions will be automatically excluded from the checker's search space across multiple iterations (see Algorithm 1, line 3).

For testcase deduplication, each state formula will form its own cluster. Therefore, bugs that require multiple disjunctions to be fully explained will increase the number of clusters, with each disjuncted state formula forming its own cluster.

Single-cycle State Formula. INSIGHT represents explanations as single-cycle state formulas. This formulation can still capture bugs that require multiple cycles to trigger, provided the relevant history is implicitly encoded in the core's internal signals at the cycle of interest, e.g., as described

in the Load-Store Queue example in Section 3.1. However, there do exist cases where a single-cycle state formula is insufficient. For instance, consider a program that writes an illegal value to a CSR and later reads it back. The RISC-V specification permits an implementation to return any legal value upon reading a CSR that was previously written with an illegal value. This divergence in returned values can cause a mismatch between two implementations that return different legal values. This mismatch only manifests upon the subsequent read, yet at the time of the read, neither core retains an internal signal recording that the prior write was illegal. Therefore, no single-cycle snapshot can perfectly capture the condition "read-after-prior-illegal-write to this CSR." In such inherently temporal cases, INSIGHT must over- or underfit: it synthesizes an imperfect explanation, e.g., describing all reads of the target CSR, or writes of an illegal value to the target CSR. Alternatively, one can introduce ghost code [38] into the reference core to handle these corner cases, e.g., by adding an internal flag that tracks which CSRs have been written to with illegal values.

Imperfect explanations. We note that even imperfect explanations remain useful. A mildly overfitting explanation might result in additional model checking queries or yield extra clusters during deduplication, but still leads to a reduction in computation time and manual effort. Mild underfitting explanations introduce the risk of missing bugs. However, the verification engineer will realize this after fixing other distinct mismatches and rerunning Algorithm 1.

8. Related Work

Invariant learning techniques have been successfully used to find loop invariants [22], [23], [39], [40], [41], modular summaries for verifying secure information flow [42], and environment invariants for hardware verification [43]. A series of recent works leverage invariant synthesis for hardware security verification [44], [45], [46]. Prior work also proposed using ILP solvers to synthesize inductive invariants [47], separating reachable from bad states.

INSIGHT differs from these works in two aspects. We are the first to find generalized explanations for mismatches between processors. In contrast to decision tree formula learning methods (e.g., ICE [23]), we formulate explanation synthesis as a constrained covering optimization problem that can leverage the power of modern ILP solvers.

Multiple recent works propose approaches for (fuzz-) testing a processor by comparing it against a golden reference implementation [6], [14], [16], [19], [48], [49], [50]. MorFuzz and DiffTest execute the DUT in co-simulation with a reference implementation, and rely on hardcoded rules to identify mismatches caused by underspecification. To avoid known bugs, existing works rely on hardcoded rules or heuristics to deduplicate [5], [19]. Prior work has pointed out that these rules can cause fuzzers to miss bugs [7]. The Cascade fuzzer [5] constructs programs such that bugs in the tested processor would result in a non-terminating run, thereby avoiding the need for a reference implementation.

9. Conclusion

We presented INSIGHT, a framework that learns generalized, machine-readable explanations for mismatches between a processor implementation and a reference model. By reformulating the problem of explanation synthesis as a constrained covering optimization problem, INSIGHT can harness the power of ILP solvers, leading to efficient explanation synthesis in practice. INSIGHT's explanations generalize beyond the generated examples and can be used to guide model checkers and deduplicate a set of mismatch-triggering programs. We demonstrated INSIGHT's efficacy in both these use cases.

Acknowledgements

Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of funding entities. Alessandro Bertani was supported by the Roberto Rocca Doctoral Fellowship during his research stay at the Massachusetts Institute of Technology. This material is based upon work supported by the Department of the Air Force under Air Force Contract No. FA8702-15-D-0001 or FA8702-25-D-B002. This work was in part supported by NSF grants 2422053, CNS 2046359, and CCF 2422052. This work was also supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] D. Graham, "RISC-V verification and implications of the 5:1 ratio of DV to design engineers," Apr. 2023, Accessed: 2026-03-10. [Online]. Available: https://www.tessolve.com/verification-futures/vf2023-uk/risc-v-verification-and-implications-of-the-51-ratio-of-dv-to-design-engineers/

[2] Tavis Ormandy. Zenbleed. Accessed: 2026-03-10. [Online]. Available: https://lock.cmpxchg8b.com/zenbleed.html

[3] D. Moghimi, "Downfall: Exploiting speculative data gathering," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 7179–7193. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/moghimi

[4] Tavis Ormandy. Reptar. Accessed: 2026-03-10. [Online]. Available: https://lock.cmpxchg8b.com/reptar.html

[5] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: CPU fuzzing via intricate program generation," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 5341–5358. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/solt

[6] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3219–3236. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/kande

[7] M. Bölcskei, F. Solt, K. Ceesay-Seitz, and K. Razavi, "Encarsia: evaluating cpu fuzzers via automatic bug injection," in *Proceedings of the 34th USENIX Conference on Security Symposium*, ser. SEC '25. USA: USENIX Association, 2025.

[8] F. Solt, P. Jattke, and K. Razavi, "RemembERR: Leveraging microprocessor errata for design testing and validation," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 1126–1143.

[9] A. Reid, R. Chen, A. Deligiannis, D. Gilday, D. Hoyes, W. Keen, A. Pathirane, O. Shepherd, P. Vrabel, and A. Zaidi, "End-to-end verification of processors with ISA-Formal," in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 42–58.

[10] L.-E. Ploix, A. Armstrong, T. Melham, R. Lin, H. Wang, and A. Courtney, "Comprehensive formal verification of observational correctness for the cheriot-ibex processor," *arXiv preprint arXiv:2502.04738*, 2025.

[11] Y. Lee and H. Cook. RISC-V Torture Test Generator. Accessed: 2026-03-10. [Online]. Available: https://github.com/ucb-bar/riscv-torture

[12] OpenHW Group. FORCE-RISCV. Accessed: 2026-03-10. [Online]. Available: https://github.com/openhwgroup

[13] V. Herdt, D. Große, E. Jentzsch, and R. Drechsler, "Efficient cross-level testing for processor verification: A RISC-V case-study," in *2020 Forum for Specification and Design Languages (FDL)*. IEEE, 2020, pp. 1–7.

[14] J. Xu, Y. Liu, S. He, H. Lin, Y. Zhou, and C. Wang, "MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1307–1324. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/xu-jinyan

[15] B. Pal, A. Sinha, P. Dasgupta, P. Chakrabarti, and K. De, "Hardware accelerated constrained random test generation," *IET Computers & Digital Techniques*, vol. 1, pp. 423–433, 2007. [Online]. Available: https://digital-library.theiet.org/doi/abs/10.1049/iet-cdt%3A20070016

[16] F. Thomas, E. G. Arribas, L. Hetterich, D. Weber, L. Gerlach, R. Zhang, and M. Schwarz, "RISCover: Automatic discovery of user-exploitable architectural security vulnerabilities in closed-source risc-v cpus," *ghostwriteattack.com*, 2025.

[17] Y. Xu, Z. Yu, D. Tang, G. Chen, L. Chen, L. Gou, Y. Jin, Q. Li, X. Li, Z. Li, J. Lin, T. Liu, Z. Liu, J. Tan, H. Wang, H. Wang, K. Wang, C. Zhang, F. Zhang, L. Zhang, Z. Zhang, Y. Zhao, Y. Zhou, Y. Zhou, J. Zou, Y. Cai, D. Huan, Z. Li, J. Zhao, Z. Chen, W. He, Q. Quan, X. Liu, S. Wang, K. Shi, N. Sun, and Y. Bao, "Towards Developing High Performance RISC-V Processors Using Agile Methodology," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1178–1199.

[18] A. Waterman and K. Asanovic, "The RISC-V instruction set manual, volume i: Unprivileged ISA document, version 20190608-base-ratified," *RISC-V Foundation, Tech. Rep*, 2019.

[19] L. Wu, M. Rostami, H. Li, J. Rajendran, and A.-R. Sadeghi, "GenHuzz: An efficient generative hardware fuzzer," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 1787–1805.

[20] S. Berezin, E. Clarke, A. Biere, and Y. Zhu, "Verification of out-of-order processor designs using model checking and a light-weight completion function," *Formal Methods in System Design*, vol. 20, no. 2, pp. 159–186, 2002.

[21] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 24, no. 1, pp. 1–24, 2018.

[22] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "ICE: A robust framework for learning invariants," in *Proceedings of the International Conference on Computer Aided Verification (CAV)*, 2014, pp. 69–87.

[23] P. Garg, D. Neider, P. Madhusudan, and D. Roth, "Learning invariants using decision trees and implication counterexamples," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 499–512. [Online]. Available: https://doi.org/10.1145/2837614.2837664

[24] C. Flanagan and K. R. M. Leino, "Houdini, an Annotation Assistant for ESC/Java," in *FME 2001: Formal Methods for Increasing Software Productivity*, J. N. Oliveira and P. Zave, Eds. Springer Berlin Heidelberg, 2001, vol. 2021, pp. 500–517.

[25] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *FMCAD*. IEEE, 2013, pp. 1–17.

[26] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2025, accessed: 2026-03-10. [Online]. Available: https://www.gurobi.com

[27] IBM Corporation, *IBM ILOG CPLEX Optimization Studio 22.1.2: CPLEX User's Manual*, 2024, accessed: 2026-03-10. [Online]. Available: https://www.ibm.com/docs/en/icos/22.1.2?topic=optimizers-users-manual-cplex

[28] MIT, *lp_solve Reference Guide*, MIT Department of Mathematics, 2024, accessed: 2026-03-10. [Online]. Available: https://web.mit.edu/lpsolve/doc/

[29] R. M. Karp, "Reducibility among combinatorial problems," in *50 Years of Integer Programming 1958-2008: from the Early Years to the State-of-the-Art*. Springer, 2009, pp. 219–241.

[30] Berkeley Architecture Research. (2021) Sodor core. Accessed: 2026-03-10. [Online]. Available: https://chipyard.readthedocs.io/en/stable/Generators/Sodor.html

[31] M. J. Kearns and U. Vazirani, *An Introduction to Computational Learning Theory*. The MIT Press, 08 1994. [Online]. Available: https://doi.org/10.7551/mitpress/3897.001.0001

[32] C.-T. Ho, H. Ren, and B. Khailany, "VerilogCoder: Autonomous Verilog Coding Agents with Graph-based Planning and Abstract Syntax Tree (AST)-based Waveform Tracing Tool." *URL https://arxiv.org/abs/2408.08927*, 2024.

[33] W. Snyder, "Verilator and systemperl," in *North American SystemC Users' Group, Design Automation Conference*, 2004.

[34] S. Pinto. Kronos core. Accessed: 2026-03-10. [Online]. Available: https://github.com/SonalPinto/kronos

[35] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, vol. 5. International Symposium on Computer Architecture Valencia, Spain, 2020, pp. 1–7.

[36] RISC-V BOOM Contributors, "Fix issue where instructions can read stale data from register file bypass," GitHub Pull Request #376, 2020, accessed: 2026-04-16. [Online]. Available: https://github.com/riscv-boom/riscv-boom/pull/376

[37] Cadence, "Jasper formal property verification app," https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/formal-property-verification-app.html, accessed: 2026-03-10.

[38] J.-C. Filliâtre, L. Gondelman, and A. Paskevich, "The spirit of ghost code," *Formal Methods in System Design*, vol. 48, no. 3, pp. 152–174, 2016.

[39] S. Padhi, R. Sharma, and T. D. Millstein, "Data-driven precondition inference with learned features," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2016, pp. 42–56.

[40] G. Fedyukovich, S. Prabhu, K. Madhukar, and A. Gupta, "Quantified Invariants via Syntax-Guided Synthesis," in *CAV, Part 1*, ser. Lecture Notes in Computer Science, vol. 11561. Springer, 2019, pp. 259–277.

[41] K. Huang, X. Qiu, P. Shen, and Y. Wang, "Reconciling enumerative and deductive program synthesis," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 1159–1174.

[42] L. Pick, G. Fedyukovich, and A. Gupta, "Automating modular verification of secure information flow," in *2020 Formal Methods in Computer Aided Design (FMCAD)*, 2020, pp. 158–168.

[43] H. Zhang, W. Yang, G. Fedyukovich, A. Gupta, and S. Malik, "Synthesizing Environment Invariants for Modular Hardware Verification," in *Verification, Model Checking, and Abstract Interpretation*, D. Beyer and D. Zufferey, Eds. Springer International Publishing, 2020, vol. 11990, pp. 202–225.

[44] D. Neider, S. Saha, P. Garg, and P. Madhusudan, "Sorcar: Property-driven algorithms for learning conjunctive invariants," in *International Static Analysis Symposium*. Springer, 2019, pp. 323–346.

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

[46] S. Dinesh, Y. Zhu, and C. W. Fletcher, "H-houdini: Scalable invariant learning," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2025, pp. 603–618.

[47] S. Bhatia, S. Padhi, N. Natarajan, R. Sharma, and P. Jain, "OASIS: ILP-Guided Synthesis of Loop Invariants," in *NeurIPS 2020 Workshop on Computer-Assisted Programming*, 2020.

[48] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1286–1303.

[49] C. Chen, R. Kande, N. Nguyen, F. Andersen, A. Tyagi, A.-R. Sadeghi, and J. Rajendran, "HyPFuzz: Formal-Assisted processor fuzzing," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1361–1378. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/chen-chen

[50] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. B. Taylor, M. Egele, and A. Joshi, "Processorfuzz: Processor fuzzing with control and status registers guidance," in *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2023, pp. 1–12.

[51] OWASP Foundation. Heartbleed bug. https://owasp.org/www-community/vulnerabilities/Heartbleed_Bug. Accessed: 2026-03-10.

[52] N. Carlini and D. Wagner, "ROP is Still Dangerous: Breaking Modern Defenses," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 385–399.

Ethics Considerations

The bugs discussed in this paper, except for those artificially injected into the cores under test, were already publicly known from prior works. Our work does not identify or expose novel vulnerabilities; therefore, no disclosure actions were necessary in accordance with standard responsible research practices.

LLM Usage Considerations

LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.

10. Appendix

10.1. Optimizations Employed by INSIGHT

To improve scalability and solver efficiency, INSIGHT incorporates several optimizations. To describe these optimizations, we introduce the notion of a state's *support*, defined as the set of predicates that cover the state. Formally,

$$supp(s) = \{\, p \in \mathcal{P} \mid s \in cov(p)) \,\}.$$

We use this notion to derive multiple optimizations for INSIGHT's algorithm.

Pre-processing the instance before passing it to the solver. We implement the following optimizations:

1) **Merging indistinguishable states:** If for two states $supp(s_2) = supp(s_1)$, they are indistinguishable with respect to the current predicate selection. INSIGHT therefore replaces all occurrences of s_1 with s_2 before constructing the instance for the ILP solver.
2) **Removing equivalent predicates** If $cov(p_1) = cov(p_2)$, we remove p_1 before constructing the ILP instance.
3) **Removing dominated predicates** Let S_{CEX} be the states occurring only in CEX traces, S_{BEX} be the states only occurring in BEX traces. If for two predicates p_1 and p_2 it holds that $(cov(p_1) \cap S_{CEX}) \subset cov(p_2) \cap S_{CEX})$, while at the same time $cov(p_2) \cap S_{BEX} \subset cov(p_1) \cap S_{BEX}$, then we can remove p_1, since selecting p_1 cannot lead to a better formula score than selecting p_2.

10.2. Injected Bugs

B3: B3 injects an off-by-one arithmetic fault, omitting the two's-complement +1 in subtraction: the ALU computes in1 + ($\sim$in2) instead of in1 - in2. This can cause incorrect outputs for subtraction and comparison instructions.

B4: B4 forces the divide functional unit onto the fast-wakeup/bypass path, causing the core to fast-wake dependents at dispatch even though DIV is variable-latency. This causes a potential stale data read.

B5: This bug changes the LSU so that it always returns the oldest store instead of the youngest in case of a store-load-forwarding match.

10.3. Security Impact Case Study

To demonstrate the security implications of microarchitectural defects, we present a case study of the software exploitability of the injected ALU off-by-one error (Bug B3). As described in Appendix 10.2, this bug omits the two's-complement $+1$ during subtraction operations, causing the ALU to compute $in1 + (\sim in2)$ instead of $in1 - in2$. Consequently, the hardware produces incorrect outputs for both subtraction and comparison instructions.

Attacks could exploit software running on hardware affected by this bug to bypass memory safety checks. In

scenarios involving array reads, this hardware-induced bypass leads to out-of-bounds memory read vulnerabilities, yielding data leaks similar to, e.g, the Heartbleed bug [51]. When applied to array writes, this bug enables arbitrary out-of-bounds memory writes. Such memory corruption vulnerabilities frequently allow attackers to overwrite adjacent data structures, achieving control-flow hijacking, privilege escalation, or remote code execution.

Consider the function in Program 4, as shown in Figure 12. When compiled to RISC-V assembly with GCC (optimization level 3), the conditional statement can rely on a branch instruction, such as `bge` (Branch If Greater Equal), to evaluate the relationship between the `index` and `length` registers, see Program 5, shown in Figure 13.

Program 4 (Standard C bounds check)

```
void secure_write(int* array, int length,
  int index, int value) {
  if (index < length) {
    array[index] = value;
  } else {
    // Out-of-bounds error handling
    return;
  }
}
```

Figure 12. Standard C bounds check for an array write.

Program 5 (Compiled RISC-V assembly)

```
// a2 is index, a1 is length
  bge  a2, a1, .L1
  slli a2, a2, 2
  add  a0, a0, a2
  sw   a3, 0(a0)
.L1:
  ret
```

Figure 13. Compiled RISC-V assembly for the bounds check.

Under correct architectural behavior, an out-of-bounds access attempt (e.g., providing `index = 8` for an array of length = 8) results in the condition $8 < 8$ evaluating to false, correctly redirecting execution to the error handler.

However, on the BOOM core with bug B3 injected, the `bge` instruction relies on the buggy ALU implementation to perform the underlying subtraction. The hardware evaluates the branch condition as follows: $\texttt{BranchCond} = 8 + (\sim 8) = 8 + (-9) = -1$.

Because $-1 < 0$, the microarchitecture incorrectly determines that the branch condition is not satisfied. Consequently, the execution flow bypasses the software-defined boundaries and executes the out-of-bounds memory write. An attacker can exploit this issue to overwrite adjacent memory structures, potentially modifying critical control data (such as return addresses or function pointers) to achieve e.g., code execution [52].

10.4. INSIGHT Formula Examples

Table 4 provides examples of INSIGHT's explananations, for the bugs described in Table 2.

TABLE 4. CANDIDATE EXPLANATIONS PRODUCED BY INSIGHT UNDER VARYING PENALTY SETTINGS. BEX PENALTY λ AND PREDICATE COST μ ARE SHOWN AS % OF GENERATED BEX AND CEX TRACES, AS DESCRIBED IN SECTION 6. FORMULAS ARE VERBATIM FROM INSIGHT'S OUTPUT. FOR READABILITY, MAGIC NUMBERS ARE REPLACED BY THEIR SEMANTIC MEANING, AND MODULE NAMES ARE ABBREVIATED.

ID	Bug	BEX λ	Pred. cost μ
Underspecifications (vs. Sodor)			
E1	US1	50	0.5
`sodor.dec.opcode==SYSTEM` ∧ `sodor.dec.immI==MARCHID` ∧ `sodor.wb.addr`≠0			
E2	US2	50	0.5
`sodor.ctl.inst_misaligned` = 1			
K3 (invalid `FENCE`)			
E3	K3	50	0.5
`sodor.dec.opcode==FENCE` ∧ `sodor.dec.funct3==0` ∧ `sodor.dec.rs1`≠0			
K1 (RaWaW Hazard)			
E4	K1	25	0.5
`kronos.dec.rs1_forward=1` ∧ `kronos.hcu.rpend==kronos.if.rf.reg_rs1` ∧ `kronos.if.state==FETCH`			
B5 (BOOM: store-to-load fwd. / ordering)			
E5	B5	50	0.5
`boom.lsu.ldst_fwd_match_0_2==1` ∧ `boom.lsu.ldst_fwd_match_0_1==1` ∧ `boom.lsu.fire_cond_205==1` ∧ (`boom.lsu.forward_idx==2`)=1 ∧ `boom.lsu.not_killed_by_branch==1` ∧ `boom.rf.lrs2_rtype=boom.lsu.stq2_lrs2_rtype`			

Signal alias legend. **`sodor.*`**: Sodor reference core; **`kronos.*`**: Kronos DUT; **`boom.*`**: BOOM core. `*.dec.*`: decoder fields; `dec.funct3`/`funct7`: RISC-V instruction fields; `*.hcu.*`: hazard control unit; `*.if.*`: fetch stage; `*.if.rf`: register file; `sodor.ctl.*`: control path; `*lrs_rtype*`: register type.

11. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

11.1. Summary of Paper

This paper develops a method for synthesizing machine-readable explanations for hardware bugs induced by mismatches between implementations and specifications. Using their technique, the authors show that their synthesis technique can effectively guide model checkers and fuzzers.

11.2. Scientific Contributions

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

11.3. Reasons for Acceptance

1) This paper provides a valuable step forward in an established field. On their own, model checkers are powerful, but cannot discern a truly new bug from an old one they have already found. This technique provides a mechanism to make this distinction.

11.4. Noteworthy Concerns

1) The technique's focus on single-state bugs rather than temporal bugs may be an issue for expressivity.