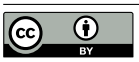
Exploiting Sophisticated Static Analysis for Verilog

QINLIN CHEN, Nanjing University, China
NAIREN ZHANG, Nanjing University, China
JINPENG WANG, Nanjing University, China
JIACAI CUI, Nanjing University, China
TIAN TAN*, Nanjing University, China
XIAOXING MA, Nanjing University, China
CHANG XU, Nanjing University, China
JIAN LU, Nanjing University, China
YUE LI*, Nanjing University, China

Static analysis has profoundly improved software quality over the past decades, evolving from compiler-integrated optimizations and simple linting to sophisticated analyses for bug detection, security, and program understanding. In contrast, static analysis for hardware remains underexploited, resembling the early state of software analysis. Most existing hardware static analyses are confined to compiler optimizations and linting, lacking the sophistication needed to uncover complex design flaws. Furthermore, we observe that many hardware bugs reported in recent literature could have been identified by sophisticated static analyses that account for hardware-specific semantics and data flow; however, such bug detection analyses are absent today. To exploit the untapped potential of sophisticated hardware analysis, we present a series of bug detection analyses for Verilog, the predominant hardware description language (HDL). Moreover, these analyses are built upon our fundamental analyses that capture essential hardware-specific characteristics—such as bit-vector arithmetic, register synchronization, and digital component concurrency—and enable the examination of hardware data and control flows. Together, these analyses form a well-organized analysis suite with a modular design, in which diverse fundamental analyses combine to support bug detection, hardware understanding, and other potential clients. To implement these analyses, we further offer dedicated infrastructure, including a Verilog front end, an intermediate representation (IR) for analysis, and an analysis manager. To validate the utility of our analyses, we applied them to real-world hardware projects. Unlike software, real-world hardware projects tend to contain fewer but harder-to-detect bugs, as they typically undergo extensive simulation and rigorous verification to prevent the prohibitive costs of hardware defects. Despite this, our preliminary experimental results are highly promising: applying these proposed analyses to popular real-world Verilog projects (averaging 1.5K+ GitHub stars) uncovered nine previously unknown bugs, all confirmed by developers; moreover, we successfully identified a total of 18 bugs beyond the capabilities of existing static analyses for Verilog bug detection (i.e., linters). These results underscore the transformative potential of sophisticated static analysis in hardware design. Our analysis suite and infrastructure are also highly reusable: on average, each bug-detection client built on our analysis suite requires about 270 LoC, compared to 5,700 LoC when

*Corresponding author.

Authors' Contact Information: Qinlin Chen, qinlinchen@smail.nju.edu.cn, Nanjing University, Nanjing, China; Nairen Zhang, nairen@smail.nju.edu.cn, Nanjing University, Nanjing, China; Jinpeng Wang, jpwang@smail.nju.edu.cn, Nanjing University, Nanjing, China; Jiacai Cui, jiacaicui@smail.nju.edu.cn, Nanjing University, Nanjing, China; Tian Tan, tiantan@nju.edu.cn, Nanjing University, Nanjing, China; Xiaoxing Ma, xxm@nju.edu.cn, Nanjing University, Nanjing, China; Chang Xu, changxu@nju.edu.cn, Nanjing University, Nanjing, China; Jian Lu, lj@nju.edu.cn, Nanjing University, Nanjing, China; Yue Li, yueli@nju.edu.cn, and all authors are also affiliated with the State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China.



ACM 2475-1421/2026/6-ART222
https://doi.org/10.1145/3808300

developed from scratch. By open-sourcing the entire system, involving substantial engineering effort (100K+ LoC), we aim to encourage further innovation and applications of sophisticated static analysis for hardware, hopefully fostering a similarly vibrant ecosystem that software analysis enjoys.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; *Development frameworks and environments*; • **Hardware** → *Functional verification.*

Additional Key Words and Phrases: Static Analysis, Verilog, Hardware, Framework

ACM Reference Format:
Qinlin Chen, Nairen Zhang, Jinpeng Wang, Jiacai Cui, Tian Tan, Xiaoxing Ma, Chang Xu, Jian Lu, and Yue Li. 2026. Exploiting Sophisticated Static Analysis for Verilog. *Proc. ACM Program. Lang.* 10, PLDI, Article 222 (June 2026), 29 pages. https://doi.org/10.1145/3808300

1 Introduction

In the software domain, static analysis is widely recognized as an effective technique for enhancing software quality. Over the past decades, numerous sophisticated static analyses have been developed to achieve various application tasks, such as detecting bugs [17, 18, 20, 34, 64–66, 116], identifying security vulnerabilities [9, 21, 46, 75], and aiding program understanding [73, 100].

Despite its proven successes in software, static analysis remains underutilized in hardware, particularly with Verilog, the predominant hardware description language (HDL) [45]. The current situation mirrors the early state of software static analysis, when its applications were limited to compiler-integrated optimizations and basic syntax checks (e.g., early linting tools), with more sophisticated software analyses, such as those for bug detection and security, yet to emerge. In hardware, static analysis techniques are primarily adopted by hardware synthesizers (e.g., Yosys [112]) for compiler optimization purposes; linting tools [26, 50, 88, 99] are popularly used to enforce code style or syntactic checks but lack the sophistication needed to detect deeper design flaws, such as improper register resets and hardware deadlocks.

Moreover, when we examine hardware bugs surveyed through commit histories from real-world programs and summarized by industrial hardware experts [29, 30, 78], we find that many can be exposed early in the design phase by more sophisticated hardware static analysis techniques that, to our knowledge, have yet to be explored. For example, a 2022 study [78] categorizes bugs in open-source FPGA projects by examining their git commit histories, resulting in a testbed of reproducible, complex real-world bugs. We find that most of these bugs can be detected by static analysis, provided the analysis carefully considers hardware program semantics such as data flow, control flow, concurrency, and synchronization. These bugs include issues like missing register resets, use of invalid signals, and hardware deadlocks, which elude linting approaches and tools.

Currently, hardware reliability typically relies on resource-intensive methods such as simulation-based testing and model checking [16, 49]; however, simulating a chip can take hours to days [12], and model checking is susceptible to state-space explosion [28], and thus can hardly scale to large programs. We argue that sophisticated Verilog analysis offers key advantages not shared by these methods: (1) it automatically analyzes the codebase, requiring minimal human involvement; (2) it scales to very large codebases comprising millions of lines of code; and (3) it provides early feedback during hardware design, which is particularly cost-effective since late-stage hardware bugs in production can lead to exponentially increasing remediation costs [7, 10].

To exploit the untapped potential of sophisticated hardware analysis, we design a series of *bug detection analyses* motivated by real-world bugs from prior studies [29, 30, 78] and insights from industrial hardware practitioners. Moreover, to establish a reusable foundation for future hardware analysis, we distilled a set of *fundamental analyses* from the process of designing bug detection analyses. On average, each fundamental analysis is reused by 9.5 (out of all 18) client analyses. These

fundamental analyses address essential hardware-specific characteristics—such as bit-vector arithmetic, register synchronization, and digital component concurrency—and enable examination of complex hardware control and data flows. In addition, we also build on these foundations to develop *hardware understanding analyses* that efficiently predict how synthesis tools map code to physical circuits (e.g., inferring clocks, registers, and reset signals). These clients not only provide useful information to support bug detection analyses, but also help developers understand the hardware synthesized from Verilog code and identify potential post-synthesis issues early, without requiring the time-consuming synthesis process. For example, synthesizing a million-line SoC [113] takes 65 minutes, whereas our hardware-understanding analyses complete in about 1 second on average.

Together, these analyses form a well-organized suite in which fundamental analyses collaborate to support hardware bug detection, hardware understanding, and future analysis clients. To our knowledge, no previous work has developed such a suite of static analyses for Verilog. This unique contribution draws on our sustained communication with industrial hardware practitioners [2, 3], in-depth understanding of Verilog formal semantics [25], and decade-long experience in foundational software static analysis [69–73, 80, 104, 105].

Then a natural question arises: how do our sophisticated hardware analyses differ from software analyses? Briefly, the formulation of hardware problems often diverges from their software analogues when hardware-specific behaviors such as synchronization and resetting are involved. These distinctions make hardware analyses fundamentally different. To clarify these differences, this paper presents a motivating analysis for detecting *missing-reset bugs* in Section 2. In software, a related issue is *use-before-initialization* (UBI), which frequently occurs in real-world programs, often leading to serious consequences and driving ongoing research for detecting UBI [65, 114]. We show how the missing-reset problem fundamentally differs from its software analogue, and describe our hardware-specific insights for designing a dedicated analysis to address it. For the remaining analyses, since it is infeasible to present them all in this paper, we provide an overview and highlight how our analyses interdepend and cooperate to support sophisticated hardware analyses in Section 3, using the missing-reset analysis as an example.

Since the analyses in our suite are interdependent and cooperative, the soundness and precision of one analysis can affect those of the analyses that depend on it. To maximize overall utility, we deliberately adopt different soundness–precision trade-offs for analyses playing different roles. Specifically, for general-purpose fundamental analyses, we prioritize soundness to ensure reliable reuse by a wide range of clients. For specific clients, we place greater emphasis on reducing excessive false positives to make the clients practically useful. Section 3 discusses more on these trade-offs.

In summary, this paper makes the following contributions:

- We exploit sophisticated static analysis for Verilog by introducing the first Verilog analysis suite, comprising both fundamental analyses addressing essential hardware characteristics and analysis clients for hardware bug detection and hardware understanding. We highlight the necessity of fundamental analyses and their collaboration to support various sophisticated clients and establish a reusable foundation for future hardware analysis research.
- We evaluate our analyses on a diverse set of real-world Verilog programs, including System-on-Chip (SoC) designs, encryption modules, communication protocols, and more. Unlike software, hardware tends to contain fewer but harder-to-detect bugs, as they typically undergo extensive simulation and rigorous verification to prevent the prohibitive costs of hardware defects [7, 10]. Despite this, our preliminary experiments exhibit very promising results: applying these proposed analyses to popular real-world Verilog projects (averaging over 1.5K stars) uncovered nine previously unknown bugs, all of which were confirmed by developers. Moreover, our analyses successfully identified 18 true bugs in total, accompanied with 67 false positive reports.

These bugs span eight categories as summarized in Table 1, while existing linting-based static analyses [26, 50, 88, 99] fail to detect all of them. These results demonstrate the transformative potential of sophisticated analysis for hardware.

- We develop *Qihe*, a Verilog static-analysis framework implemented from scratch in over 100K LoC. Qihe includes not only our analysis suite but also a dedicated infrastructure consisting of a Verilog front end, an IR for analysis, and an analysis manager. (The front end and IR are designed to work collaboratively, addressing specific analysis needs such as supporting incomplete programs, while the engineering-focused analysis manager enables developers to conveniently develop, integrate, and execute analyses.) Qihe is open-source and can be obtained from its website (https://qihe.pascal-lab.net); to facilitate the development of various analysis clients, we provide extensive documentation at https://qihe-docs.pascal-lab.net. We also release an artifact [22] to reproduce all experimental results, including all details of the identified bugs.

The growth of hardware analysis may not mirror the widespread success of software analysis. Yet, over 30 years ago, it was also hard to foresee that static analysis would help enhance software quality so significantly. Our work takes a preliminary step toward uncovering the potential of sophisticated hardware analysis. We hope it serves as a catalyst for collaboration between software and hardware communities to jointly shape the still-blurry yet promising future of hardware analysis.

2 Motivating Example

This section presents a motivating example that illustrates three key questions: (1) What do hardware bug-detection problems look like, and how do they differ from their software analogues? (2) How to design sophisticated hardware static analyses that leverage hardware-specific insights to address these problems? (3) What is the challenge in building sophisticated hardware static analysis?

Our motivating example is the *missing-reset* analysis, which detects a class of critical hardware bugs where registers fail to initialize properly during circuit reset, creating unstable system states that attackers could potentially exploit [6, 32, 78, 81]. It serves as a representative Verilog analysis in our analysis suite that not only handles diverse hardware-specific semantics—such as clocks, registers, and resets—that are uncommon in software analysis, but also demonstrates the level of sophistication required to detect real-world hardware bugs (summarized in Table 1) that elude existing static analyses for Verilog bug detection [26, 50, 88, 99].

Below, we first introduce the *missing-reset* problem in Verilog and distinguish it from the classical *use-before-initialization* (UBI) problem in software (Section 2.1). We then present the core insights underlying our missing-reset analysis for addressing this problem (Section 2.2). Finally, we use this analysis to discuss the key challenge in developing sophisticated hardware analyses (Section 2.3).

2.1 The Missing-Reset Problem in Verilog

2.1.1 Background. The *missing-reset* problem occurs when hardware *registers* lack proper *reset*. To prepare for our later formulation of the problem, we first provide the necessary Verilog background on registers and reset.

Registers. A hardware register is a digital circuit component that stores and updates values under the control of a driving clock signal alternating between 0 and 1; Verilog uses variables and `always` blocks to model hardware registers. As exemplified in the code snippet of Figure 1, `acc` is a hardware register whose driving clock is specified in the `always` block header (line 1) and whose value update logic is specified in the block body (lines 2–5). Functionally, this Verilog program describes an accumulator where register `acc` updates to 0 when `reset` is 1 (lines 2–3) and to `acc + in` otherwise (lines 4–5), whenever the driving clock signal `clock` transitions from 0 to 1,

known as a positive clock edge (as specified by the posedge keyword in line 1). To illustrate this dynamic behavior, the table in Figure 1 shows an example of how variable values evolve over time:

- The "Cycle" column enumerates clock cycles, where the number i indicates that this row shows the values of all variables during the i-th *clock cycle*, denoted Cycle_i hereafter. Here, a clock cycle is the time interval between consecutive positive edges of the clock signal `clock`.
- The "Input" columns provide the values of input variables such as `reset` and `in`, which are not defined within this code snippet but are externally supplied each cycle (e.g., by toggling a switch or pressing a button during circuit execution).
- The "`acc`" column shows the value of register `acc` over time. For simplicity, we use x_i to denote the value of variable x during Cycle_i. Initially, all registers hold undefined values at Cycle_0, i.e., $\texttt{acc}_0 = \texttt{undefined}$ in this example. During each cycle, lines 2–5 execute, and the *new register value takes effect in the next cycle*, reflecting the special semantics of Verilog's non-blocking assignment operator <=, which models clock-driven updates of hardware registers—unlike software assignments. For example, (1) during Cycle_0, `acc <= 0` (line 3) executes since $\texttt{reset}_0 = 1$, making $\texttt{acc}_1 = 0$; (2) during Cycle_1, `acc <= acc + in` (line 5) executes since $\texttt{reset}_1 = 0$, giving $\texttt{acc}_2 = \texttt{acc}_1 + \texttt{in}_1 = 0 + 2 = 2$; (3) during Cycle_2, the same update applies again, yielding $\texttt{acc}_3 = \texttt{acc}_2 + \texttt{in}_2 = 2 + 3 = 5$.

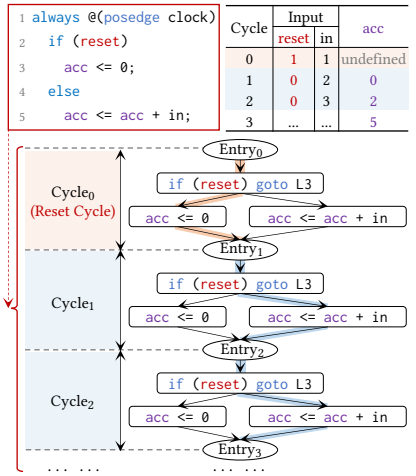


Fig. 1. A properly-reset register example.

For further clarity, the lower portion of Figure 1 presents a cycle-expanded control flow graph (CFG) that visualizes this dynamic behavior by highlighting execution paths in orange and blue. In this representation, Entry_i denotes entry to the `always` block body (lines 2–5) during Cycle_i.

Reset. The *reset signal* represents one of the most fundamental control signals critical for *reliable* behavior in digital design [111], ensuring circuits begin operation from a known initial state rather than the default undefined state at power-up. For example, in Cycle_0 of Figure 1 (highlighted in orange), $\texttt{reset}_0$ is set to 1, ensuring that normal operation of the accumulator begins with a determined $\texttt{acc} = 0$ instead of the undefined $\texttt{acc} = \texttt{undefined}$. We refer to the cycle in which the reset signal is active (i.e., $\texttt{reset} = 1$) as the *reset cycle*. This reset cycle is essential for reliable normal operation: only after that cycle can we guarantee that during any subsequent non-reset Cycle_i (where $i \geq 1$ and $\texttt{reset}_i = 0$), the accumulator *reliably* maintains the invariant $\texttt{acc}_{i+1} = \sum_{j=1}^{i} \texttt{in}_j$.

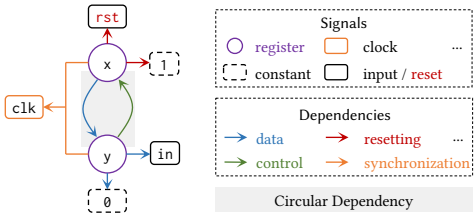
By convention, any *legal execution* of a circuit with reset capability should begin with a reset cycle to establish an initial state, followed by non-reset cycles for normal operation, as shown in Figure 1. This creates a clear division of responsibility: hardware developers should ensure their Verilog programs function reliably under all legal executions, while hardware users should perform legal executions (i.e., reset before normal operation). Consequently, the primary bugs of interest to developers are those that can occur during legal executions, as these fall within their expected domain of responsibility.

2.1.2 Problem Formulation. The *missing-reset* problem occurs on a register if there exists a *legal execution* path (i.e., a reset cycle followed by non-reset cycles, as illustrated in Figure 1) in which this register's undefined value is *infinitely often* used. "Infinitely often" is a standard term in

```
1 always @(posedge clk)
2   if (rst)
3     x <= 1;
4   else
5     x <= y;
6     y <= x ? in : 0;
```

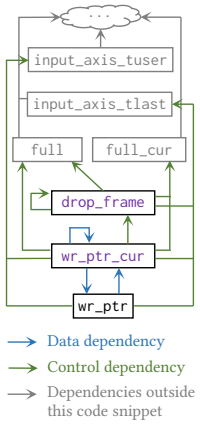
Cycle	Input		x	y
	rst	in		
0	1	0	undefined	undefined
1	0	1	1	undefined
2	0	2	undefined	1
3	0	1	1	undefined
4	...	...	undefined	1

(a) A simple missing-reset example.



(b) The hardware dependency graph of (a).

```
1  always @(posedge clk or posedge rst)
2    if (rst)
3      wr_ptr <= 0;
4      // drop_frame <= 0; <- missing reset
5      // wr_ptr_cur <= 0; <- missing reset
6    else
7      if (full | full_cur | drop_frame)
8        drop_frame <= 1;
9        if (input_axis_tlast)
10         wr_ptr_cur <= wr_ptr;
11         drop_frame <= 0;
12     else
13       wr_ptr_cur <= wr_ptr_cur + 1;
14       if (input_axis_tlast)
15         if (input_axis_tuser)
16           wr_ptr_cur <= wr_ptr;
17         else
18           wr_ptr <= wr_ptr_cur + 1;
```



(c) A real-world missing-reset example [78] containing the first two bugs (B01 & B02) in our evaluation (Table 1).

Fig. 2. Missing-reset examples and their hardware dependency graphs.

mathematics [38]. In our context, it means that for a register x, after any given Cycle_N, the undefined value of x will be used again in some later cycle Cycle_n where $n > N$. This problem is particularly dangerous because it implies that a register may unpredictably hold undefined values at any time, even during a legal execution, undermining hardware reliability and potentially leading to fatal functional flaws. In contrast, finite occurrences of undefined values are common and acceptable in hardware (e.g., in pipelined designs [110]), and such transient behavior should not be considered a bug.

Figure 2a illustrates a missing-reset example in which registers x and y both exhibit this problem, as shown in the table on the right. In the reset cycle Cycle_0 (shaded in orange), line 3 executes, updating x from `undefined` to 1 in Cycle_1, while y remains `undefined` because it is not assigned in this cycle. In each non-reset cycle Cycle_i ($i \geq 1$, shaded in blue), lines 5–6 execute, resulting in the value updates $x_{i+1} = y_i$ and $y_{i+1} = x_i \ ? \ \texttt{in}_i : 0$, according to the Verilog non-blocking assignment semantics introduced in Section 2.1.1. More concretely, after Cycle_1, we have $x_2 = y_1 = \texttt{undefined}$ and $y_2 = (x_1 \ ? \ \texttt{in}_1 : 0) = (1 \ ? \ 1 : 0) = 1$; after Cycle_2, we have $x_3 = y_2 = 1$ and $y_3 = (x_2 \ ? \ \texttt{in}_2 : 0) = (\texttt{undefined} \ ? \ 2 : 0) = \texttt{undefined}$; and so on. As a result, the undefined value propagates back and forth between x and y, as reflected by the table in Figure 2a, demonstrating the missing-reset problem where the undefined values of x and y are used infinitely often. To fix this issue, the developer should add a reset for y in Cycle_0, in addition to the existing reset for x (line 3), ensuring that y_1 is defined and preventing the undefined values of x and y from being used infinitely often.

2.1.3 Missing-Reset vs. UBI. The missing-reset problem in Verilog bears a resemblance to the classical *use-before-initialization* (UBI) problem in software, which is prevalent in large-scale systems such as the Linux kernel and has driven decades of extensive research into static analyses for UBI detection [65, 114]. Despite their shared focus on undefined-value-related bugs, analyses for missing resets are fundamentally distinguished from those for UBIs due to their *hardware-specific nature*:

- *Hardware-Specific Problem Formulation:* The missing-reset problem concerns a register whose undefined value is *infinitely often* used in a *legal* Verilog execution, whereas the UBI problem concerns a variable whose undefined value is *once* used in *any possible* software execution. Consequently, analyses for missing resets must establish hardware-specific insights to handle the

"infinitely often" and "legal" constraints that do not arise in the UBI formulation. These insights will be elaborated in Section 2.2.

- *Hardware-Specific Information Requirements:* The missing-reset problem is inherently grounded in hardware semantics, including clocks, registers, and resets, beyond the scope of UBI. Therefore, detecting such bugs naturally requires fundamental hardware information that goes beyond the needs of traditional UBI analyses. Notably, obtaining all sorts of fundamental information from Verilog programs demands a series of non-trivial hardware analyses, a challenge further elaborated in Section 2.3.

These distinctions highlight the necessity of establishing hardware-specific insights to design hardware bug detection analyses capable of capturing hardware-specific problems and fundamental hardware analyses that can retrieve essential hardware semantic information from Verilog programs.

2.2 The Insight of Our Missing-Reset Analysis

The key insight of our missing-reset analysis is that only circuits involving *circular dependencies* among registers can have missing-reset problems. To see why, consider the opposite case: if a circuit has no circular dependencies among registers, then a topological order must exist for these registers. Following this order, defined values from constants or inputs can eventually propagate to all registers, ensuring that no register's undefined value is infinitely often used in any legal execution. The trivial case in which a register is used but never defined is handled by a separate analysis, `undefined-use` in Figure 3; thus, the missing-reset analysis only needs to focus on the non-trivial cases described above.

Based on this hardware-specific insight, our missing-reset analysis identifies registers that lack proper reset by reporting unreset registers involved in circular dependencies within the *hardware dependency graph*. This graph captures the data, control, resetting, and synchronization dependencies among hardware signals, including registers, clocks, resets, inputs, and constants. For instance, Figure 2b shows the hardware dependency graph of the missing-reset example from Figure 2a. We explain this hardware dependency graph as follows.

- The blue edge from x to y represents that x is data-dependent on y (introduced by `x <= y` in line 5 of Figure 2a), while the green edge from y to x indicates that y is control-dependent on x (introduced by `y <= x ? ...` in line 6 of Figure 2a). Together, these edges form a circular dependency (highlighted in the gray region) between x and y, underlying the situation where undefined values propagate back and forth between them, as elaborated in Section 2.1.2.
- The orange synchronization edges from x and y to the clock signal `clk` indicate that x and y are registers synchronized by the clock `clk`.
- The red resetting edges from x to the reset signal `rst` and the constant 1 capture x's reset logic—resetting x to 1 under the control of `rst`.

More concretely, our missing-reset analysis includes the following four steps: (1) *Non-Synthesizable Code Exclusion*: Verilog codebases typically include non-synthesizable portions written for simulation purposes, which do not appear in the final synthesized hardware and cannot contain real hardware bugs. Thus, we exclude such code from our analysis. (2) *Hardware Dependency Graph Construction*: Build the hardware dependency graph by resolving all sorts of dependencies in the Verilog program (e.g., as shown in Figure 2b). (3) *Circular Dependency Identification*: Locate cycles in the hardware dependency graph (the gray region in Figure 2b) to identify registers involved in circular dependencies (e.g., x and y in Figure 2b). (4) *Missing Reset Reporting*: Report registers from step (2) that lack proper reset. In Figure 2b, y is reported because it is unreset, whereas x is not since it is reset to 1 by `rst` (indicated by the red resetting edges in Figure 2b).

To illustrate real-world missing-reset bugs, Figure 2c presents such an example along with its hardware dependency graph, which retains only data and control dependencies for clarity. The code snippet originates from a FIFO (first-in, first-out) buffer in the Verilog-AXIS project [41], a popular open-source design with over 800 GitHub stars. Our analysis detected two missing-reset bugs in the registers `drop_frame` and `wr_ptr_cur` (lines 4–5 in Figure 2c) through the four steps described above. These bugs are confirmed by Ma et al. (2022a) to be *fatal*: the missing reset of `drop_frame`, a frame-dropping state register, leads to unintended packet drops, while the missing reset of `wr_ptr_cur`, a write-pointer register, causes writes to incorrect buffer addresses.

2.3 The Challenge in Building Sophisticated Hardware Static Analysis

Even with the core insights established, building a sophisticated hardware analysis remains challenging due to the lack of analyses focused on extracting fundamental hardware information—a topic that few prior works have explored.

For example, to detect missing-reset problems involving hardware *registers* driven by *clock* signals and reset by *reset* signals, as guided by the insights established in Section 2.2, the analysis requires essential hardware information about whether each variable represents a hardware register, a clock signal, a reset signal, or none of these. However, this is nontrivial because Verilog variables do not explicitly declare such information. This information is typically revealed after lengthy synthesis processes that map Verilog code to physical hardware components based on behavioral semantics. To bridge this gap we developed hardware understanding analyses for both the missing-reset analysis and other clients that require this hardware information. These analyses reason about such behavioral semantics, e.g., by inferring registers, clocks, and reset signals, allowing us to efficiently obtain the information needed for missing-reset analysis without resorting to heavyweight synthesis.

Moreover, these hardware understanding analyses require more fundamental hardware information, which is also crucial for bug detection clients and future potential clients, such as hardware security analyses. This necessitates diverse analyses, including but not limited to: (1) Verilog module hierarchy analysis for inter-module reasoning; (2) hardware data-flow and control-flow analyses for constructing various graph structures (e.g., the hardware dependency graph used by the missing-reset analysis); (3) analyses for capturing hardware's synchronization and concurrency characteristics, which are essential for reasoning about behavioral semantics in hardware understanding; (4) analyses of bit selects/part selects and bit-vector arithmetic, which are indispensable for achieving practically useful precision in hardware analysis.

Ultimately, building sophisticated hardware static analysis challenges us to uncover diverse fundamental analyses, understand their interdependencies, and enable their interoperability. To this end, we introduce a well-organized analysis suite, detailed in the next section.

3 Analysis Suite

We introduce the first analysis suite for Verilog exploiting sophisticated static analysis, in which a variety of fundamental analyses collaborate to support hardware bug detection, hardware understanding, and other potential analysis clients. We begin with an overview of the analysis suite. Next, since it is infeasible to discuss all analyses in detail within this paper, we present a focused case study to illustrate how our analyses collaborate to empower the key hardware bug detection client, `missing-reset`, building on its background and key insights introduced in Section 2.

3.1 Overview

Figure 3 presents all the major analyses in our suite by their short names and illustrates their dependencies with arrows. To avoid clutter, dependencies between analyses on the left and right

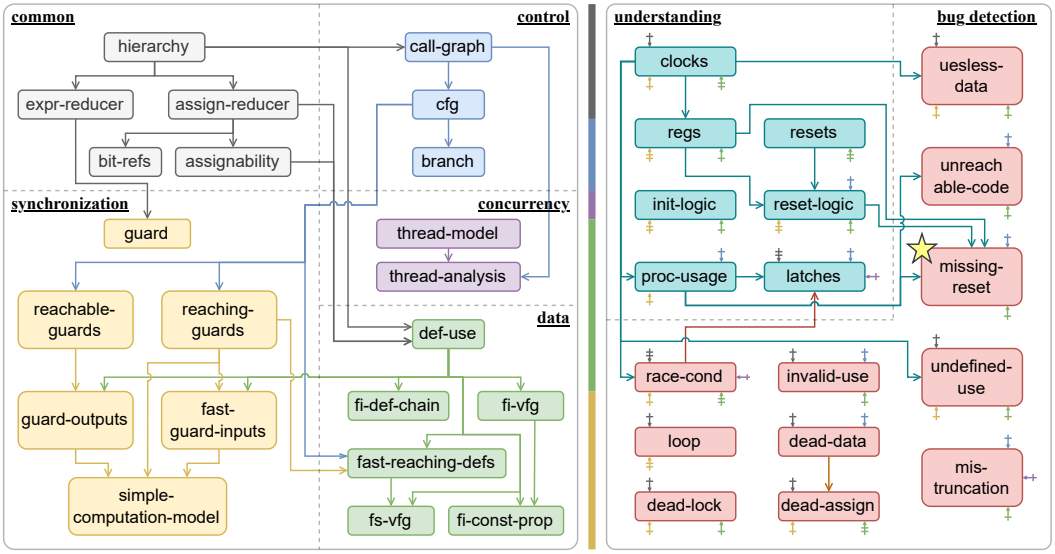


Fig. 3. All the major analyses in our suite and their dependencies. Each analysis is represented by its short name. The left section presents fundamental analyses, categorized into five groups, while the right section illustrates analysis clients for hardware bug detection and understanding. Arrows depict dependency relations between these analyses. To maintain clarity, we use short arrows ("↕" and "⇕") to succinctly represent the number of dependencies that an analysis client has on fundamental analyses. These arrows denote dependencies on one or two fundamental analyses within a specific category, with the category identified by the arrow's color (or position in black-white printing). For example, the starred analysis `missing-reset` has two short arrows "↕": one top-right and one bottom-right, signifying its dependency on *one* analysis from the *control* category and *one* from the *data* category. We use the central bar to visualize the relative usage of each category of fundamental analyses by color and length.

sides of the figure are indicated by special arrows, as explained in the figure caption, rather than drawing every dependency explicitly.

To facilitate a clear understanding of our analyses, Figure 3 divides our analysis into two parts based on their design purposes: *fundamental analyses* (left) and *analysis clients* (right). Fundamental analyses extract general and essential hardware information to support a variety of clients. They are further categorized into five groups according to their focus: *common* language features, *data* and *control* flows, and *concurrency* and *synchronization* characteristics. Analysis clients are primarily designed for application-specific tasks, currently spanning two domains: hardware *bug detection* and hardware *understanding*. Analyses for bug detection are motivated by real-world bugs from prior studies [29, 30, 78] and insights from industrial hardware practitioners. Analyses for hardware understanding not only provide useful information to support bug detection analyses, but also assist developers in understanding the hardware synthesized from Verilog code and identifying potential post-synthesis issues early, without requiring the time-consuming synthesis process.

To maximize modularity and reusability, each analysis in our suite is designed with a single, well-defined purpose and delegates other responsibilities to upstream analyses as dependencies. This approach naturally leads to dependency relationships, as depicted in Figure 3.

In addition to providing an overview of our analysis suite, Figure 3 also serves as a conceptual takeaway for future Verilog analyzer developers. It depicts clients reflecting real-world hardware

analysis needs, categorizes fundamental analyses distilled from client-driven practice, and illustrates the dependency structure that naturally emerges from a fine-grained, single-purpose design principle. We expect this experience can help future researchers understand what analyses are needed by hardware developers and how such analyses can be modularly organized.

For a deeper understanding of these analyses and their dependencies, the next section (Section 3.2) details the dependencies of the `missing-reset` analysis as a representative example.

3.2 Supporting Sophisticated Analysis: A Case Study

Recall from Section 2 that the `missing-reset` analysis statically detects registers that lack proper reset logic, which can cause them to repeatedly hold undefined values after a circuit reset. This analysis involves four main steps: (1) excluding non-synthesizable code from the Verilog program, (2) constructing a hardware dependency graph, (3) enumerating cycles within this graph, and (4) identifying registers involved in these cycles and reporting those that lack proper reset logic.

Although the four steps are succinctly described, their implementation relies on the collaboration of nearly twenty analyses in our suite, totaling 9,700 lines of code. Figure 4 illustrates all analyses supporting `missing-reset`, arranged in topological order from fundamental analyses (top) to analysis clients (bottom). Below, we use Figure 4 to help readers understand why the `missing-reset` analysis requires its dependencies and how these dependencies contribute to its overall effectiveness, thereby providing insight into the organization of our analysis suite for supporting sophisticated analyses. Due to space constraints, we do not discuss algorithmic details here; interested readers can find them in our open-source project.

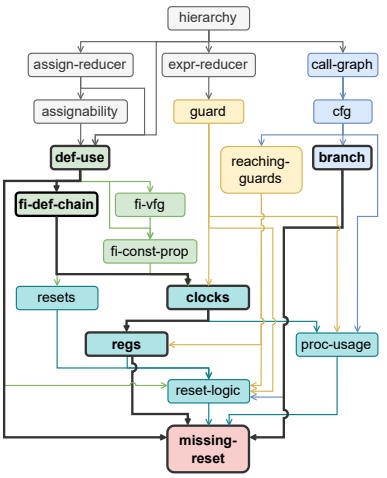


Fig. 4. All analyses that support the `missing-reset` analysis.

Interdependencies. To clarify why the `missing-reset` analysis requires its dependencies in Figure 4 while maintaining simplicity and readability, we backtrack the bold dependency arrows in the figure, emphasizing both the functionality of these analyses and the necessity of their interdependencies. To begin, (1) Since missing-reset bugs are confined to registers, `missing-reset` directly depends on the `regs` analysis to infer physical registers, and utilizes `def-use` and `branch` analyses to build a graph containing data and control dependencies for cycle detection, as described in Section 2.2; (2) Since registers are synchronized by clock signals, `regs` tightly depends on `clocks` to infer physical clocks; (3) Since clocks are propagated as data across modules in Verilog, the `clocks` analysis directly depends on `fi-def-chain`, which provides an inter-module graph representing data flow to support such reasoning; (4) Lastly, `fi-def-chain` leverages the `def-use` analysis to obtain define-use relations between variables, including inter-module relations, to build the inter-module graph.

Analysis and Its Downstream Effectiveness. These interdependencies underscore that the effectiveness of each analysis can be critical for downstream analyses. For example, reduced precision in the `clocks` analysis (i.e., misclassifying non-clock variables as clocks) directly affects the precision of `regs`, since registers are synchronized by clock signals; insufficient precision in `regs` can lead to excessive false reports in `missing-reset`, undermining its practical value. Likewise, insufficient recall in `clocks` and `regs` (i.e., reduced soundness that omits some true behaviors) may cause `missing-reset` to overlook genuine missing-reset problems. To address these challenges, key supporting analyses are designed to achieve high effectiveness. Due to space constraints, we present

our design considerations for two representative analyses: `regs` and `fi-const-prop`. The former shows our approach to designing a hardware analysis, which is not found in software, to achieve high recall and precision. The latter—a flow-insensitive constant propagation analysis—shows that some analyses in our suite, though inspired by software concepts, require Verilog-specific adaptations to achieve effective results on hardware.

`regs` infers which Verilog variables are synthesized as hardware registers. To achieve high precision, it cannot simply report every variable declared with Verilog's `reg` keyword, because many such declarations denote intermediate variables rather than physical registers (the keyword is misleading), yielding numerous false positives. Instead, `regs` classifies registers by their semantics. Register updates are synchronized to clock edges, which are expressed in Verilog as non-blocking assignments guarded by event controls that reference clock signals; accordingly, the core step of `regs` is to locate such non-blocking assignments and report their left-hand-side (LHS) variables as registers. Recall that Figure 1 illustrates an event control (e.g., `@(posedge clock)`) guarding a non-blocking assignment (e.g., `acc <= acc + in`), where the LHS variable `acc` is the register. To achieve high recall, the key is to soundly extract the guarding relationship between non-blocking assignments and event controls. Simply matching the pattern shown in Figure 1 provides unsound results because event controls and non-blocking assignments can appear in arbitrary orders within an `always` block, yielding nontrivial guarding relationships. For example, the body of an `always` block could be `@(posedge clock); x <= 1; @(negedge clock); y <= 0;`, where `y <= 0` is guarded by the later `@(negedge clock)`, not the earlier `@(posedge clock)`. Therefore, we compute the guard relation via a sound data-flow analysis over the control-flow graph using the fundamental `reaching-guards` analysis, on which `regs` depends, as shown in Figure 4.

`fi-const-prop` is widely used in our suite, and its precision is crucial for the effectiveness of downstream analyses. Unlike typical software programs, achieving high precision in Verilog requires bit-level constant reasoning, as Verilog programs frequently operate on individual bits or bit-vectors corresponding to physical wires. Accordingly, `fi-const-prop` accurately models Verilog's four-valued logic (0/1/X/Z) and bit-vector arithmetic to enable precise bit-level constant propagation. We discuss a bug detected thanks to this design in Section 5.1. Note that we do not adopt the flow-sensitive variant of constant propagation commonly used in software analysis, as we found it provides only marginal precision improvement while notably reducing analysis speed. This is mainly because most Verilog variables are defined in a single location, reflecting the fact that hardware signals typically have only one definition, which makes flow-sensitivity less beneficial. This further underscores the necessity of Verilog-specific adaptation.

Discussion on Soundness. As mentioned above, unsoundness in our hardware understanding analyses (e.g., `clocks` and `regs`) can render `missing-reset` unsound. Although we strive to maintain their high recall in practice, it is important to note that such unsoundness is inherent. This arises because these hardware understanding analyses attempt to predict the circuit structure produced by synthesis tools, while the Verilog standard [1] specifies only circuit behaviors, not their structure. Consequently, different synthesis tools may generate structurally distinct circuits from the same Verilog code, with no formal standard serving as ground truth for sound prediction. Nevertheless, in practice, synthesis tools tend to follow common mapping conventions, allowing our analyses to effectively predict useful synthesis outcomes despite this limitation.

Excluding this source of unsoundness, the *reachable closure* of `missing-reset`'s results—the set of registers reachable from the reported registers in the hardware dependency graph—can *soundly* capture all possible missing-reset problems. Registers outside this closure cannot exhibit missing-reset issues, as they do not participate in circular dependencies within the circuit. Based

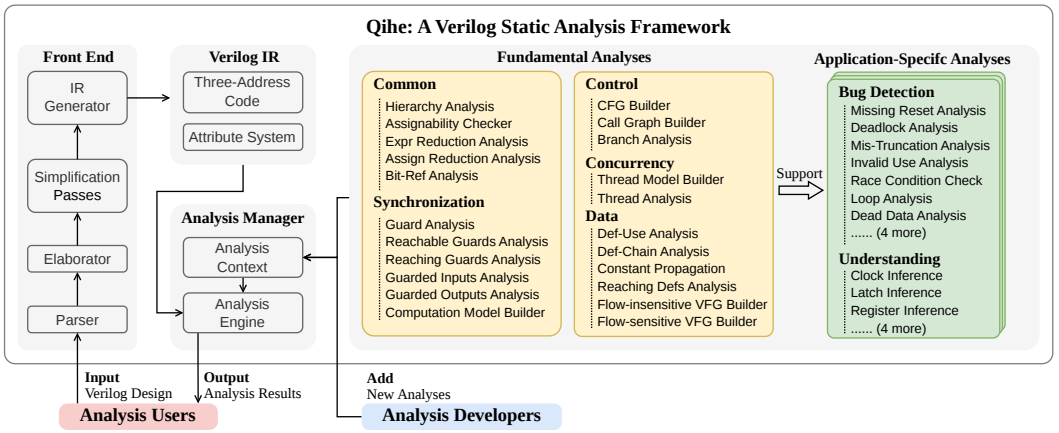


Fig. 5. Workflow of our Verilog static analysis framework. It supports both *analysis users* in executing analyses on Verilog designs and *developers* in creating new analyses, enabled by the collaboration of its components.

on our insight in Section 2.2, their values are ultimately computed from constants, inputs, or reset registers, all of which obtain defined values after the reset cycle in any legal execution.

The collaborative nature of our analyses goes beyond those mentioned above and can be more intricate when developing hardware static analyses that are more sophisticated. We hope the example of `missing-reset` can serve as a reference for developers to create their own applications by leveraging existing analyses in our analysis suite.

4 Framework Implementation

Due to space constraints, we refer readers to an extended version of this paper [24] for a detailed description of our analysis infrastructure, including a Verilog front end, an analysis IR, and an analysis manager. These components, together with our analysis suite, constitute *Qihe*, a Verilog static analysis framework implemented from scratch in over 100K lines of Java. This section offers an overview of how these components interact to support the typical use cases of Qihe, highlighting their analysis-specific features.

Qihe is designed to address two key use cases: enabling *analysis users* to execute available analyses on their hardware designs, and empowering *analysis developers* to create diverse new analyses using the framework. Figure 5 illustrates the underlying workflow for each use case, supported by the four core components: the front end, IR, analysis suite, and analysis manager.

For analysis users, the workflow begins by inputting Verilog designs into the *front end*, which converts them into the *Verilog IR*. The *analysis manager* then executes the requested analyses, as configured by users, on the IR and delivers the results to users. Specifically, the manager detects available analyses and configurations, registers them into an analysis context, and invokes an analysis engine to apply them to the IR. Analyses are executed in topological order based on their dependency relations to ensure correctness. Given the complexity of manually managing a suite of 40 interdependent analyses, the analysis manager greatly streamlines the execution process for users.

For analysis developers, the *analysis manager* provides an intuitive interface for creating new analyses (whether fundamental or application-specific) based on existing ones. Once new analyses are implemented, the manager automatically detects and registers the new analysis into the analysis context, making it available for users to execute. Moreover, the Verilog front end and IR work together to address specific analysis requirements, such as supporting incomplete hardware

Table 1. Hardware bugs identified by our bug detection analyses in real-world projects. All listed bugs are undetectable by existing static analyses for Verilog bug detection [26, 50, 88, 99]. Newly discovered bugs are shown in **bold**; they were previously unknown and have been confirmed by developers. "#Reports(#Confirmed)" indicates the total number of reports produced for each project, and among these, how many are confirmed by developers (i.e., correspond to the listed bugs).

Category	ID	Description	Project	#Reports (#Confirmed)	Freshness
Missing Reset	B01 B02	Register `drop_frame` lacks proper reset logic. Register `wr_ptr_cur` lacks proper reset logic.	Verilog-AXIS [41]	20 (2)	Known from [78].
	B03	Register `correct` lacks proper reset logic.	PULPissimo [93]	1 (1)	Known from [81].
	B04 **B05**	Register `TIMER` lacks proper reset logic. Register `TIMEUS` lacks proper reset logic.	DarkRISCV [91]	3 (2)	Newly discovered.
Unreachable States	B06	Used state `dma_ctrl_reg == CTRL_ABORT` is unreachable.	OpenPiton [11]	1 (1)	Known from [30].
	B07 **B08** **B09**	Used state `count == 2` is unreachable. Used state `count == 8` is unreachable. Used state `count == 32` is unreachable.	32-Verilog-Projects [83]	8 (3)	Newly discovered.
Deadlock	B10	`o_sclk` gets stuck from a deadlock.	SDSPI [43]	4 (1)	Known from [78].
	B11	`rd_addr` gets stuck from a deadlock.	HDL Lib [61]	2 (1)	
Undriven Signals	**B12**	Use of values from the undriven signal `clock_div`.	Basic Verilog [86]	6 (1)	Newly discovered.
	B13	Use of values from the undriven signal `r_cv`.	ZipCPU [44]	1 (1)	
	B14	Use of values from the undriven signal `v`.	32-Verilog-Projects [83]	10 (1)	
Unloaded Signals	B15	Module `aes_1cc` has an unloaded signal `rst`	PULPissimo [93]	16 (1)	Known from [81].
Submodule Misuse	**B16**	Misuse of submodule `invert` due to wrong port order.	32-Verilog-Projects [83]	10 (1)	Newly discovered.
Mis-Truncation	B17	Mis-truncation of `rx_mmio_channel` causes data loss.	Sha512 [19]	1 (1)	Known from [78].
Invalid Use	B18	Use of invalid value from signal `axi_adrv9001_rx_channel`.	HDL Lib [61]	2 (1)	

programs and retaining additional source information, while maintaining a simple IR for ease of analysis. We encourage developers to leverage Qihe to explore and build more useful analyses.

5 Evaluation

To examine the utility of our hardware static analyses, we first assess whether our bug detection clients can effectively identify bugs in real-world hardware designs (Section 5.1). Behind our bug detection clients lies a series of fundamental analyses that combine to build increasingly sophisticated analyses, which ultimately enable useful analysis clients for real-world bug detection and other potential applications. To our knowledge, no prior work has explored these fundamental analyses or investigated their composition. Section 5.2 presents case studies showing how such a collaborative process enables the detection of critical missing-reset and dead-lock bugs by uncovering the internal statistics integral to this process.

5.1 Hardware Bug Detection

Detecting hardware bugs remains a formidable challenge, even in industrial settings. According to Siemens' comprehensive global industrial study in 2024 [42], merely 14% of IC/ASIC projects achieved first silicon success, and only 13% of FPGA projects reported zero bug escapes into production. Unlike software, where bugs can be patched remotely over the internet at a low cost, late-stage hardware bugs in production can result in exponentially increasing remediation expenses [7, 10].

Static analysis offers a lightweight approach for early-stage bug detection at design time, making it especially cost-effective for hardware. To showcase the potential of sophisticated static analysis for hardware bug detection, we developed a set of Verilog analyses. The insights behind them came from three sources: studying known bugs in the most recent hardware bug survey [78, 79], the renowned annual hardware bug detection hackathon [29–31], and through direct communication with industrial hardware practitioners [2, 3].

To evaluate the effectiveness of these bug-detection analyses, we applied them to two categories of Verilog projects: (1) Projects with previously studied bugs, from which we derived analysis insights. For these projects, we applied the corresponding analyses to evaluate whether they could rediscover the bugs that inspired them. (2) Projects without previously studied bugs. These projects are high-star GitHub repositories selected primarily based on the convenience of compilation (projects that cannot be compiled using open-source tools were excluded). For these projects, we applied all bug-detection analyses, since the bug types were unknown in advance, and report projects with confirmed newly discovered bugs. For other analyzed projects without confirmed new bugs, we include them in our artifact [22] for future reference.

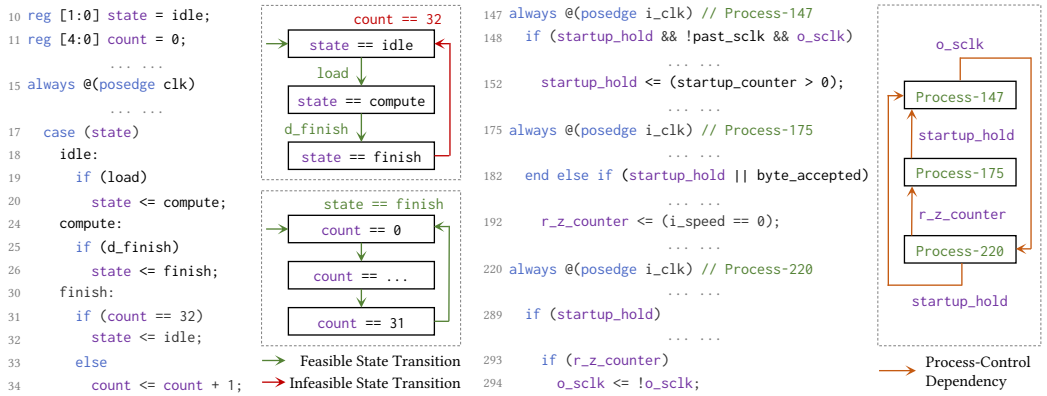
As shown in Table 1, the preliminary experimental results are very promising. Our analyses successfully identified 18 bugs across 8 categories—such as missing resets, unreachable states, and deadlocks—in a diverse set of real-world hardware projects, such as system-on-chips (SoCs), communication protocols, and encryption modules. As confirmed by our attempts, none of these bugs could be detected by existing Verilog linters [26, 50, 88, 99], the dominant static analysis tools for hardware bug detection. Their syntax-based approach and lack of fundamental semantic analysis prevent them from identifying bugs that require deeper hardware semantic reasoning. Notably, 9 of the 18 bugs were newly uncovered by our analyses from popular open-source projects [44, 83, 86, 91] (averaging over 1.5K GitHub stars). All of these new bugs have since been confirmed by the respective developers. These results not only highlight the effectiveness of our analyses but also underscore the promise of sophisticated static analysis for practical hardware bug detection.

In addition, we assess the false-positive burden in bug detection. For each project in Table 1, we run the client to report the bug(s) listed for that project and record the total number of bug reports (#Reports). We then count how many of these reports were confirmed by developers (#Confirmed), i.e., match the listed bugs; the rest are false positives. As shown in Table 1, there are 85 reports in total, of which 18 (21%) are confirmed and the remaining 67 are false positives. These results suggest that the false positive burden is manageable in the bug-finding setting.

To illustrate how our analyses identify additional categories of real-world hardware bugs beyond the missing-reset bugs already discussed in Section 2.2, we present two additional case studies—an unreachable-state bug (B09) and a hardware deadlock bug (B10)—as supplementary representative examples of sophisticated Verilog static analyses for practical hardware bug detection.

5.1.1 An Unreachable-State Bug Case Study. Finite-state machine (FSM) design is a common task for Verilog engineers [82]. An unreachable state is one that the FSM cannot reach from any initial state. Such states often indicate functional issues, such as unintended infeasible state transitions [92].

Figure 6a presents a real-world unreachable-state bug from a CRC-32 serial counter in an open-source Verilog project [83], where `idle`, `compute`, and `finish` are constant enumeration values for `state`. The diagram on the right visualizes two FSMs defined by the Verilog code on the left: nodes represent states (corresponding to the values of `state` and `count`, respectively), edges denote transitions, and expressions on edges indicate transition conditions. Since all transitions in the `count` FSM (the lower one) share the same condition, we show it only once to avoid redundancy. The bug arises as follows. Lines 30–34 of Figure 6a describe the interaction between the `state` FSM (the upper one) and the `count` FSM: when `state` is `finish`, `count` starts incrementing, and when `count == 32`, `state` should return to `idle`. However, the developer failed to notice that `count` is declared as a 5-bit vector (line 11) that overflows after reaching 31 and wraps back to 0, an easy-to-overlook case in Verilog programming. Consequently, the state where `count == 32` is unreachable in the `count` FSM, breaking the intended interaction between the two FSMs and causing the `state` FSM to exhibit an infeasible transition from `state == finish` to `state ==`

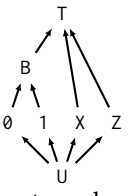


(a) A real-world unreachable-state bug and its finite-state machine (FSM) diagram.

(b) A real-world hardware deadlock bug and its process dependency graph.

Fig. 6. Real-world bug case studies: B09 (a) and B10 (b) from Table 1. Line numbers correspond to those in the original source projects.

`idle` (highlighted in red). This flaw is severe, as it ultimately traps the system in the `finish` state and prevents it from ever resuming normal operation.



To detect such unreachable states, our analysis performs *Verilog-specific bit-level* whole-program constant propagation to determine all possible values of every variable with bit-level precision. As discussed in Section 3.2, this precision is critical in Verilog, where programs frequently manipulate individual bits, and loss of precision can cause the analysis to miss bugs that would otherwise be detected, such as the one in this case study. For each variable, our constant propagation overapproximates its concrete bit-vector value using a specially designed abstract bit vector, where each bit is drawn from the lattice shown on the right: `U` represents an undefined bit, `0`/`1`/`X`/`Z` represent the fixed Verilog bit values 0/1/X/Z (where X denotes an undetermined bit and Z denotes a high-impedance bit), `B` (short for *both*) represents a bit that can be either `0` or `1`, and `T` represents the top lattice element encompassing all possible bit values.

In this case study, we focus primarily on the `0`/`1`/`B` portion; for example, for a three-bit signal x, x = `B01` implies that x can take only two possible values: `001` or `101`. In Figure 6a, the transition condition `count == 32` (line 31) is interpreted by our analysis as comparing `0BBBBB`—with `count` zero-extended to 6 bits (an implicit type conversion per the Verilog specification [1])—against `100000`, representing the fixed value 32. This equality is impossible because the most significant bit never matches ($0 \neq 1$); therefore, the state where `count == 32`, used in lines 31–32 of Figure 6a, is unreachable. In contrast, directly using a conventional non-bit-level constant propagation would treat the left-hand side as a non-constant value (denoted `NAC`), so evaluating `NAC == 32` would yield a "possible" (reachable) rather than "impossible" (unreachable) result, causing this unreachable-state bug to be missed. This example underscores that directly applying software-like analyses to hardware is often ineffective, and that Verilog-specific customizations, such as maintaining bit-level precision, are essential. Additional designs and algorithms for Verilog-specific static analysis are available in our open-source framework and documentation, as introduced in Section 4.

5.1.2 A Hardware Deadlock Bug Case Study. According to the Verilog language specification [1], a running Verilog program consists of multiple concurrent processes, each typically corresponding to a run-time instance of an `always` block. A hardware deadlock occurs when circular dependencies

among concurrent Verilog processes cause the program to stall infinitely [78], manifesting physically as signals that always fail to update as expected.

Figure 6b presents a real-world deadlock bug from the `llsdspi` module of the SDSPI project [43], which implements an SD card controller. This controller is expected to generate a driver clock signal named `o_sclk`. However, `o_sclk` fails to update due to a deadlock caused by circular dependencies among three processes: `Process-147`, `Process-175`, and `Process-220`, where `Process-`i denotes the run-time Verilog process corresponding to the `always` block starting at line i. Specifically, the update of `o_sclk` (line 294) in `Process-220` relies on the update of `r_z_counter` (line 192) in `Process-175`, which in turn relies on the update of `startup_hold` (line 152) in `Process-147`, which again relies on the update of `o_sclk` (line 294) in `Process-220`. As a result, all three processes remain infinitely stalled, preventing the SD card controller from generating the active driver clock `o_sclk`. Without this clock, the SD card becomes completely non-functional, unable to perform any data access operations.

To detect such hardware deadlock bugs, our analysis first constructs a *process dependency graph* that captures whether the value updates in one Verilog process depend on those in another, as exemplified in Figure 6b. In this graph, an edge from `Process-`i to `Process-`j ($i \neq j$) labeled with x indicates that `Process-`i is dependent on `Process-`j through a shared signal x; that is, x is used in a branch condition within `Process-`i and defined by an assignment in `Process-`j. To identify circular dependencies among concurrent Verilog processes, our analysis detects all cycles in this graph. For the example in Figure 6b, the analysis reports a circular process dependency among `Process-147`, `Process-175`, and `Process-220`, pinpointing the deadlock that stalls the system.

5.1.3 Preliminary Results on Hardware Security Analysis. In addition to the hardware bug detection focus of this section, hardware security represents another promising application domain for sophisticated Verilog static analysis. Security concerns have become increasingly critical with the widespread use of integrated circuits (ICs) in systems vulnerable to threats such as sensitive information leakage [53] and malicious hardware Trojans [54]. As a preliminary exploration, we also developed two security analyses on top of our analysis suite introduced in Section 3: taint analysis (1800 LoC) and X-propagation [59] analysis (300 LoC). Although we cannot elaborate on them due to space constraints, our preliminary results (fully reproducible from our artifact [22]) are also very encouraging: the taint analysis successfully detected 15 out of 19 Verilog information-leak vulnerabilities in the renowned Trust-Hub benchmark suite [90, 96], and the X-propagation analysis identified a hardware Trojan known from [59]. We plan to continue exploring how static analysis could assist in addressing hardware security challenges in future work.

5.2 Snowballing Process in Analysis Suite: A Case Study

The growth of analysis capabilities in our analysis suite resembles a snowballing process, where existing analyses collaboratively contribute to increasingly sophisticated analyses. This process ultimately allows us to detect bugs that were previously unattainable with existing static analyses [26, 50, 88, 99] for Verilog bug detection, as summarized in Table 1. Section 3.2 has discussed an example of this snowballing process, where key analyses interdepend and collaborate to support the `missing-reset` analysis. In this section, we take a deeper look at this process by examining the experimental results associated with `missing-reset`. Specifically, we apply the `missing-reset` analysis and its dependencies to real-world programs, emphasizing three key aspects from the experimental results: (1) each analysis contributes to downstream effectiveness (i.e., analysis soundness and precision), (2) our analysis suite significantly reduces development costs, and (3) the efficiency of fundamental analyses dominates overall system efficiency. Finally, to assess whether these findings extend beyond a single client analysis, we conducted a complementary case study

Table 2. Results of `missing-reset` and its representative dependencies, ordered topologically by their dependencies, across 13 Verilog programs. Columns represent programs, rows represent analyses (except the `Meta` row lists lines of code and IR of these programs), and cells show summarized statistics. The last column aggregates statistics from previous columns, using averages for the hardware-understanding analyses while using totals for the remaining analyses. For `cfg` and `fi-def-chain` that provide graphical program abstraction, collecting complete ground truth is infeasible and thus omitted. For `clocks`, `regs`, and `resets`, results are measured by precision and recall against the ground truth (#Truth). For `missing-reset`, since no ground truth exists, we present how many reported bugs are confirmed by developers in the format "#Reported (#Confirmed)". If none of the reported bugs are confirmed, we omit "(#Confirmed)."

		XS	ziu	biv	FPC	ha3	riv	ax1	sev	pi2	dav	adf	st1	sha	Summary
Meta	LoC	1821858	22290	13601	7751	7662	6832	6804	3461	3049	2753	2724	2553	2171	1903509
	LoIR	8261716	26554	25435	41065	16053	12309	28151	5825	5747	3592	7128	9429	2281	8445285
`cfg`	#Node	5020594	15308	14762	18641	8130	7035	17159	3362	3170	2013	4215	5844	1846	5122079
	#Edge	4355660	14646	13773	24735	8583	6635	15812	2892	3398	1799	4161	5100	1932	4459126
`fi-def-chain`	#Node	1549502	6369	5784	11280	4272	3024	7753	1211	1932	1024	1476	2124	1656	1597407
	#Edge	3176863	12009	10387	15147	7486	4653	13575	1694	9414	1628	2106	3479	87631	3346072
`clocks`	#Truth	2651	29	36	13	10	14	39	13	3	9	23	16	1	–
	Precision	100%	100%	100%	87%	100%	100%	100%	100%	100%	100%	100%	100%	100%	99%
	Recall	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
`regs`	#Truth	63678	367	197	152	97	153	604	50	156	52	66	263	32	–
	Precision	100%	99%	86%	100%	100%	100%	100%	100%	99%	100%	100%	100%	100%	99%
	Recall	100%	100%	97%	100%	100%	99%	99%	100%	97%	100%	94%	100%	100%	99%
`resets`	#Truth	2218	37	36	14	8	14	38	7	4	10	19	6	1	–
	Precision	98%	97%	100%	92%	80%	100%	100%	100%	100%	50%	100%	100%	100%	94%
	Recall	88%	81%	100%	79%	100%	100%	100%	86%	75%	100%	100%	83%	100%	92%
`missing-reset`	#Reported	50	9	0	0	0	0	20 (**2**)	10	5	3 (**2**)	1 (**1**)	12	0	110 (**5**)

of the `dead-lock` analysis under the same evaluation methodology. Due to space constraints, we summarize the results in Section 5.1.2 and provide the detailed tables, figures, and corresponding explanations in the supplementary material [23].

Program Set. To provide practical statistics associated with the `missing-reset` analysis, we reused hardware programs that had already been compiled and analyzed for bug detection in Section 5.1, subject to the following additional criteria. The selected programs span diverse domains, such as CPU design, encryption, AXI protocols, and computer vision, and vary substantially in size, range from thousands to millions of lines of Verilog. We also include programs with known missing-reset bugs to enable characterization of the false-positive behavior of the `missing-reset` analysis on buggy codebases. Programs that cannot be synthesized by Yosys [112] are excluded, since we rely on Yosys to obtain the ground truth for computing precision and recall for the `clocks`, `regs`, and `resets` analyses. The top of Table 2 lists these programs by shorthand names for brevity, as well as their metadata, including lines of code (LoC) and lines of IR (LoIR) after compilation. All these programs are available in our artifact [22].

Representative Analyses. To showcase how analyses in Figure 4 enhance downstream effectiveness in the snowballing process, Table 2 presents representative analyses and their results, organized in topological order. The absence of certain analyses from this table, such as `hierarchy` (for module hierarchy resolution in Verilog) and `reaching-guards` (for synchronization information extraction), does not diminish their importance. They are excluded because their results are either unsuitable for concise presentation in the table or require substantial human effort to collect their ground truth. Nevertheless, their effectiveness is still reflected in the results of the analyses listed in Table 2, as these analyses depend on them.

Result Metrics. Precision and recall are the primary metrics for evaluating the effectiveness of static analyses, and their calculation depends on establishing ground truth: precision measures the proportion of identified results that are true, while recall indicates the proportion of ground truth results that are successfully identified. However, the difficulty of establishing ground truth varies across analyses, which leads to different reported metrics in Table 2: (1) For `cfg` and `fi-def-chain`, which provide graphical program abstractions, collecting complete ground truth is infeasible; therefore, Table 2 reports the number of graph nodes and edges solely to illustrate the scale of these abstractions. (2) For hardware understanding tasks (`clocks`, `regs`, and `resets`) that predict synthesis tool outcomes, different tools may occasionally produce varying results; however, they generally agree on the majority of cases, as discussed in Section 3.2. Therefore, we use the results of Yosys [112]—the most popular open-source synthesizer—as ground truth; e.g., for `regs` we use the set of registers inferred by Yosys. Note that relying on Yosys for hardware understanding tasks is considerably slower. Specifically, for the XS program with 1.8M lines in Table 2, synthesizing to a netlist (a logical circuit) with Yosys takes 65 minutes, whereas all of our hardware understanding analyses complete in about 3 seconds. Even when limiting Yosys to partially synthesize XS to produce only ground truth for the three tasks studied in this section, Yosys takes 330 seconds, while our three corresponding analyses require only 0.5 seconds. Moreover, Yosys typically does not provide APIs to access results commonly required by static analysis, as it is not designed for this purpose. For example, it does not provide APIs to dump the commonly needed clock tree in a circuit—a feature supported by our `clocks` analysis. This is limitation also complicates the process of building ground truth. To obtain the true clock tree, we first modified Yosys to dump the clock tree leaves, i.e., clock signals directly connected to registers. We then extracted the variables connected to those leaves to approximate the clock tree and manually verified which of them were valid clock signals. (3) For `missing-reset`, no ground truth exists for open-source projects; however, we can assess its usefulness by reporting how many detected bugs are confirmed by developers. Accordingly, Table 2 lists the number of reported bugs, along with the count of confirmed cases, in the format "#Reported (#Confirmed)". If no bugs are confirmed, we do not count them as true bugs and omit "(#Confirmed)", as our goal is to highlight how many reported bugs are actually of concern to developers. Based on our manual inspection, we believe that the unconfirmed bug reports are mostly false positives.

5.2.1 Analysis Results. `cfg` (control-flow graphs, CFGs) and `fi-def-chain` (flow-insensitive def-use chains) appear first in the table, highlighting their foundational role in enabling subsequent key analyses. For example, the `clocks` analysis, conducted later, utilizes edges from the `fi-def-chain` to propagate clock signal information. Initially, we explored a flow-sensitive implementation (`fs-def-chain`) due to concerns that false positive edges in the `fi-def-chain` might generate infeasible propagation paths, thereby reducing the precision of clock inference in the `clocks` analysis. However, we discovered that the flow-insensitive version provides sufficient precision and significant speed benefits in practice, making it our preferred approach. This case also implies that our analysis suite actually allows developers to explore alternatives to make preferred trade-offs in practice by offering a range of fundamental analyses.

The next three analyses in the table (`clocks`, `resets`, and `regs`) infer the physical usages of variables. Their losses in precision and recall can significantly compromise the effectiveness of their dependent analyses and the practical utility in real-world scenarios, as discussed in Section 3.2. Therefore, we dedicated careful effort for good precision and recall, and the results align with our expectations, as shown in Table 2 and explained below.

For `clocks`, our semantic-reasoning approach enables excellent precision and recall across all evaluated programs. The key insight is that hardware clocks form a tree structure whose boundaries

with the rest of the circuit can largely be inferred from code semantics. Once these boundaries are identified, the entire tree can be extracted, starting from any tree node. The `regs` analysis also achieves high precision and recall, leveraging the effectiveness of the `clocks` analysis it depends on and its reasoning about hardware semantics, as described in Section 3.2. Its recall is slightly reduced in some programs because it misses registers updated via blocking assignments, rather than the standard non-blocking assignments. While Yosys accommodates such cases, blocking assignments cannot express the register's "update value in next clock cycle" semantics [25], so this loss of recall is not concerning. The `resets` analysis attains high precision and recall using a similar insight to `clocks`, but does not reach perfect scores because reset signals have more complex usages in real-world designs compared to clock signals. Although its results are sufficiently high for our practical use, further improvements can be made by contextualizing the algorithm to the usage characteristics of reset signals in future work.

Finally, `missing-reset` effectively identifies missing-reset bugs in real-world programs that could lead to unintended hardware behavior, leveraging the effectiveness of the preceding analyses. As shown in Table 2, bold values indicate five bugs confirmed by developers. It is worth noting that the number of reported missing-reset bugs remains manageable. For example, even in the million-line XS program, only 50 bug reports are generated, which is acceptable for manual inspection.

Controlling the number of false bug reports is critical in hardware bug detection, as excessive reports can overwhelm developers and obscure true issues. We address this in two ways: (1) By inspecting false positives in real-world programs, we introduce refinements tailored to each client when necessary, reducing spurious bug reports without significantly sacrificing recall. For example, in the `missing-reset` analysis, we skip excessively large cycles when enumerating cycles in the hardware dependency graph. Such cycles are typically caused by false positive edges, which are inevitably introduced when building a sound dependency graph, and rarely correspond to genuine bugs. (2) We employ a general strategy for bug detection analyses: Verilog programs often use meta-programming to describe repetitive circuit patterns. After elaboration, which expands meta-programming constructs, the same issue—especially a false positive—may be reported multiple times. To address this, we implement reusable utility components to collapse near-duplicate reports and emit a single representative. While this approach may merge multiple instances of a true bug, fixing the representative root cause eliminates all related occurrences. Note that the underlying fundamental analyses remain general-purpose and are reused unchanged across the clients. As a result of these efforts, the numbers of bugs (i.e., the bugs in Table 1) reported by our detection clients on real-world programs are *all* manageable, similarly to the situation with `missing-reset`, making it easier for us to validate the true bugs among them.

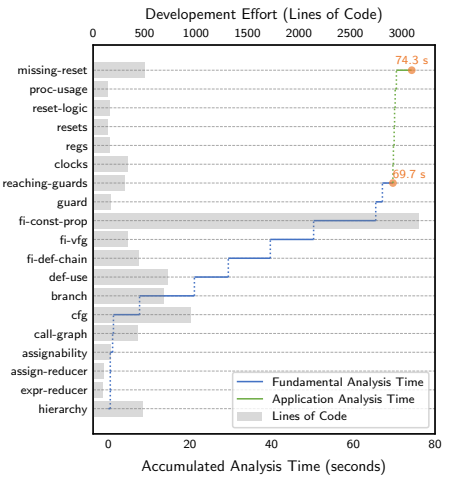


Fig. 7. Missing-reset analysis on a large-scale RISC-V SoC (XS from Table 2).

5.2.2 Development Effort. In the snowballing process, we observe that developing a new client becomes significantly easier by leveraging our analysis suite. The left axis of Figure 7 shows all analyses contributing to `missing-reset`, arranged from bottom to top in topological order based on their dependencies. The gray bars represent the lines of code (LoC) for implementing each analysis. As shown, implementing `missing-reset` based on our analyses requires significantly less development effort (540 LoC) compared to

implementing all these analyses from scratch (9,700 LoC). On average, each bug detection client built on our suite requires about 270 LoC, versus 5,700 LoC from scratch. This substantial reduction in development effort underscores that our analysis suite is capable of facilitating the development of new analysis clients.

5.2.3 Analysis Time. Fundamental analyses, which address essential analysis needs with a focus on hardware characteristics, also shoulder the majority of the analysis workload required to support downstream analyses. Figure 7 shows the accumulated run-time cost of each analysis when running `missing-reset` on XS, a large-scale RISC-V SoC from Table 2 with 1.8M lines of code, measured on a machine with an Intel i9-13905H CPU and using 16 GB of memory. In this case, fundamental analyses (from `hierarchy` to `reaching-guards`) contribute 94% of the total time (69.7 s of 74.3 s). Similarly, across the other twelve programs from Table 2, they average 85% of the total time.

This highlights that fundamental analyses dominate our suite's run time, so optimizing them can yield significant overall speedups. We accelerate these analyses by leveraging Verilog-specific properties—such as the acyclicity of combinational logic, the branchless nature of continuous assignments, and the synchronicity of guard statements—alongside conventional software optimizations. Before optimization, some analyses on XS took up to an hour using 128 GB of memory. After optimization, the *entire* analysis suite (all 40 analyses in Figure 3) on XS completes in about 5 minutes using 40 GB. For the other twelve programs in Table 2, totaling 82K lines, the entire suite completes in just 6 seconds. Note that running multiple analysis clients together in one process enables the sharing of analysis results and reduces recomputation, but increases peak memory usage by about 1 GB per additional analysis on this million-line XS program. Therefore, running `missing-reset` (involving 19 analyses) on XS takes about 1 minute and 16 GB, whereas executing the entire suite (11 bug detection clients involving 40 analyses) completes in about 5 minutes using 40 GB.

5.2.4 A Complementary Case Study on Deadlock Analysis. We applied the same evaluation methodology used in the `missing-reset` case study to another important bug-detection client, the `dead-lock` analysis introduced in Section 5.1.2. Notably, `dead-lock` relies primarily on fundamental analyses targeting hardware synchronization characteristics—such as `guard-inputs` and `simple-computation-model` (see the bottom-left of Figure 3)—rather than the hardware-understanding analyses central to `missing-reset`. Despite these differences, the results support the same findings as the original case study. Due to space constraints, we provide the detailed result tables, figures, and corrsponding explanations for `dead-lock` to our supplementary material, and summarize the main results below. First, across 13 programs, `dead-lock` reports 52 alarms, 2 of which correspond to the confirmed bugs in Table 1; this number of alarms is comparably manageable to `missing-reset`, which reports 110 alarms in total, 5 of which are confirmed bugs. Second, the overall run time also remains practical: on XS, `dead-lock` completes in 34.0 s, compared with 74.3 s for `missing-reset`. Moreover, fundamental analyses still account for most of the workload, contributing 30.8 s out of 34.0 s (90%) for `dead-lock`, which is close to the 94% observed for `missing-reset`. Third, developing `dead-lock` on top of our suite requires about 260 LoC, whereas implementing it together with its dependencies from scratch would require about 5,000 LoC; this reduction is comparable to `missing-reset` (540 versus 9,700 LoC).

6 Related Work

In recent years, the software community has shown growing interest in applying programming-language techniques to hardware [25, 27, 35, 55, 60, 77, 84, 85, 94, 98, 101, 107]. Encouraged by these works, we apply static analysis to Verilog for hardware bug detection and introduce supporting infrastructure for Verilog static analysis. This section first reviews the most relevant prior work on applications and infrastructures for Verilog static analysis, and then discusses other relevant work.

Applications of Verilog Static Analysis. The application of static analysis in Verilog is still nascent compared to software, with current approaches primarily relying on syntax- or pattern-based checks on the AST for hardware bug detection and security. For example, linters [26, 50, 88, 99] are widely used to enforce hardware code style and catch simple bugs, while recent work [5] continues to rely on similar pattern-based analyses to identify hardware vulnerabilities. However, such analyses lack the sophistication necessary to detect deeper, semantic-level bugs and vulnerabilities like those discussed in Section 5.1. At the same time, the rapid growth in hardware complexity and diversity has introduced new challenges in reliability, security, and maintainability [47, 58, 76, 109], which are beyond the reach of such simple static analyses and often force developers to rely on resource-intensive testing and verification [16, 36, 49, 57, 106].

This coincides with the history of static analysis in software: researchers and engineers initially relied on simple pattern-based checks [52, 56], testing, and verification [37, 51], until the emergence of sophisticated static analyses enabled effective bug detection and vulnerability identification, even for very large codebases [13, 14, 48, 97]. We believe the vast potential of sophisticated static analysis for addressing emerging challenges in hardware is yet to be explored. Our analyses, which can detect a broader range of hardware bugs beyond the capabilities of currently dominant linting-based Verilog analyses [26, 50, 88, 99], serve to demonstrate this potential.

Infrastructures for Verilog Static Analysis. Effective infrastructures are essential for reducing the development cost of sophisticated static analyses and lowering the barrier to future research. Currently, Verilog analysis development generally depends on three kinds of options, each limiting the sophistication of analysis: (1) Linters such as Slang [88], Verible [26], SVLint [50], and Verilator-Lint [99] provide ASTs where analyses are confined to syntax- or pattern-based approaches. The limitations of these analyses have been discussed in the previous paragraph. (2) Tools like Pyverilog [102] and VeriPy [89] provide hardware abstractions beyond ASTs but lack the infrastructure to address critical hardware features like synchronization, concurrency, and resetting, highly limiting their utility for sophisticated analysis tasks. (3) While CIRCT [62], a comprehensive hardware compiler framework primarily used for synthesis, can also be adapted for static analyses, it likewise lacks the fundamental analyses necessary for building sophisticated analyses that require intricate value flows and deep semantic reasoning. In addition, CIRCT does not provide infrastructure tailored to Verilog analysis, such as an analysis-specific Verilog IR or an analysis manager. Given these limitations, we developed a complete Verilog analysis framework from scratch (as shown in Section 4) to support future community efforts toward sophisticated Verilog static analyses.

Traditional Verilog Verification Methods. Detecting deep issues in Verilog designs has traditionally depended on resource-intensive approaches such as simulation-based testing [49], fuzzing [57, 106], and formal verification techniques, typically based on SAT/SMT solving, such as IC3 [15, 16], UCLID5 [87, 95], PrediCore [8], and SymbiYosys [4]. Among these, formal verification is most closely related to static analysis, as both employ formal reasoning to verify hardware properties. Formal verification provides strong guarantees for user-specified hardware properties but suffers from state explosion, which limits its scalability to large designs. In contrast, static analysis can efficiently scale to entire hardware designs, but often relies on overapproximations, leading to possible false positives or negatives. To mitigate this limitation, we have focused on improving the precision and soundness of static analyses, making them practical for real-world programs, as discussed in Section 3.2 and evaluated in Section 5.2. Ultimately, we view static analysis as a lightweight solution for checking hardware properties across a wide range of design scales. It can be effectively combined with formal verification, which remains more suitable for small-scale module verification. We plan to further enhance the effectiveness of these analyses and propose additional analyses in future research.

Hardware Type Systems. Type systems are effective programming-language techniques for enforcing correctness properties at compile time. A range of novel type systems for hardware is introduced, providing insights into how types can enforce hardware-specific properties. These advances have enabled notable applications, such as information-flow security [39, 67, 68, 115] and safe module composition [27, 85]. As the underlying formalisms of type systems and static analysis have been unified [33], future Verilog static analyses can draw inspiration from their designs. Specifically, Verilog static analyses can approximate the information encoded by these type systems to detect bugs they aim to prevent, although such information is absent in standard Verilog [1] types. For example, timeline types [85] encode information about the latency and throughput of hardware modules to eliminate structural hazards in statically scheduled hardware pipelines. Inspired by this design, future Verilog static analyses could approximate such information to detect structural hazards in existing codebases. Similarly, Wire-Sorts [27], a type system designed to exclude combinational loops, can inspire Verilog static analyses for detecting combinational loops.

Synergies Between Static Analysis and LLMs. Recently, techniques based on LLMs have demonstrated significant potential in both software and hardware domains for detecting bugs and vulnerabilities [6, 117]. However, directly applying LLMs to programs is limited in their ability to find deep bugs, as this requires complex program reasoning [74]. Therefore, this progress has spurred active research in integrating static analysis with LLM prompting in software [65, 74, 117]. These approaches typically involve providing LLMs with key information distilled from static analysis, enabling LLMs to better understand program semantics, reducing the need for frequent and computationally expensive LLM calls, and ultimately yielding more effective and efficient results. Such strategies are also promising for hardware, highlighting new opportunities for hardware static analysis with LLM support.

Framework Design References. Our framework design (Section 4) is inspired by well-engineered analysis and compiler frameworks such as LLVM [63], Tai-e [103], Soot [108], and WALA [40]. In particular, our analysis manager orchestrates interdependent analyses over our Verilog IR in a way analogous to LLVM's pass manager, which coordinates interdependent passes over LLVM IR. We adopt this decoupled and extensible architecture to support long-term maintainability and to facilitate the integration of new analyses into the framework.

7 Conclusion

This work takes a preliminary step toward unlocking the full potential of sophisticated static analysis for Verilog. We presented a suite of sophisticated Verilog analyses that successfully detected 18 bugs from real-world hardware projects—none detectable by existing analysis tools [26, 50, 88, 99]—including 9 previously unknown bugs in popular open-source projects (averaged over 1.5K GitHub stars), all confirmed by developers. In building these analyses, we found that a wide range of fundamental hardware information (e.g., clocks, registers, synchronization, and control/data flows) is essential for effective bug detection. Accordingly, we designed a well-organized suite of analyses that extract such information from Verilog programs, establishing a common foundation for future research in Verilog static analysis. We implemented these analyses along with the necessary infrastructure—including a Verilog analysis IR, its front end, and an analysis manager—in *Qihe*, a framework developed entirely from scratch with over 100K lines of code. Qihe not only enables users to run our analyses on their hardware designs for detecting deep bugs, but also empowers analysis developers to create diverse new analyses by leveraging the existing foundations. By open-sourcing Qihe, we hope to foster collaboration between researchers and engineers from both the software and hardware communities to advance static analysis for hardware.

Data-Availability Statement

Qihe is open source and can be obtained from its website (https://qihe.pascal-lab.net); to facilitate the development of various analysis clients, we provide extensive documentation at https://qihe-docs.pascal-lab.net. Due to space constraints, this paper omits details of Qihe's framework design (e.g., IR design and front end); we provide these details in an extended version [24] available at https://arxiv.org/abs/2601.11408. We also release an artifact [22] that enables reproducing all experimental results and includes full details of the identified bugs; it is available at https://doi.org/10.5281/zenodo.18921919. To reproduce the results, please follow the instructions in the `README.pdf` included in the artifact package. As described in Section 5.2, we provide supplementary material [23] detailing the complementary evaluation case study, available at https://doi.org/10.5281/zenodo.19061406.

Acknowledgments

This project was undertaken out of curiosity and a passion for exploration. We are grateful to Xuanlin Li (HiSilicon, Huawei) for introducing us to the significance of static analysis in hardware. We sincerely thank the anonymous reviewers for their thoughtful and constructive comments, which substantially strengthened the paper. We also thank Yufei Liang, Teng Zhang, Menglong Chen, and Zhiwei Zhang for their helpful suggestions on improving earlier versions of the paper, and we acknowledge Zhongyi Cai and Sitao Lin as early users of Qihe for providing constructive feedback. This work was supported in part by National Key R&D Program of China under Grant No. 2023YFB4503804, National Natural Science Foundation of China under Grant Nos. 62402210 and 92582201, and Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China under Grant No. JYB2025XDXM118. We thank the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China, for its support. Tian Tan, the co-corresponding author, was also supported by the Xiaomi Foundation.

References

[1] 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), 1–590. doi:10.1109/IEEESTD.2006.99495

[2] 2025. HiSilicon. https://www.hisilicon.com/en.

[3] 2025. T-Head. https://www.t-head.cn/?lang=en.

[4] 2026. SymbiYosys: Front-End for Yosys-Based Formal Verification Flows. https://github.com/YosysHQ/sby.

[5] Baleegh Ahmad, Wei-Kai Liu, Luca Collini, Hammond Pearce, Jason M. Fung, Jonathan Valamehr, Mohammad Bidmeshki, Piotr Sapiecha, Steve Brown, Krishnendu Chakrabarty, Ramesh Karri, and Benjamin Tan. 2022. Don't CWEAT It: Toward CWE Analysis Techniques in Early Stages of Hardware Design. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design* (San Diego, California) *(ICCAD '22)*. Association for Computing Machinery, New York, NY, USA, Article 157, 9 pages. doi:10.1145/3508352.3549369

[6] Baleegh Ahmad, Shailja Thakur, Benjamin Tan, Ramesh Karri, and Hammond Pearce. 2024. On Hardware Security Bug Code Fixes by Prompting Large Language Models. *IEEE Transactions on Information Forensics and Security* 19 (2024), 4043–4057. doi:10.1109/TIFS.2024.3374558

[7] Ken Albin. 2016. The Cost of SoC Bugs. https://dvcon-proceedings.org/document/the-cost-of-soc-bugs/.

[8] Sophie Andrews, Matthew Sotoudeh, and Clark Barrett. 2025. Efficient SAT-Based Bounded Model Checking of Evolving Systems. In *2025 Design, Automation & Test in Europe Conference (DATE)*. 1–7. doi:10.23919/DATE64628.2025.10993135

[9] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *SIGPLAN Not.* 49, 6 (June 2014), 259–269. doi:10.1145/2666356.2594299

[10] Francine Bacchini, Robert F. Damiano, Bob Bentley, Kurt Baty, Kevin Normoyle, Makoto Ishii, and Einat Yogev. 2004. Verification: what works and what doesn't. In *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*, Sharad Malik, Limor Fix, and Andrew B. Kahng (Eds.). ACM, 274. doi:10.1145/996566.996648

[11] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. 2016. OpenPiton: An Open Source

Manycore Research Framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 217–232. doi:10.1145/2872362.2872414

[12] Lionel Bening and Harry Foster. 2001. *RTL Logic Simulation.* Springer US, Boston, MA, 69–101. doi:10.1007/0-306-47631-2_5

[13] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. doi:10.1145/1646353.1646374

[14] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '03)*. Association for Computing Machinery, New York, NY, USA, 196–207. doi:10.1145/781131.781153

[15] Aaron R. Bradley. 2011. SAT-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation* (Austin, TX, USA) *(VMCAI'11)*. Springer-Verlag, Berlin, Heidelberg, 70–87.

[16] Aaron R. Bradley and Zohar Manna. 2007. Checking Safety by Inductive Generalization of Counterexamples to Induction. In *Proceedings of the Formal Methods in Computer Aided Design (FMCAD '07)*. IEEE Computer Society, USA, 173–180. doi:10.1109/FAMCAD.2007.15

[17] Yuandao Cai, Peisen Yao, Chengfeng Ye, and Charles Zhang. 2023. Place Your Locks Well: Understanding and Detecting Lock Misuse Bugs. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 3727–3744. https://www.usenix.org/conference/usenixsecurity23/presentation/cai-yuandao

[18] Yuandao Cai, Peisen Yao, and Charles Zhang. 2021. Canary: practical static detection of inter-thread value-flow bugs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1126–1140. doi:10.1145/3453483.3454099

[19] Ciro Luiz Araujo Ceissler. 2025. An SHA512 Accelerator. https://github.com/efeslab/hardcloud/tree/e28ca96fdbb67904ef909fb04e026cf6dc724198/samples/sha512.

[20] Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 363–374. doi:10.1145/1542476.1542517

[21] Menglong Chen, Tian Tan, Minxue Pan, and Yue Li. 2025. PacDroid: A Pointer-Analysis-Centric Framework for Security Vulnerabilities in Android Apps. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 2803–2815. doi:10.1109/ICSE55347.2025.00208

[22] Qinlin Chen, Nairen Zhang, Jinpeng Wang, Jiacai Cui, Tian Tan, Xiaoxing Ma, Chang Xu, Jian Lu, and Yue Li. 2026. *Exploiting Sophisticated Static Analysis for Verilog (Artifact)*. doi:10.5281/zenodo.19486343

[23] Qinlin Chen, Nairen Zhang, Jinpeng Wang, Jiacai Cui, Tian Tan, Xiaoxing Ma, Chang Xu, Jian Lu, and Yue Li. 2026. Exploiting Sophisticated Static Analysis for Verilog (Supplementary Material). doi:10.5281/zenodo.19414138

[24] Qinlin Chen, Nairen Zhang, Jinpeng Wang, Jiacai Cui, Tian Tan, Xiaoxing Ma, Chang Xu, Jian Lu, and Yue Li. 2026. Qihe: A General-Purpose Static Analysis Framework for Verilog. arXiv:2601.11408 [cs.PL] https://arxiv.org/abs/2601.11408

[25] Qinlin Chen, Nairen Zhang, Jinpeng Wang, Tian Tan, Chang Xu, Xiaoxing Ma, and Yue Li. 2023. The Essence of Verilog: A Tractable and Tested Operational Semantics for Verilog. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 230 (oct 2023), 30 pages. doi:10.1145/3622805

[26] Chipsalliance. 2025. Verible. https://chipsalliance.github.io/verible/.

[27] Michael Christensen, Timothy Sherwood, Jonathan Balkind, and Ben Hardekopf. 2021. Wire sorts: a language abstraction for safe hardware composition. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 175–189. doi:10.1145/3453483.3454037

[28] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. 2012. *Model Checking and the State Explosion Problem.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1–30. doi:10.1007/978-3-642-35746-6_1

[29] HACK@DAC-18 Competition Committee. 2018. Hack@DAC 2018 Buggy SoC. https://github.com/HACK-EVENT/hackatdac18.git.

[30] HACK@DAC-21 Competition Committee. 2021. Hack@DAC 2021 Buggy SoC. https://github.com/HACK-EVENT/hackatdac21.git.

[31] HACK@DAC Competition Committee. 2024. What is HACK@DAC? https://www.dac.com/Conference/HackDAC.

[32] CWE Community. 2024. CWE-1271: Uninitialized Value on Reset for Registers Holding Security Settings. https://cwe.mitre.org/data/definitions/1271.html.

[33] Patrick Cousot. 1997. Types as abstract interpretations. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 316–331. doi:10.1145/263699.263744

[34] Alexandru Dura, Christoph Reichenbach, and Emma Söderberg. 2021. JavaDL: automatically incrementalizing Java bug pattern detection. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 165 (Oct. 2021), 31 pages. doi:10.1145/3485542

[35] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. 2020. Type-directed scheduling of streaming accelerators. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 408–422. doi:10.1145/3385412.3385983

[36] Niklas Een, Alan Mishchenko, and Robert Brayton. 2011. Efficient implementation of property directed reachability. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*. 125–134.

[37] E. Allen Emerson and Edmund M. Clarke. 1980. Characterizing correctness properties of parallel programs using fixpoints. In *Automata, Languages and Programming*, Jaco de Bakker and Jan van Leeuwen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 169–181.

[38] E. Allen Emerson and Joseph Y. Halpern. 1986. "Sometimes" and "not never" revisited: on branching versus linear time temporal logic. *J. ACM* 33, 1 (Jan. 1986), 151–178. doi:10.1145/4904.4999

[39] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2017. Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 555–568. doi:10.1145/3037697.3037739

[40] Watson Libraries for Analysis. 2006. WALA. http://wala.sf.net.

[41] Alex Forencich. 2025. Verilog AXI Stream Components. https://github.com/alexforencich/verilog-axis.

[42] Harry Foster. 2024. IC/ASIC Functional Verification Trend Report - 2024. https://verificationacademy.com/topics/planning-measurement-and-analysis/2024-siemens-eda-and-wilson-research-group-functional-verification-study/.

[43] Dan Gisselquist. 2025. SD-Card controller. https://github.com/ZipCPU/sdspi.

[44] Dan Gisselquist. 2025. ZipCPU: A Small, Lightweight, RISC CPU Soft Core. https://github.com/ZipCPU/zipcpu/.

[45] Steve Golson and Leah Clark. 2016. Language wars in the 21st century: verilog versus vhdl–revisited. *Synopsys Users Group (SNUG)* (2016).

[46] Neville Grech and Yannis Smaragdakis. 2017. P/Taint: unified points-to and taint analysis. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 102 (Oct. 2017), 28 pages. doi:10.1145/3133926

[47] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *2011 IEEE Symposium on Security and Privacy*. 490–505. doi:10.1109/SP.2011.22

[48] Ben Hardekopf and Calvin Lin. 2007. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 290–299. doi:10.1145/1250734.1250767

[49] N B Harshitha, Y G Praveen Kumar, and M Z Kurian. 2021. An Introduction to Universal Verification Methodology for the digital design of Integrated circuits (IC's): A Review. In *2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS)*. 1710–1713. doi:10.1109/ICAIS50930.2021.9396034

[50] Naoya Hatta. 2025. svlint. https://github.com/dalance/svlint/tree/master.

[51] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. doi:10.1145/363235.363259

[52] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. *SIGPLAN Not.* 39, 12 (Dec. 2004), 92–106. doi:10.1145/1052883.1052895

[53] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. 2021. Hardware Information Flow Tracking. *ACM Comput. Surv.* 54, 4, Article 83 (May 2021), 39 pages. doi:10.1145/3447867

[54] Wei Hu, Baolei Mao, Jason Oberg, and Ryan Kastner. 2016. Detecting Hardware Trojans with Gate-Level Information-Flow Tracking. *Computer* 49, 8 (2016), 44–52. doi:10.1109/MC.2016.225

[55] Minseong Jang, Jungin Rhee, Woojin Lee, Shuangshuang Zhao, and Jeehoon Kang. 2024. Modular Hardware Design of Pipelined Circuits with Hazards. *Proc. ACM Program. Lang.* 8, PLDI, Article 148 (June 2024), 24 pages. doi:10.1145/3656378

[56] Stephen C Johnson. 1977. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill.

[57] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3219–3236. https://www.usenix.org/conference/usenixsecurity22/presentation/kande

[58] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2020. Spectre attacks: exploiting speculative execution. *Commun. ACM* 63, 7 (June 2020), 93–101. doi:10.1145/3399742

[59] Christian Krieg, Clifford Wolf, Axel Jantsch, and Tanja Zseby. 2017. Toggle MUX: How X-Optimism Can Lead to Malicious Hardware *(DAC '17)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. doi:10.1145/3061639.3062328

[60] Shriram Krishnamurthi, Benjamin S. Lerner, and Liam Elberty. 2019. The Next 700 Semantics: A Research Challenge. In *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA (LIPIcs, Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:14. doi:10.4230/LIPIcs.SNAPL.2019.9

[61] Rejeesh Kutty. 2025. HDL libraries and projects. https://github.com/analogdevicesinc/hdl.

[62] Chris Lattner. 2025. CIRCT: Circuit IR Compilers and Tools. https://circt.llvm.org/.

[63] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[64] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 2022. Finding real bugs in big programs with incorrectness logic. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 81 (April 2022), 27 pages. doi:10.1145/3527325

[65] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 111 (April 2024), 26 pages. doi:10.1145/3649828

[66] Haonan Li, Hang Zhang, Kexin Pei, and Zhiyun Qian. 2025. Towards More Accurate Static Analysis for Taint-Style Bug Detection in Linux Kernel. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 380–392. doi:10.1109/ASE63991.2025.00039

[67] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. 2014. Sapper: a language for hardware-level security policy enforcement. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, Rajeev Balasubramonian, Al Davis, and Sarita V. Adve (Eds.). ACM, 97–112. doi:10.1145/2541940.2541947

[68] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: a hardware description language for secure information flow. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 109–120. doi:10.1145/1993498.1993512

[69] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Precision-Guided Context Sensitivity for Pointer Analysis. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (10 2018), 141:1–141:29. doi:10.1145/3276511

[70] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2018. Scalability-first pointer analysis with self-tuning context-sensitivity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140. doi:10.1145/3236024.3236041

[71] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Transactions on Programming Languages and Systems* (2020). doi:10.1145/3295739

[72] Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and Analyzing Java Reflection. *ACM Transactions on Software Engineering and Methodology* 28, 2 (4 2019), 7:1–7:50. doi:10.1145/3295739

[73] Yue Li, Tian Tan, Yifei Zhang, and Jingling Xue. 2016. Program Tailoring: Slicing by Sequential Criteria. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:27. doi:10.4230/LIPIcs.ECOOP.2016.15

[74] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=9LdJDU7E91

[75] Yufei Liang, Teng Zhang, Ganlin Li, Tian Tan, Chang Xu, Chun Cao, Xiaoxing Ma, and Yue Li. 2025. Pointer Analysis for Database-Backed Applications. *Proc. ACM Program. Lang.* 9, PLDI, Article 204 (June 2025), 25 pages. doi:10.1145/3729307

[76] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. 2020. Meltdown: reading kernel memory from user space. *Commun. ACM* 63, 6 (May 2020), 46–56. doi:10.1145/3357033

[77] Andreas Lööw. 2025. The Simulation Semantics of Synthesisable Verilog. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 126 (April 2025), 26 pages. doi:10.1145/3720484

[78] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Haoyang Zhang, Andrew Quinn, and Baris Kasikci. 2022. Debugging in the brave new world of reconfigurable hardware. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 946–962. doi:10.1145/3503222.3507701

[79] Jiacheng Ma, Gefei Zuo, Kevin Loughlin, Haoyang Zhang, Andrew Quinn, and Baris Kasikci. 2022. Debugging in the brave new world of reconfigurable hardware (Bugbase Artifact). https://github.com/efeslab/hardware-bugbase.git.

[80] Wenjie Ma, Shengyuan Yang, Tian Tan, Xiaoxing Ma, Chang Xu, and Yue Li. 2023. Context Sensitivity without Contexts: A Cut-Shortcut Approach to Fast and Precise Pointer Analysis. *Proceedings of the ACM on Programming Languages* 7, PLDI (6 2023), 128:1–128:26. doi:10.1145/3591242

[81] Xingyu Meng, Shamik Kundu, Arun K. Kanuparthi, and Kanad Basu. 2022. RTL-ConTest: Concolic Testing on RTL for Detecting Security Vulnerabilities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 3 (2022), 466–477. doi:10.1109/TCAD.2021.3066560

[82] Cody Miller. 2010. State Machine Design Techniques. https://www.edn.com/state-machine-design-techniques/.

[83] Sudhamshu B N. 2025. Implementing 32 Verilog Mini Projects. https://github.com/sudhamshu091/32-Verilog-Mini-Projects.

[84] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. 2020. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 393–407. doi:10.1145/3385412.3385974

[85] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. *Proc. ACM Program. Lang.* 7, PLDI, Article 120 (June 2023), 25 pages. doi:10.1145/3591234

[86] Konstantin Pavlov. 2025. Must-have Verilog SystemVerilog Modules. https://github.com/pConst/basic_verilog.

[87] Elizabeth Polgreen, Kevin Cheang, Pranav Gaddamadugu, Adwait Godbole, Kevin Laeufer, Shaokai Lin, Yatin A. Manerkar, Federico Mora, and Sanjit A. Seshia. 2022. UCLID5: Multi-modal Formal Modeling, Verification, and Synthesis. In *Computer Aided Verification*, Sharon Shoham and Yakir Vizel (Eds.). Springer International Publishing, Cham, 538–551.

[88] Michael Popoloski. 2025. Slang. https://sv-lang.com/.

[89] Md Imtiaz Rashid and B. Carrion Schaefer. 2024. VeriPy: A Python-Powered Framework for Parsing Verilog HDL and High-Level Behavioral Analysis of Hardware. In *2024 IEEE 17th Dallas Circuits and Systems Conference (DCAS)*. 1–6. doi:10.1109/DCAS61159.2024.10539889

[90] Hassan Salmani, Mohammad Tehranipoor, and Ramesh Karri. 2013. On design vulnerability analysis and trust benchmarks development. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 471–474. doi:10.1109/ICCD.2013.6657085

[91] M. Samsoniuk. 2025. DarkRISCV. https://github.com/darklife/darkriscv.

[92] Shaker Sarwary and Michael A Beaver. 2005. A Systematic Approach to Verifying FSMs. https://www.edn.com/a-systematic-approach-to-verifying-fsms/.

[93] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. 2018. Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX. In *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*. 1–3. doi:10.1109/S3S.2018.8640145

[94] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: a multi-level intermediate representation for hardware description languages. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 258–271. doi:10.1145/3385412.3386024

[95] Sanjit A. Seshia and Pramod Subramanyan. 2018. UCLID5: integrating modeling, verification, synthesis and learning. In *Proceedings of the 16th ACM-IEEE International Conference on Formal Methods and Models for System Design* (Beijing, China) *(MEMOCODE '18)*. IEEE Press, 1–10.

[96] Bicky Shakya, Tony He, Hassan Salmani, Domenic Forte, Swarup Bhunia, and Mark Tehranipoor. 2017. Benchmarking of hardware trojans and maliciously affected circuits. *Journal of Hardware and Systems Security* 1 (2017), 85–102. doi:10.1007/s41635-017-0001-6

[97] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 693–706. doi:10.1145/3192366.3192418

[98] Zachary D. Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. 2023. Loop Rerolling for Hardware Decompilation. *Proc. ACM Program. Lang.* 7, PLDI, Article 123 (June 2023), 23 pages. doi:10.1145/3591237

[99] Wilson Snyder. 2025. Verilator. https://veripool.org/verilator.

[100] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin slicing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Diego, California, USA) *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 112–122. doi:10.1145/1250734.1250748

[101] Milijana Surbatovich, Naomi Spargo, Limin Jia, and Brandon Lucia. 2023. A Type System for Safe Intermittent Computing. *Proc. ACM Program. Lang.* 7, PLDI, Article 136 (June 2023), 25 pages. doi:10.1145/3591250

[102] Shinya Takamaeda-Yamazaki. 2015. Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL. In *Applied Reconfigurable Computing*, Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz (Eds.). Springer International Publishing, Cham, 451–460.

[103] Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1093–1105. doi:10.1145/3597926.3598120

[104] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making Pointer Analysis More Precise by Unleashing the Power of Selective Context Sensitivity. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (10 2021), 147:1–147:27. doi:10.1145/3485524

[105] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and Precise Points-to Analysis: Modeling the Heap by Merging Equivalent Automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, Barcelona, Spain, June 18-23, 2017 (PLDI)*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 278–291. doi:10.1145/3062341.3062360

[106] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2022. Fuzzing Hardware Like Software. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3237–3254. https://www.usenix.org/conference/usenixsecurity22/presentation/trippel

[107] Lenny Truong and Pat Hanrahan. 2019. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA (LIPIcs, Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:21. doi:10.4230/LIPICS.SNAPL.2019.7

[108] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: a Java bytecode optimization framework. In *CASCON First Decade High Impact Papers (CASCON '10)*. IBM Corp., USA, 214–224. doi:10.1145/1925805.1925818

[109] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 991–1008.

[110] VLSIFacts. 2025. Understanding Pipeline Design in Verilog: How to Stage Data Across Clock Cycles for High Performance. https://vlsifacts.com/understanding-pipeline-design-in-verilog-how-to-stage-data-across-clock-cycles-for-high-performance/.

[111] VLSIFacts. 2025. Understanding Reset Signals in Digital Design: Types, Pros & Cons, and Best Practices. https://vlsifacts.com/understanding-reset-signals-in-digital-design-types-pros-cons-and-best-practices/.

[112] Claire Wolf. 2025. Yosys Open SYnthesis Suite. https://yosyshq.net/yosys/.

[113] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. 2022. Towards Developing High Performance RISC-V Processors Using Agile Methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1178–1199. doi:10.1109/MICRO56248.2022.00080

[114] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V. Krishnamurthy, and Paul Yu. 2020. UBITect: a precise and scalable method to detect use-before-initialization bugs in Linux kernel. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 221–232. doi:10.1145/3368089.3409686

[115] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 503–516. doi:10.1145/2694344.2694372

[116] Xiangyu Zhang, Yucheng Su, Lingling Fan, Miaoying Cai, and Sen Chen. 2025. GlassWing: A Tailored Static Analysis Approach for Flutter Android Apps. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 521–533.

[117] Xin Zhou, Sicong Cao, Xiaobing Sun, and David Lo. 2024. Large Language Model for Vulnerability Detection and Repair: Literature Review and the Road Ahead. *ACM Trans. Softw. Eng. Methodol.* (Dec. 2024). doi:10.1145/3708522 Just Accepted.

Received 2025-11-12; accepted 2026-04-03