Lost in Translation: Enabling Confused Deputy Attacks on EDA Software with TransFuzz

Flavien Solt
ETH Zurich

Kaveh Razavi
ETH Zurich

Abstract

We introduce MIRTL, a confused deputy attack on EDA software such as simulators or synthesizers. MIRTL relies on *gadgets* that exploit vulnerabilities in the EDA software's translation of RTL to lower-level representations. Invisible to white-box testing and verification methods, MIRTL gadgets harden traditional hardware trojans, enabling unprecedentedly stealthy attacks. To discover translation bugs, our new fuzzer, called TRANSFUZZ, generates randomized RTL designs containing many operators with complex interconnections for triggering translation bugs. The expressiveness of RTL, however, makes the construction of a golden RTL model for detecting deviations due to translation bugs challenging. To address this, TRANSFUZZ relies on comparing signal outputs from multiple RTL simulators for detecting vulnerabilities. TRANSFUZZ uncovers 20 translation vulnerabilities among 31 new bugs (25 CVEs) in four popular open-source EDA applications. We show how MIRTL gadgets harden traditional backdoors against white-box countermeasures and demonstrate a real-world instance of a MIRTL-hardened backdoor in the CVA6 RISC-V core.

1 Introduction

Community-driven open-source hardware development is on the rise [3, 4, 11, 12, 30, 45, 46, 56, 71, 72, 92, 108], exposing Electronic Design Automation (EDA) software used for Register-Transfer Level (RTL) simulation and synthesis to hardware designs from many, potentially distrusting, developers. Like standard software, RTL simulators and synthesizers are subject to standard vulnerabilities such as buffer overflows. The unique nature of hardware development flows, however, exposes EDA software to a more silent and sinister class of vulnerabilities that we explore in this paper. These vulnerabilities enable a hardware developer or a malicious intermediate EDA application to turn a seemingly benign piece of hardware design into synthesized malicious hardware.

Translation bugs. RTL simulators are used in all hardware development stages for testing design validity by transforming the RTL, expressed in a Hardware Description Language (HDL), into a lower-level model suitable for execution on the designer's system. Once the hardware design is final in the late stages of hardware development, the HDL representation is transformed into a gate-level netlist by a synthesizer. The translation of the HDL by an RTL simulator or a synthesizer to lower-level representations involves a variety of complex optimizations, e.g., for performance or area. This complexity can cause the EDA software to mistranslate the given HDL design. We show that we can reliably use these translation bugs to create a new class of attacks that we call MIRTL (Mistranslated RTL). When targeting RTL simulators or other validation EDA software, MIRTL enables certain HDL behavior not to register during validation, but to appear in the synthesized hardware. When targeting synthesizers, simulators register all the values correctly, but the malicious behavior is injected by the synthesizer. To employ MIRTL, attackers must discover translation bugs in the target EDA software.

TRANSFUZZ. Fuzzing is a common technique to find vulnerabilities in software [10,17,19,34,37–40,70,78,81,94,100,101] and hardware [14,16,53,57,88,104] by generating input against the hardware or software interface. The input to EDA software is an arbitrary RTL design that takes arbitrary stimuli. Hence, our new fuzzer, TRANSFUZZ, generates randomized RTL designs and stimulates them with randomized inputs under different RTL simulators and synthesizer. TRANSFUZZ ensures that the generated RTL designs are sufficiently random to provide ample opportunities for optimizations. Generating randomized RTL at the HDL-level often results in regular representations and interconnections of macrocells after parsing by the EDA software. Instead, TRANSFUZZ generates randomized RTL designs at the macrocell-level which provides maximum flexibility for randomizing the number and type of macrocells as well as their interconnection.

TRANSFUZZ-generated designs may trigger translation bugs, yet detecting them is not trivial. Unlike software fuzzers

that can rely on crashes [10, 17, 19, 34, 37–39, 70, 81, 94, 100, 101] or sanitizers [40, 70], and hardware fuzzers that rely on golden models [14, 16, 53, 104] or the ISA itself [88], there exists no golden model for HDL and constructing one is challenging due to the expressiveness of HDLs. Instead, TRANSFUZZ generates randomized designs in a way that a configurable number of output signals provide a signature of the design over an arbitrary number of cycles. TRANSFUZZ then compares these values across different EDA applications to detect deviations, in the spirit of differential fuzzing [35, 65, 76]. Given the freedom in the specification of HDLs, however, we must slightly constrain the generation of our circuits to avoid false positives. Namely, TRANSFUZZ avoids race conditions with respect to signals that control state-saving elements and avoids the generation of 'undefined' X signals in the designs.

Our evaluation shows that TRANSFUZZ could discover 31 new bugs (25 CVEs) in 3 open-source RTL simulators: Verilator, CXXRTL and Icarus Verilog, and in the Yosys synthesizer. Of these 31 bugs, 20 are translation bugs, enabling MIRTL attacks on all of these four popular EDA applications.

Exploiting translation bugs. The Achilles' heel of classical RTL trojan hiding [15, 96] is the application of white-box techniques [7–9, 50, 52, 61, 67, 69, 80, 93, 99], which can detect RTL trojans despite complex triggering conditions that are based on some specific design state. To mitigate this unique weakness of classical RTL trojan hiding, we introduce MIRTL gadgets, a primitive that produces `1` under normal circumstances, and `0` under a translation bug. These gadgets make the trojans invisible to the white-box techniques applied by targeted EDA applications. For example, using MIRTL gadgets, we bypass standard formal hardware-level leakage detection techniques such as SAT solving [9, 42, 55, 63, 74, 77] and information flow tracking [2, 51, 89]. We further construct a malicious version of CVA6 [108] (a RISC-V CPU) that allows unprivileged software to leak supervisor memory in a way that is undetectable by all three simulators in our study.

Contributions. The following lists our contributions:

- We explore MIRTL, a new class of confused deputy attacks exploiting RTL translation bugs on EDA software.
- We design and implement TRANSFUZZ, a new fuzzer that uncovers translation bugs by generating complex randomized RTL designs and a differential approach for detecting these vulnerabilities.
- We apply TRANSFUZZ on 4 popular EDA applications to discover 31 new bugs, including 20 translation bugs.
- We instantiate MIRTL gadgets using the discovered translation bugs and build a malicious version of a CPU core that evades white-box trojan detection techniques such as SAT solving.

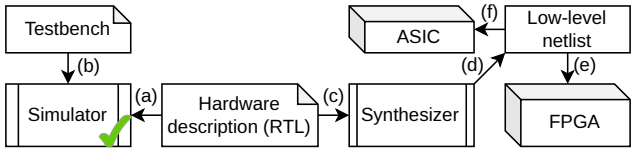

Figure 1: Typical EDA flow for digital hardware.

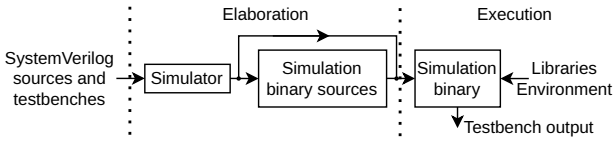

Figure 2: Elaboration and execution in RTL simulations.

Open sourcing. TRANSFUZZ is available at https://comsec.ethz.ch/mirtl/

2 Background

This section provides the necessary background about the hardware development flow, RTL simulation, and fuzzing.

2.1 Digital hardware development flow

A typical hardware development flow is illustrated in Figure 1 using a sequence of EDA applications. A hardware design (also known as circuit) is described at the RTL. Designs at such a level are usually expressed in VHDL or Verilog. The latter language has gained significant traction in the last decade, to the extent that almost all significant open-source hardware projects are written in Verilog, or in a language that is typically compiled to Verilog [5, 24, 79, 105]. The hardware design is simulated (a) with some testbench to check that the design fulfills the intended functionality (b). The validated design is then synthesized (c) to a lower-level netlist (d) that is destined for a specific FPGA family (e) or ASIC fabrication process (f). Simulation of the lower-level netlist are often orders of magnitude slower than pre-synthesis simulations [89]. Further steps depend on the target technology and often include steps like floorplanning, place and route.

2.2 RTL simulation

RTL simulators take hardware descriptions and testbenches, and can produce multiple forms of outputs such as binary pass/fail or a simulation trace. Popular open-source RTL simulators include Verilator [86], Icarus Verilog [102] and CXXRTL [103]. RTL simulators generally operate in two stages, illustrated in Figure 2. First, they take a hardware design and produce a simulation model. We call this step *elaboration*. Verilator and CXXRTL translate the design into C++ code meant to be compiled against the testbench to produce a simulation model, while Icarus produces an intermediate simulation model that is an input to a static executable (`vvp`). Second, they take a simulation model and

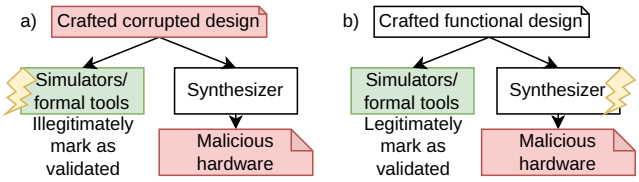

Figure 3: High-level MIRTL attacks exploiting (a) simulator bugs and (b) synthesizer bugs. Red hardware is malicious.

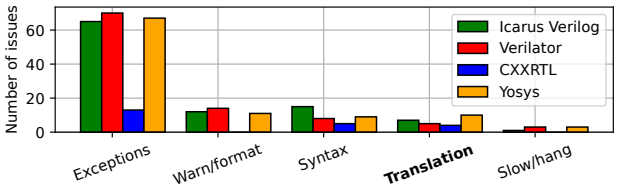

Figure 4: Recent public bug reports. Exceptions: crashes and assertion failures. Warn/format: wrong warning or text/trace formatting. Syntax: syntax-bound. Translation: translation bugs. Slow/hang: hang or unjustified significant slowdown.

a set of input values and run the simulation to produce the expected form of output. We call this step *execution*.

To simulate a design efficiently, RTL simulators perform a series of optimizations, such as constant propagation, dead code elimination, and common subexpression elimination. These optimizations are intended to preserve the design's functionality for all synthesizable parts that are not affected by undetermined "X" or high-impedance "Z" values.

2.3 Fuzzing

Fuzzers apply random inputs to a tested (hardware or software) unit and observe its behavior. Often classified as white-, gray- or black-box, they may rely on various mechanisms for collecting feedback to generate more effective inputs [10, 14, 16, 17, 19, 34, 37–40, 57, 70, 78, 81, 88, 94, 100, 101, 104]. Then, they use crashes [37, 40], sanitizers [28, 33, 82] reference implementation comparison [53] to detect bug occurrences.

Discussion. Existing fuzzers are not suitable for discovering deep bugs in EDA software, particularly translation bugs. The inputs must represent complex hardware and stimuli. For simulators, this even requires fuzzing the simulator and the simulation model. Furthermore, output deviations are not covered by the capabilities of traditional fuzzers.

3 MIRTL

Classical RTL trojans [7–9, 15, 52, 61, 80, 96] are almost undetectable once integrated in silicon hardware, as they might require an arbitrarily specific triggering condition to express their effect. However, white-box techniques such as SAT solving [9, 42, 55, 63, 74, 77], symbolic execution [1, 29, 36, 84] and information flow tracking [7–9, 52, 61, 80] can detect them in a white-box manner. We show that exploiting bugs in EDA software in a confused deputy scenario is a realistic and practical solution for inserting vulnerabilities that are *undetectable* under normal testing conditions, as we illustrate in Figure 3.

A translation bug in a simulator or in a formal verifier enables a MIRTL attack. In such an attack, an attacker-crafted behavior in HDL is not detectable by the EDA validation application and gets synthesized into malicious hardware (a). A translation bug in the synthesizer enables a MIRTL attack where the malicious behavior is not present in the HDL (hence also undetectable by simulators or formal tools) and injected by the synthesizer into malicious hardware (b).

We build TRANSFUZZ to find such translation bugs in designs that are expressed in synthesizable Verilog, which is nowadays arguably the most popular RTL language. Interestingly, only a few of such bugs were discovered in the last years despite the high activity around open-source EDA. In Figure 4, we show the distribution of up to the 100 most recent public bug reports from four popular open-source EDA applications. We provide a detailed methodology in Section 4.1. In total, only 22 of the 322 reported bugs are translation bugs. We provide an overview of challenges in the design and implementation of TRANSFUZZ and performing MIRTL attacks using the vulnerabilities that it discovers.

Threat Model. We consider a malicious hardware IP, an EDA software vendor or a contributor trying to discreetly alter a hardware design. We do not assume any property of the victim Verilog design. In the concrete exploits that we present, the victim uses some of the three popular open-source simulators Icarus Verilog [102], Verilator [86] or CXXRTL, and the state-of-practice open-source synthesis tool Yosys [103].

3.1 Overview of challenges

The first challenge in constructing MIRTL attacks regards the input structure and abstractions supplied to the EDA software.

> **Challenge 1.** Design suitable abstractions and structures for fuzzing RTL simulators.

In Section 4, we start by analyzing the recent bug reports from popular open-source EDA software to understand and summarize properties of hardware that empirically tend to reveal translation bugs. From this analysis, we deduce beneficial characteristics of the inputs that we supply to the EDA software to exert translation bugs. To comply with these requirements, we introduce netlists of macro-cells as a new preferred abstraction level. We propose a concrete way to construct these netlists, along with random stimuli sequences with suitable dimensions.

The second challenge regards the detection of bugs, both for simulators and synthesizers, as they do not have a golden model or a formal specification.

> **Challenge 2.** Detect bug occurrences.

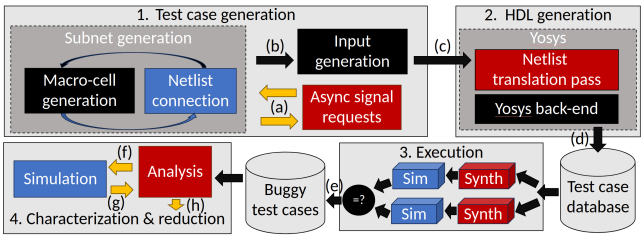

Figure 5: Overview of TRANSFUZZ.

In Section 5, we leverage differential fuzzing, using different simulators or parameters, to account for the absence of a formal model. TRANSFUZZ avoids all non-deterministic behaviors that would occur with fully random circuits to enable sound differential fuzzing by slightly constraining the input space. In Section 6, we evaluate the performance of TRANSFUZZ and describe the vulnerabilities that it uncovers.

The final challenge regards exploitation.

Challenge 3. Exploit the newly discovered bugs.

To implement practical exploits, Section 7 introduces the notion of MIRTL *gadget*, a synthesizable primitive that relies on simulator or synthesizer bugs to inject a mistaken value into the hardware design. We demonstrate that MIRTL gadgets can be used to escape white-box trojan detection techniques. We then inject a concrete kernel information leakage trojan into the CVA6 RISC-V CPU [108] where a MIRTL gadget protects a trojan from classical detection techniques.

3.2 Overview of TRANSFUZZ

Figure 5 summarizes the overall design of TRANSFUZZ. TRANSFUZZ operates in four steps. (1) It generates test case descriptions. (2) From the test case description, it generates a testbench and a standard Verilog design. (3) It then performs differential execution. (4) For each test case that triggered a mismatch, it categorizes and reduces it.

Test case generation. TRANSFUZZ first generates designs as networks of macrocells with randomized types, attributes and connections, as we will discuss in Section 4. It further populates asynchronous signals for state elements iteratively (a), and adds a stimuli component (b).

HDL generation. TRANSFUZZ then transmits a description (c) of the hardware component to an HDL generator written as a pass in the Yosys synthesizer. The output is a standard Verilog description of the final hardware component (d).

Test case execution. TRANSFUZZ submits the test case to several EDA applications for differential fuzzing and records mismatching instances (e).

Characterization and reduction. TRANSFUZZ performs simulations with modified parameters to characterize the bug (f) and may attempt to reduce it further in the hardware (space) and stimuli (time) dimensions (g). Eventually, TRANSFUZZ provides the reduced test case to be reported (h).

Table 1: Previous reports of translation bugs.

Report reference	Title
CXXRTL #4074	bmux does not mask the result
CXXRTL #3820	incorrect result of shl operator
CXXRTL #2780	CXXRTL [...] when routing signal via module
CXXRTL #2746	In CXXRTL edge eval is before calculating value
Verilator #4536	Shift when using streaming operator on 32 bit signal
Verilator #3824	Bit OR tree misoptimization
Verilator #3773	Seemingly incorrect terms in condition in V3Tristate
Verilator #3770	Signal skips flip-flop under some circumstances
Verilator #3509	Wire tie-off causing bad logic optimization
Verilator #3470	Wrong expression evaluation results
Verilator #3445	V3Const BitOpTree optimization is incorrect
Verilator #3409	complex assign in always_comb
Verilator #3399	Incorrect tristate enable logic
Yosys #4064	Frontend/AST: signed assign to indexed part-select
Yosys #4010	Synthesis optimization error, inconsistent simulation
Yosys #3879	LEC failed after yosys synthesis
Yosys #3867	Inconsistency Issue [...] opt_expr -fine Pass in Yosys
Yosys #3848	During synthesis [...] errors in register assignment
Yosys #3748	write_smt2: bugs caused by the '»' operator
Yosys #3680	Possible initialization issue in Xilinx DSP48E1 cell
Yosys #3431	Wrong smt-lib model behavior since yosys v0.15
Yosys #3360	synth_xilinx [...] output bit is driven 'Z'

4 Input Design

We first analyze existing bug reports to understand the properties of hardware that may trigger translation vulnerabilities in EDA software. Based on these observations, we propose a design and implementation of these potentially bug-triggering low-level inputs which we call subnets.

4.1 Analysis of past bug reports

We analyze previously-reported bugs in Verilator, Icarus Verilog and CXXRTL in the corresponding public issue trackers. For each simulator, we iterate through the bug reports, in reverse chronological order. We study the 100 most recent relevant bug reports for Verilator, Icarus and Yosys, and all the bug reports of the last three years for CXXRTL. We filter out bugs denied by the maintainers and duplicates.

Results. We summarize the translation bugs from Figure 4 in Table 1. Verilator and Yosys are the most affected. On the contrary, there has been no report of a translation bug in Icarus in the past 100 relevant bug reports (in a period of roughly three years), until TRANSFUZZ's finding reported in Section 6.5. We make a number of observations about these bugs. First, a diverse set of operator types such as shift, bmux and OR are required to cover some individual bugs and the complete set of these bugs.

Observation 1. Translation bugs are triggered by a diverse set of specific operator types.

Second, not a single one of these bugs can be triggered with a single-operator test case. Instead, all of them require some non-trivial interconnection of multiple operators, and these interconnection patterns vary from one bug to another.

Observation 2. Translation bugs require non-trivial and diverse interconnection patterns.

Third, out of the 22 bugs that were translational bugs among the 322 analyzed bugs, up to 21 of them can be triggered with operators with a width between 2 and 4 bits.

Observation 3. Most translation bugs can be triggered with narrow cells.

These observations lead us to the following requirements for the design of TRANSFUZZ.

Requirements. The test cases produced for fuzzing a single RTL simulator must satisfy the following requirements:

1. *Operational diversity.* The test case generator must produce a large diversity of basic hardware operations.
2. *Relational diversity.* The test case generator must produce a large diversity of interconnection patterns.
3. *Operator size distribution.* The distribution of operator sizes should favor narrower cells, without excluding larger ones completely.
4. *Soundness.* The hardware that is produced by the test case generator must comply with the fundamental principles of digital hardware designs. In particular, being exempt from combinational loops, and multi-driven nets.
5. *Syntactic correctness.* The hardware that is produced by the test case generator must be syntactically correct.

The last two requirements enable TRANSFUZZ to find translation bugs in valid RTL designs.

4.2 Test case generation

The usual testing approaches systematically lack one of the aforementioned requirements. HDL-level fuzzing, while finding front-end crashes, fails at producing any non-trivial hardware [27]. AST-level fuzzing guarantees syntactic correctness but hampers relational diversity [41, 107].

Input abstraction. Given these requirements, we propose the network of macrocells as an abstraction for fuzzing EDA software. The macrocell abstraction corresponds to simple synthesizable stateful (e.g., registers with enable signals) or combinational (e.g., adders) operators, as defined in the intermediate representation of EDA software like Yosys [103]. We construct a network of diverse interconnected macrocells to produce a sound relationally and operationally diverse hardware component. We use the terms cells and macrocells interchangeably.

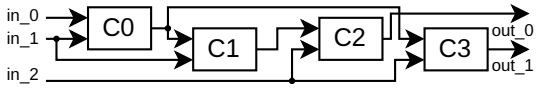

Figure 6: Subnet example.

4.2.1 Subnet structure and stimuli

We define subnets as hardware circuits in which the input of each cell is a design input or the output of another cell in the subnet. The *asynchronous* signals of stateful macrocells in the network, such as reset and clock signals, are supplied by dedicated inputs or by the output signals of other networks.

Figure 6 shows a four-cell subnet with three 32-bit input words and two 32-bit output words. The cells may individually be of any synthesizable combinational or stateful type.

Constructing sound subnets. Sound netlists require no combinational loops and require all wires to have a single driver. We enforce these rules at no computational cost by assigning specific roles to wires and ensuring that cells never output any signal that may impact their own input. Appendix A provides further details about enforcing these rules.

Stimuli specification. Stimuli values are lists of binary words. We define stimuli as a (temporal) sequence of pairs `(subnet_id, input_values)`, where `subnet_id` identifies the subnet or asynchronous wire, and `input_values` is a (spatial) list of values to be applied to the inputs to this entity. Stimuli pairs are applied sequentially to the respective inputs without any explicit form of reset in between, to let the hardware component enter subsequent states.

4.2.2 Implementation

Wire concatenations. In accordance with relational diversity, TRANSFUZZ allows cell inputs to be concatenations of multiple wires. Wire concatenation is necessary for connecting a cell's output to a wider input of another cell.

Input selection. Always selecting the subnet's input words as cell inputs would impair relational diversity. So would always selecting the output of the previously generated cell in the subnet, which would underwhelm optimizations that exploit parallelism in the circuit. TRANSFUZZ embeds an algorithm that selects the inputs of a cell as a potential concatenation of previous cell outputs and favors wires that have not been connected to a cell input yet. Algorithm 2 in Appendix B provides additional detail.

Cell width selection. Following an earlier requirement on cell widths, when generating each cell, we select the width of the cell's inputs and outputs following an offsetted geometric

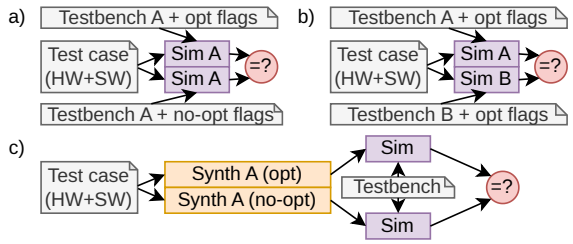

Figure 7: Differential fuzzing (DF). (a) Simulator (internal). (b) Simulator (external). (c) Synthesizer (internal).

law with parameter $p = 1/8$: $P(W >= x) = (1-p)^{x-2}$. This leaves a 2% chance for cells of at least 32 bits, while two thirds of the widths will be below 10 bits.

Verilog backend. EDA software generally operates on Verilog sources, not on netlists of macrocells. To output networks of macrocells, TRANSFUZZ relies on the Yosys Verilog backend, which is a mature and well-tested implementation of a Verilog netlist generator. Hence, TRANSFUZZ's input generator, implemented in Python, produces a serialized netlist. A TRANSFUZZ pass in Yosys, written in C++, then translates this description into a Yosys's internal representation. The Yosys backend eventually produces a Verilog source.

5 Differential Fuzzing for Bug Detection

EDA software does not generally have a formal specification or a golden model. Additionally, translation bugs usually do not produce obvious signals like error messages or crashes.

Naively, one could observe all intermediate values of a given design in space and time, e.g., by instrumenting it with many probes, or by enabling tracing. However, this approach has two drawbacks. First, monitoring all values is expensive in terms of time and memory, while fuzzing is generally performance-sensitive. Second, intermediate acquisitions may prevent some optimizations by EDA applications and may hence prevent triggering certain bugs. To address this challenge, we propose to employ differential fuzzing [35, 65, 76] to detect bugs in EDA software. Differential fuzzing requires the selection of variants for fuzzing and a way to compare these variants.

5.1 Enabling differential fuzzing

Variant selection. We find that differential fuzzing (DF) can be applied in two ways. In *internal* DF, the test cases are executed differentially between two parameter settings of the same application, like optimizations or tracing, while in *external* DF, two distinct applications are used, as illustrated in Figure 7. TRANSFUZZ uses external DF for the simulators and internal DF for the synthesizer.

Note that differential fuzzing does not, per se, specify which variant is incorrect. The most straightforward and classical solution is majority voting, yet the majority could be wrong, especially when it comes to misinterpretations of the Verilog standard. Another approach is to disable all optimizations and see how the behavior evolves. We could attribute all the bugs found by TRANSFUZZ and listed in Section 6.5 using a combination of these two methods.

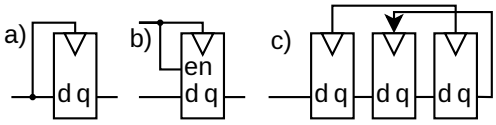

Figure 8: Examples of race conditions. a) Incoming data and clock are connected. b) Enable and clock signals are connected. c) The input and clock of the central register change simultaneously.

Cumulative signature. Differential fuzzing requires the ability to compare variants' outputs. TRANSFUZZ relies on a few *explicit* hardware output signals that it cumulates over the required number of cycles, resulting in a *cumulative signature* that can then be compared across variants. For assisting the propagation of unexpected values, we bias the macrocell input selection towards macrocell outputs that are not yet used. This reduces occurrences of cases where a bug is not observed due to the absence of a path between an erroneous value and the output.

5.2 Ensuring consistency

The Verilog standard offers slack on some aspects, such as the ordering of some events, or the precision of X propagation. Hence, even when simulating the same RTL, legitimate divergences between two simulators may exist. To avoid false positives when comparing variant outputs, we must slightly constrain test-case generation as discussed next.

Avoiding X propagation. Verilator and CXXRTL do not explicitly support X (undefined value) propagation, yet Icarus does. Additionally, simulators may support X propagation with various levels of precision [58, 73, 95, 98]. We avoid the generation of X values by using constructs that do not generate X values and constraining the input signals to certain operators. Appendix C provides additional details on how TRANSFUZZ precisely prevents X propagation in the generated circuits.

Unordered asynchronous events. Asynchronous signals could cause race conditions that would impair cross-EDA software consistency. Figure 8 summarizes some examples of race conditions. They have in common that multiple input ports of some register can simultaneously toggle. In such cases, like for eventual hardware implementations, the Verilog standard does not specify the event ordering.

To overcome the race conditions in a generic way, we impose the following safety invariant on the test case hardware: two inputs of a register will never toggle at the same time. We

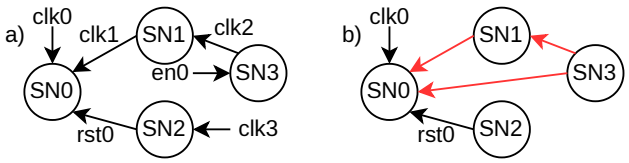

Figure 9: Examples of (a) valid and (b) invalid (race condition prone) network of subnets (SN). The input and output words of each subnet are omitted in the figure.

guarantee this invariant at a subnet level, by distinguishing two wire types. *Asynchronous* wires belong to the sensitivity list of some stateful cell. The others are called *synchronous* wires. We assimilate synchronous wires to data signals produced inside the same subnet, and asynchronous wires to data signals produced outside the subnet, as illustrated in Figure 9 (a). Concretely, when TRANSFUZZ generates a subnet, this subnet creates a set of requests for asynchronous wire connections (e.g., clock, reset or enable signal). Each of these requested asynchronous values is produced either by another subnet, or by a new asynchronous input wire.

To guarantee the absence of asynchronous race conditions, TRANSFUZZ imposes that the network of subnets, seen as an undirected graph, must be acyclic, and that the set of fanout subnets of the set of toggling asynchronous signals is always disjoint. Figure 9 (b) illustrates a violation of this constraint.

Summary. TRANSFUZZ operates at the level of cell netlists to satisfy the requirements formulated in Section 4.1. TRANSFUZZ ensures that designs are valid by constructing sound subnets and allows splitting and merging wires to maximize diversity. Furthermore, differential fuzzing allows the detection of translation bugs, at the condition that all legitimate divergences between EDA applications are alleviated. TRANSFUZZ achieves this by preventing X propagation and unordered events. TRANSFUZZ then uses cumulative signatures to compare the outputs of different EDA applications. Cumulative signature mismatches indicate translation bugs.

6 Evaluation

We evaluate TRANSFUZZ in terms of raw performance (Section 6.1) and cell output coverage (Section 6.2). From these measurements, we deduce an optimal duration for the stimuli after which the circuit must be regenerated (Section 6.3), and find circuit sizes for optimal differential fuzzing performance (Section 6.4). We then describe the 31 new bugs found by TRANSFUZZ in Verilator, Icarus Verilog, CXXRTL and Yosys and evaluate the time to find each one of them (Section 6.5). We finally compare TRANSFUZZ with the state-of-the-art in synthesizer testing [48] (Section 6.6).

Evaluation setting. We obtain the performance results on a machine equipped with two AMD EPYC 7H12 processors at 2.6 GHz with 256 logical cores and 1 TB of DRAM.

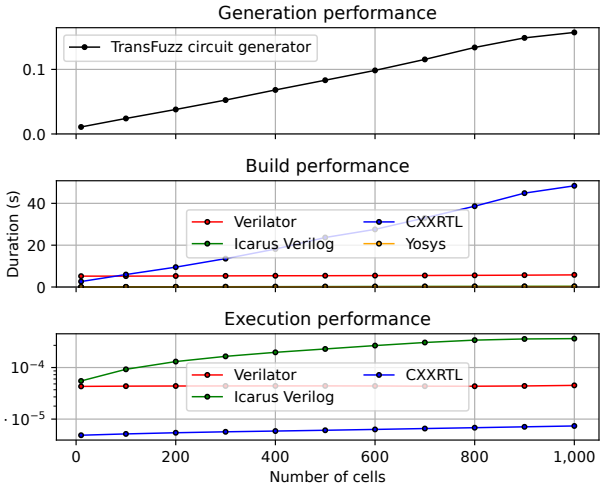

Figure 10: Raw performance evaluation. Generation represents the construction of the test case by TRANSFUZZ. Build represents the process that transforms TRANSFUZZ's test case representation until the simulation model. Note the logarithmic scale on the execution plot, which represents a single stimulus execution (averaged over 1000 stimuli).

For measuring performance, we use the following tool versions: `Verilator 5.021 g6b8531f0a`, `Icarus Verilog g77d7f0b8f` and `Yosys/CXXRTL 0.37+21 3d9e44d18` with all optimizations enabled and tracing disabled by default. TRANSFUZZ is implemented as roughly 5000 lines of Python and 1000 lines of C++ code.

6.1 Raw performance

We evaluate the performance of TRANSFUZZ in terms of generation, elaboration and execution of test cases.

Methodology. To evaluate the performance of the test case generation and build, we generate 1000 test cases for each circuit size and average the resulting performance. To measure the execution time per stimulus, we measure the average time to execute 1000 stimuli on each of the 1000 test cases and obtain an average execution time per stimulus.

Results. We summarize the results in Figure 10. The generation time is generally small compared to the build time. Different simulators have vastly different behaviors. Yosys and Icarus Verilog do not build an executable simulation model (Yosys is a synthesizer and Icarus is an interpreter), hence yield faster build times. Verilator's build time is remarkably flat over circuit sizes. Note that this is not a general benchmark of the three simulators, as the inputs are not typical simulator inputs. We will show how these results impact the overall performance of TRANSFUZZ in Section 6.3.

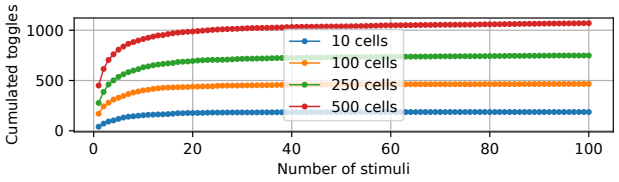

Figure 11: Cumulated toggle coverage of cell outputs.

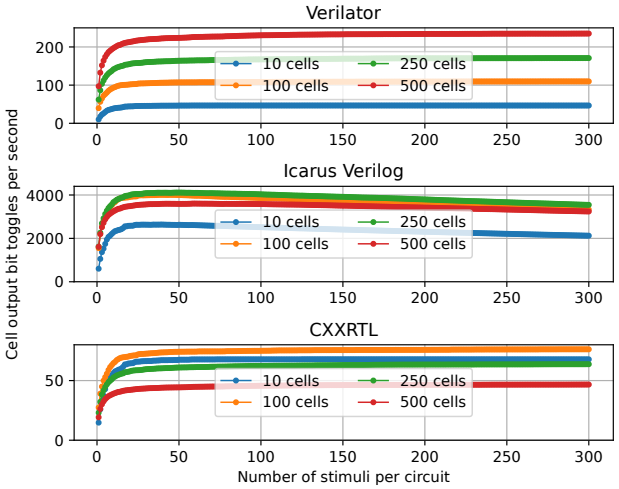

Figure 12: Toggle performance for each simulator in function of simulation length and circuit size.

6.2 Cell output toggling

Intuitively, fuzzing the same circuit with more stimuli brings diminishing returns as increasingly fewer new cell behaviors will be explored over time.

Methodology. We measure the effectiveness of inputs over time by measuring the bit toggles at the cells' outputs. We use this metric to reflect the different behaviors of the cells being explored. This measurement is independent of the target EDA software. We execute 1000 test cases for each circuit size over 1000 stimuli and measure the average cell output toggle coverage by analyzing the execution traces over time.

Results. We summarize the results in Figure 11. The considered toggle coverage increases rapidly for the first tens of stimuli, and then the rate of progress decreases. This observation complies with the intuition of diminishing coverage returns for increasing stimuli lengths. The increase in coverage past 100 stimuli, not illustrated in the figure, is small. From 100 to 1000 stimuli, the coverage increases by less than 6% for all circuit sizes. Toggle coverage also increases sublinearly with the circuit size. Next, we combine this coverage with raw performance to converge on the goodput of TRANSFUZZ over arbitrary stimuli and circuit sizes.

6.3 Stimuli's length

We intend to set the parameters of TRANSFUZZ in a region that will maximize its efficiency. Under the hypothesis that strong toggle coverage of cell outputs indicates effective fuzzing, we want to maximize the number of cell output toggles per second to maximize the effective performance of the fuzzer. The first question that we address is how many stimuli to execute per circuit. Concretely, given the diminishing returns of the stimuli shown in Section 6.2, we expect that after a certain number of stimuli, it will be more advantageous to start a new circuit by paying the fixed generation and build costs again, instead of fuzzing the same circuit further.

Methodology. For each simulator, we calculate the average number of cell output toggles achievable per second for a given circuit size and stimuli duration. For a fixed circuit size, this value is given by Equation 1, where `prep_time` is the average time to produce a circuit of a given size, i.e., generation and build times, and `exec_time` is the average time to execute a stimulus for this size, as measured in Section 6.1, and `cumul_toggles` is illustrated in Figure 11. The simulation length (`simlen`) that maximizes the toggle performance is the number of stimuli after which to regenerate a fresh circuit.

$$\text{toggles}_{/sec}(\text{simlen}) = \frac{\text{cumul\_toggles}(\text{simlen})}{\text{prep\_time} + \text{exec\_time} \cdot \text{simlen}} \quad (1)$$

Results. We summarize the results in Figure 12. Icarus Verilog's performance declines after tens of cells. The other simulators' performance stabilizes at this point to increase by less than 5% to reach peaks for simulation lengths between 124 and 2420 stimuli before decreasing. We make three observations. First, always renewing the test case after a single stimulus would be inefficient. Second, the diversity in the raw performance detailed in Section 6.1 translates into diversity in the toggle performance. Icarus Verilog, whose marginal execution cost per stimulus is the highest, meets its peak throughput between 38 (10 cells) and 52 (250 cells) stimuli per circuit, and is the most efficient to fuzz in absolute numbers. Verilator is the most efficient when it comes to larger circuits because of its flat performance plot but high build cost for small circuits. Finally, after around 20 stimuli, the performance is remarkably stable. We choose a simulation length of 70, which is in bounds with 95% of the peak performance in all configurations. With optimal stimuli size per simulator known for different circuit sizes, we measure the optimal selection of circuit sizes for differential fuzzing.

6.4 Circuit size for differential fuzzing

When differentially fuzzing two simulators, we must choose a circuit size that maximizes the global performance given the simulators that are involved.

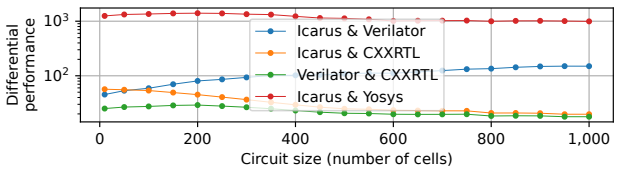

Figure 13: Differential toggle performance. Maximizing this abstract metric maximizes the performance (cell output toggles per second) for differential fuzzing.

Methodology. We aim to maximize the metric $m = (P_a \cdot P_b)/(P_a + P_b)$, where P_a and P_b are the respective toggle performances. We hence measure m for different circuit sizes and 70 stimuli per circuit (as measured in Section 6.3).

Results. Figure 13 shows the performance of differential fuzzing for each pair of simulators. These results show that we should fuzz Verilator against Icarus Verilog for circuits of 800 to 1000 cells, Icarus against CXXRTL for 100 cells or less, and Yosys against Icarus for approximately 200 cells. Interestingly, this advocates for separate fuzzing circuit sizes for each differential pair, instead of fuzzing all simulators alike. To the best of our knowledge, in the domain of differential fuzzing, this is a unique finding.

6.5 Discovered bugs

TRANSFUZZ discovered 31 new bugs in 4 popular EDA applications. Table 2 summarizes them and highlights the 20 discovered translation bugs. We first analyze the new bugs, then evaluate TRANSFUZZ's performance in finding them.

6.5.1 Bug descriptions

Translation bugs. TRANSFUZZ found translation bugs in all four tested EDA applications. Some occur in particularly complex cases. For example, I2 is the first reported translation bug in Icarus for at least 3 years, corresponding to the last 100 relevant bug reports analyzed in Section 4.1. The mistranslation I2 causes some arithmetic operations to produce wrong outputs when supplied with inputs which are specific patterns of concatenations of specific constant bits and X (unknown) bits. With the multiple concatenations, this bug particularly benefits from TRANSFUZZ's netlist-level abstraction, which eases fractioning and concatenating wires.

Verilator and CXXRTL are particularly affected by translation bugs, potentially because of more aggressive optimizations. We observe, indeed, their better marginal runtime execution performance in Section 6.1. In total, TRANSFUZZ found respectively 9 and 6 translation bugs in Verilator and CXXRTL, involving a large diversity of cell types (operational diversity) and interconnection patterns (relational diversity). Fixing these bugs has sometimes been challenging. For example, V10 and V12 are similar in their expression, as they can both be expressed through exponentiation cells. Yet they affect different internal structures, hence after fixing V10, V12 was still detected by TRANSFUZZ, enabling the eventual of fix this bug regarding representations of wide numbers.

Table 2: Bug reports for Icarus Verilog (I), CXXRTL (C), Verilator (V) and Yosys (Y). Translation bug IDs are underlined.

Id	Bug Description
I1	Segfault when using out-of-scope variable in slices
I2	Arithmetics deviation in specific circumstances
I3	Performance issue for a design with xor reductions
C1	Shifts consider signed operand in some cases
C2	Bad assignment under specific conditions
C3	Bug with modulo when operands intersect
C4	Both left shifts sometimes overflow the output signal
C5	Incorrect division
C6	Clock edges sometimes require 2 evaluations
C7	Performance issue with many concatenations
C8	Elaboration fails for some stateful cells in some cases
V1	Evasive malloc failure in some instances
V2	Segfault during evaluation
V3	Segfault in traceInit
V4	Segfault with many parallel operators
V5	Segfault in Vtop___024root__trace_init_sub__TOP__0
V6	Wrong not when checking (n)eq under and/or tree
V7	Wrong simulation result with add and xor gates
V8	Optimization error for 5 optimization types
V9	Sometimes wrong conversion to 32-bit integers
V10	Under some conditions, 0 power 0 gives 0
V11	Evasive compilation-sensitive mis-simulation
V12	Pow operator supplied with wide constants
V13	Incorrect bit-op-tree not optimization
V14	Bit-op-tree should not touch some subtrees
V15	Incorrect widthMin in replaceShiftOp
V16	Ignore if eq/ne is under shiftr
V17	VCD corruption for 5 optimization types
V18	Compiler sees empty input due to file system races
Y1	`opt_muxtree` wrong if twice the same mux
Y2	Misoptimization of wide shifts

The two translation bugs Y1 and Y2 found in Yosys concern uncommon cell interconnections and internal representations. Y1 arises because of redundant multiplexers in a specific configuration involving signal concatenations ahead. In that case, Y1 mistranslates one of these concatenations by replacing one input with a constant value, while the input is free to toggle. Somehow similar to V10, Y2 misrepresents wide constants in shift operations.

Other bugs. TRANSFUZZ found a segmentation fault in Icarus (I1) and 5 in Verilator (V1-V5). It additionally found exceptions in CXXRTL (C8) and Verilator (V18), as well an issue that eventually causes tracing issues (V17). Finally, TRANSFUZZ found two confirmed performance bugs (C7 and I3). The former is sensitive to circuit depth and would require splitting operations into smaller pieces for speeding up. The latter happens with concatenations of many inputs, which,

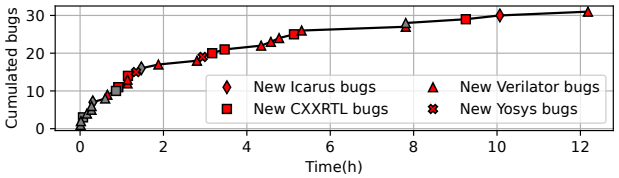

Figure 14: Cumulated time to bug. Translation bugs are colored in red.

themselves, share bits. The evaluation time was exponential in the number of shared bits. This bug I3 has been fixed by deferring their evaluation to the end of the evaluation call.

CVEs. Given the security relevance of the discovered bugs, 25 CVEs were assigned, as listed in Table 4 of Appendix D.

6.5.2 Time to bug discovery

We fuzz each pair Verilator/Icarus Verilog, Icarus Verilog/CXXRTL and Icarus Verilog/Yosys on 128 processes for 24 hours and summarize the time to discover each bug in Figure 14. We note that generally, translation bugs require more time to be discovered than other bugs.

6.6 Comparison with Verismith

We now compare TRANSFUZZ with Verismith, the state-of-the-art synthesizer tester [48]. Verismith generates hardware designs at the HDL level, as opposed to TRANSFUZZ, which generates cell-level netlists. Another fundamental difference between the two approaches is that Verismith uses formal verification to check the equivalence of the generated designs, while TRANSFUZZ fuzzes them in simulation to detect bugs. We use the Verismith open-source tool to generate circuits and verify their equivalence using Yosys.

Performance. Comparing the performance of the two tools is delicate as they offer slightly different guarantees. TRANSFUZZ complies with the heuristic that testing more circuits, each with relaxed guarantees, will yield more bugs. On the other hand, Verismith proves the equivalence of a design with the synthesized output. However, we measure that 46% of the Verismith tests time out, hence Verismith provides no guarantee for these circuits. Note that the timeouts are necessary in Verismith as our experiments show that individual proofs would often not terminate even in 48 hours.

To compare the performance of the two approaches, we test Yosys with Verismith and with TRANSFUZZ. We measure the single-threaded time for testing 100 input designs provided by both TRANSFUZZ and Verismith, and normalize this duration by the total number of tested cells reported by Yosys. As a result, TRANSFUZZ spent 2.6 ms per cell, while Verismith spent 604 ms per cell on average. Hence, the number of cells tested per second by TRANSFUZZ is 232× higher. Note that this raw performance comparison is not a general benchmark, as the guarantees over a given input design are different.

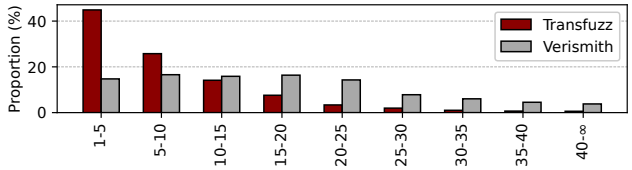

Figure 15: Distribution of cell widths generated by TRANSFUZZ and Verismith.

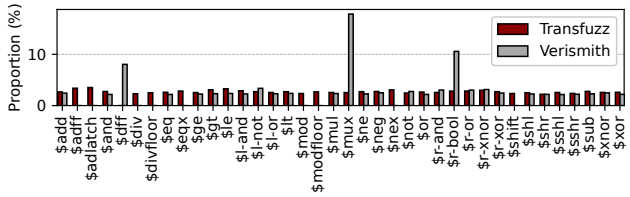

Figure 16: Cell types generated by TRANSFUZZ and Verismith. "r-" or "l-" are respectively reduction and logic cells.

Memory usage. Our experiments show that a single Verismith instance can exceed 64 GB of resident memory usage for equivalence testing which is a significant barrier for parallel testing. On the other hand, TRANSFUZZ uses less than 1 GB of resident memory per process, enabling it to to saturate the server's computing resources.

Cell diversity. Optimizations usually happen at the intermediate representation level made of cells [103], as observed in Section 4.1 and Section 6.5, hence the operational diversity is a key factor in finding bugs in EDA tools. However, operating at the HDL level impairs controlling exact properties of the cells that will compose the design, may it be widths or types.

Figure 15 shows the distribution of cell widths. Many bugs are reproducible with narrow cells, as observed in Section 4.1 and Section 6.5, yet Verismith does not seem to leverage this observation. Figure 16 shows the distribution of cell types generated by TRANSFUZZ and Verismith. We observe that Verismith-generated designs are composed by a large proportion of multiplexers and boolean reductions to the detriment of operational diversity as discussed in Section 4.1.

Generated design structure. Verismith is implemented in Haskell and generates designs at the HDL level. We observe its bias towards generating tree-like expressions, which reduces the diversity of the operator interconnections, to the detriment of relational diversity as discussed in Section 4.1.

In comparison, the netlist approach idiomatically produces diverse interconnection patterns. In particular, TRANSFUZZ's netlist approach tends to often slice and recombine non-trivial wire and register segments. This has been critical for finding bugs in EDA applications, as slices and concatenations were involved in bugs I1, I2, C2, C3, C7, V13, V16, Y1 and Y2. Such constructs are rare in Verismith since they are complex to express at the HDL level.

Target scope. Limited by the formal equivalence aspect, Verismith can only find bugs in synthesis tools. While Veri-

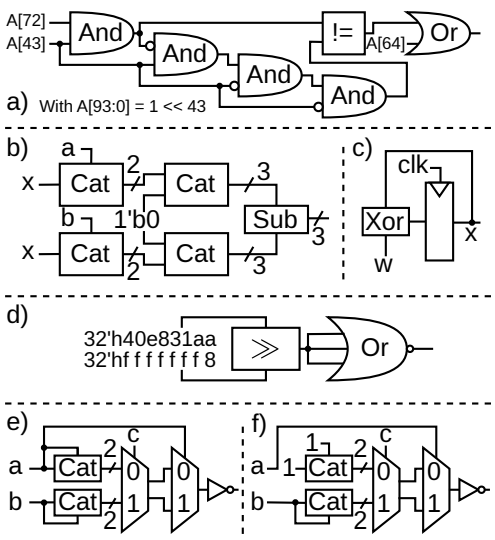

Figure 17: MIRTL gadgets. `Cat` are concatenations. (a) Verilator gadget (V6). (b) Icarus Verilog gadget (I2). (c) Primitive for X injection, useful for the Icarus gadget. (d) CXXRTL gadget (C1). (e) Yosys gadget (Y1). (f) Yosys gadget once mistranslated.

smith reports 1 issue in the Icarus Verilog simulator [47], this bug is reported to have been found coincidentally during the development of Verismith. Hence, Verismith could, at best, find two out of the 31 new bugs discovered by TRANSFUZZ. In practice, in 24h of testing, Verismith did not find any of the new bugs discovered by TRANSFUZZ.

7 End-to-End Attack

To enable the exploitation of translation bugs, we introduce MIRTL gadgets, which are primitives that produce predictable inverted values when processed by a targeted EDA application. We then show how MIRTL gadgets complement traditional hiding techniques to create stealthy hardware trojans by bypassing classical white-box RTL trojan detection techniques. For example, we use MIRTL gadgets to introduce a kernel data leakage trojan that exposes supervisor data to user-mode processes in the CVA6 CPU in a manner that is undetectable in all three simulators in our study.

7.1 MIRTL gadgets

We design MIRTL gadgets to produce `1` under normal circumstances, and `0` when exploiting a translation bug. First, such gadgets must be non-intrusive, i.e., must not alter the functionality of the design under normal circumstances. Second, they must have a negligible impact on area, power and timing. Third, they should not trigger warnings.

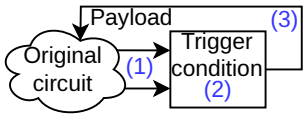

Figure 18: Structure of a classical RTL trojan and potential insertion points of a MIRTL gadget.

Individual simulator gadgets. From the bugs discovered by TRANSFUZZ, we construct a suitable gadget for each simulator illustrated in Figure 17 (a)-(d), respectively building on the translation bugs V6, I2, C1.

Compatibility. To build a gadget compatible with all three simulators, we connect the three gadgets as the three inputs of a new 3-input `and`. This way, the gadget will produce the mistranslated `0` value if any of the three simulators is used.

Synthesizer gadget. Figure 17 shows a gadget for the Yosys synthesizer (e), and how it is mistranslated during the default `opt_muxtree` Yosys optimization pass (f). By setting `a=c=1'b0`, the gadget will mistakenly produce an output value of `0` in the synthesized version of the design.

Head register. MIRTL gadgets take specific input constants. Yet, these constants cannot be hardcoded directly in the design, as gadgets usually require them to be unknown at elaboration or synthesis time. Instead, we find that inserting a register between the input constant and the gadget is sufficient to make the gadget work in all cases, as the EDA software does not see a fixed value. This register takes X as the initial default value and uses the design's global clock.

Tail register. It may happen that the end of the gadget gets optimized together with this logic before the MIRTL trait appears, which would disarm the gadget. Another issue of the following logic is that together with the gadget, it may form a critical path. Therefore, we terminate the gadget with a flip-flop to mitigate these side effects. This flip-flop is also connected to a global clock. While tail registers were compatible with all the translation bugs that we tested, there is no guarantee that they will never disarm a gadget, hence the gadgets must always be tested before use.

7.2 Empowering RTL trojans with MIRTL

Classical RTL trojans can be very stealthy once inserted into a design, but they are subject to white-box detection, which can reveal RTL trojans by not requiring blindly guessing their arbitrarily specific triggering conditions. We show how MIRTL gadgets harden classical RTL trojans against white-box detection techniques by erasing the trojan during translation, before the verification operations happen.

Classical RTL trojans. Classical RTL trojans are generally made of a trigger condition and a payload [15, 96] which both connect to the original design, as illustrated in Figure 18. The payload is activated only when the trigger condition is

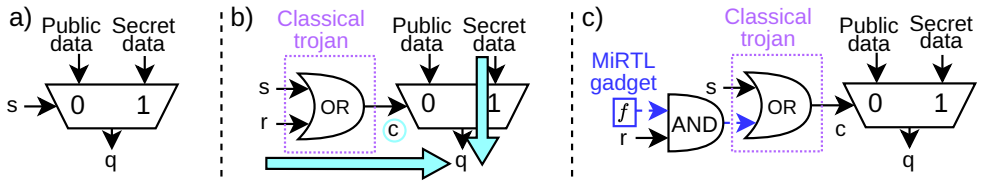

Figure 19: Example classical RTL trojan and MIRTL hardening in access control logic. The signal s represents whether the design is currently operating in high-privilege mode. a) Original logic. b) Classical trojan inserted, where r is some rare attacker-controllable condition on the design's state or inputs. c) MIRTL gadget insertion to prevent white-box detection.

met, which can be an arbitrarily specific condition on the design's state or inputs. This condition is typically designed to be easy to reach if known, but very hard to guess otherwise. For example, it might be based on specific values in some specific registers. The payload then tampers with the design's behavior, for example by allowing unauthorized access to privileged resources or data, or by denying some functionality.

While blindly guessing the trigger condition is an arbitrarily difficult task, extensive previous research has shown that white-box techniques can detect the trojan by multiple ways without requiring guessing the trigger. First, SAT solving [9, 42, 55, 63, 74, 77] or symbolic execution [1, 29, 36, 84] can assert the correctness of the affected functionality, which will result in the functionality being proven incorrect when the trojan is active. Second, the triggering condition is generally based on design state and inputs that should not influence the affected functionality [7–9, 52, 61, 80]. Hence, this creates an information flow from these points to the payload anchor. Previous research has shown that such breaches can be detected by standard information flow techniques [2, 50, 67, 69, 93, 99]. Finally, the rise in hardware fuzzing that involves various coverage metrics [16, 53, 57, 59, 97, 104] will also vastly improve the trigger guessing capabilities in RTL simulation.

MIRTL-based hardening. MIRTL gadgets can prevent white-box detection techniques from detecting RTL trojans. The key point is to insert MIRTL gadgets so that they cancel the trojan's effect. In practice, this offers a large flexibility in the insertion location of MIRTL gadgets. As illustrated in Figure 18, the MIRTL gadget can be inserted at any point in the RTL trojan, may it be in its design state sampling (1), in its trigger condition (2), or in its payload (3). In any of these cases, the MIRTL gadget must cancel the trojan's effect completely. Practically, MIRTL gadgets are best inserted along with an `and` gate that will filter out a signal critical to the trojan to deactivate it.

A concrete trojan example. Figure 19 (a) illustrates a simple example access control circuit, and Figure 19 (b) the insertion of a classical trojan in this circuit. The access logic is designed to prevent user-mode software from accessing kernel memory, and the trojan is inserted to allow such access under some specific condition r that is extremely rare in normal operation, but that the attacker can easily control.

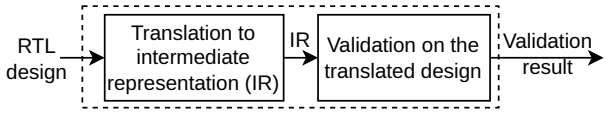

Figure 20: Internal operation of a white-box validation EDA application. Mistranslation gadgets affect the translation front-end, hence are agnostic of the applied validation technique.

In a black-box setting, detecting the trojan is as hard as guessing the condition r, but the aforementioned white-box techniques [2, 7–9, 16, 50, 52, 53, 57, 59, 61, 67, 69, 80, 93, 97, 99, 104] can detect it. First, one can formally ensure that the condition c (circled in cyan) that controls the multiplexer is never true when the design is in user mode by constraining the privilege level signal s to zero and proving that the control input c is always zero, by the means of SAT solving or symbolic execution. Second, information flow tracking, when tainting secret data and proving, under the constraint that the privilege s is zero, proves whether taint can reach the output q of the multiplexer (vertical cyan arrow). Information flow tracking can also be used to taint all design states that do not have anything to do with the access control condition (horizontal cyan arrow). Given that r is a condition on these states that should not directly influence the access control such as the values in registers, taint flowing from these registers to the control condition, and hence to the output of the multiplexer will reveal a confidentiality breach. Third, simulation-based white-box techniques such as coverage-based fuzzing might jeopardize the trojan. In particular, covering the multiplexer control [59], control registers [53] or toggling wires [16, 57] will eventually trigger the trojan.

Figure 19 (c) shows the insertion of a MIRTL gadget that, when mistranslated, filters the condition r out to prevent validation methods from detecting the RTL trojan. Any EDA software affected by this MIRTL gadget would see the trojan as provably always inactive and would then successfully demonstrate or prove the absence of any unexpected behavior, as it is hidden under cover of the MIRTL gadget.

Affected EDA software. All white-box techniques affected by translation bugs can be confused into analyzing a falsified version of the design that does not contain the trojan. The key, as shown in Figure 20, is that MIRTL gadgets affect the translation front-end of the EDA application that is used

Listing 1: User-accessible bit check in CVA6's MMU.

```
daccess_err = en_ld_st_translation_i && ((
    ld_st_priv_lvl_i == S && !sum_i &&
    dtlb_pte_q.u) || (ld_st_priv_lvl_i == U
    && !dtlb_pte_q.u));
```

for validation, hence the hiding is agnostic of the white-box validation technique. Experimentally, we confirm, for example, that when the MiRTL gadget based on Y1 is inserted, Yosys-based SAT solving that aims at proving the integrity of c in Figure 19 (c) fails to detect the trojan, and that the information flow tracking logic generated by the state-of-the-art information flow tracking tool CellIFT [89, 91], confused by the MiRTL gadget, mistakenly proves the absence of the trojan.

7.3 Hiding a trojan in CVA6 wtih MiRTL

To show the practicality of MiRTL gadgets, we inject a trojan into the CVA6 RISC-V CPU [108], also known as Ariane, that allows an attacker in user mode to read the content of kernel memory (CWE-118). Such a gadget can be inserted by a contributor with malicious intents, or by a compromised version an intermediate RTL flow tool such as sv2v [85].

Background. In the Sv39 virtual memory system specified by RISC-V and supported by CVA6, the bottom-most bits of a page table entry are, in increasing index order, V: valid, R: readable, W: writable, X: executable, U: user accessible. When virtual memory is enabled, user software attempting to access a page with $U = 0$ will trigger an exception.

7.3.1 Strategy

Classical trojan. We aim to alter the functionality of CVA6 to allow user software to load data from a page with attribute $U = 0$ if, arbitrarily, the operand of the last `FMV.X.D` instruction was `0x12345678badcable`. We then harden this trojan with a MiRTL gadget to prevent white-box detection.

Gadget insertion location. We insert the MiRTL gadget at the location where the U bit is checked. We find that this check is located in the `i_cva6_mmu` module instance in the load-store unit. We report the exact check operation in Listing 1 when it comes to data accesses. We propose to `and` the gadget output bit together with the `!dtlb_pte_q.u` signal and with the test whether the operand of the last `FMV.X.D` instruction was `0x12345678badcable`. Hence, the RTL trojan will be activable only if the MiRTL gadget produces a `1`.

Scenarios. We introduce two exploitation scenarios. In the first, the adversary compromises the design immediately but relies on the combined simulator gadget to hide the bug from all three simulators. In the second, the attacker submits a design that contains no vulnerability, but relies on the synthesizer gadget to introduce the vulnerability at synthesis time.

Table 3: Design metric deviations caused by gadget insertion.

	Area	Power	Registers	Timing
Scenario 1	+0.1%	+0.07%	+0.02%	+0%
Scenario 2	+0%	+0%	+0%	+0%

In both cases, the eventually-synthesized design is vulnerable, which is invisible or absent in the preceding RTL.

7.3.2 Evaluation

We first evaluate the functional correctness of the exploit. We then evaluate its impact in terms of area, power and timing.

Functional correctness. To validate the correctness of both scenarios, we design a simple compliance test including a basic operating system taken from the RISC-V compliance test suite [87], where we integrate a user application attempting to read from a page with $U = 0$ after executing `FMV.X.D` with the right operand. We observe that the simulators do not see access exceptions when MiRTL gadgets are inserted. By successfully executing the RISC-V compliance tests [87], we ascertain that the overall CPU functionality is unaffected.

Area, power and timing. To evaluate the impact of the exploit on design metrics, we synthesize Yosys's output in a popular 12 nm technology using Synopsys Design Compiler 2022.03, targeting a 1 GHz clock. Table 3 summarizes the deviations with the unaffected CVA6 design. In particular, any variation in timing would be problematic for the attacker, as it would make the gadget obvious. We observe that the evaluated MiRTL exploits negligibly impact all metrics.

Stealthiness. It is beneficial for gadgets to be discreet in front of code inspection. Typical gadgets require around 10 lines of Verilog code. While this is few lines, they can be intertwined into the design sources on a case-by-case basis to look innocuous. If the design is programmed in a hardware construction language (HCL) [5, 24, 79, 105] (then automatically compiled into Verilog), a gadget inserted into the Verilog representation, e.g., through a malicious HCL compiler, is unlikely to be obvious in the complex Verilog output.

8 Discussion

Mitigations. A straightforward solution against MiRTL attacks is to sure that EDA software remains free of bugs. We encourage further fuzzing work against EDA software, yet it is still unclear how to guarantee bug-freedom of simulators. Furthermore, we observe that many of the bugs rely on operational diversity. Hence, reducing the design to the gate level early in the development stages would mitigate some of the bugs, but this is known to incur significant performance loss [2, 89]. While rarer, translation vulnerabilities could also occur at gate level. Finally, one could check the equivalence of the design before and after synthesis [75], but this approach has two shortcomings. First, it has challenging

scalability issues as we showed in Section 6.6. Second, analysis approaches that rely on EDA software themselves, such as DIFT, remain vulnerable to MIRTL attacks, even if they offer formal guarantees as we demonstrated in Section 7.2. Finally, manual inspection, with known limitations [32], will be mostly ineffective against MIRTL attacks. Synthesizer MIRTL gadgets describe valid functionality and hence can only be detected by their potential shape in RTL until they are processed by the victim synthesizer. Worse, if a MIRTL gadget is inserted by some corrupted EDA preparation software like bender [25] or sv2v [85], then humans must inspect the output of these tools, which is prohibitively hard to read, and already contains many complex patterns, hampering the identification of suspicious potential gadget patterns.

Syntactic limitations. TRANSFUZZ uses the Yosys Verilog backend to generate the final design representation. This backend does not produce behavioral blocks, and has its own flexible output structure. Hence, TRANSFUZZ is currently unable to find some syntax-dependent bugs. A dedicated backend would overcome this limitation. Currently, the bugs found by TRANSFUZZ are syntactically simple, simplifying exploitation as we showed in Section 7. It is unclear whether practical exploitation would be possible if the bug requires a more complex syntax. For example, compromised EDA preparation software like sv2v cannot inject complex syntax, as its goal is generally to reduce the language spectrum.

FPGA testing for ASIC validation. FPGA testing is generally powerless against MIRTL attacks. First, this is because classical hiding techniques can be combined with MIRTL gadgets as demonstrated in Section 7.2. This also renders post-silicon detection ineffective. Given that many natural bugs already escape validation in commodity hardware as of today [90], voluntarily hidden bugs would likely be even harder to detect than most natural bugs. Second, FPGA and ASIC flows are often completely distinct, from the synthesis on. Let us assume that an attacker wants to insert a bug in the ASIC design but not in the FPGA design. In a scenario where simulator bugs are exploited to hide a vulnerability, a translation bug in the FPGA synthesis tool (e.g., Xilinx Vivado [21]) could be exploited to remove the vulnerability in the FPGA target. In another scenario where a translation bug in an ASIC synthesis tool (e.g., Synopsys DC [23]) would be exploited for injecting the bug, the bug would only be injected into the ASIC implementation and not into the FPGA.

Open-source EDA. In this paper, we focused on open-source EDA software, which is increasingly popular. For example, on the ASIC fabrication side, Yosys is used, among others, by qflow [26] and the OpenLANE flow [83], whose results are becoming increasingly competitive with commercial flows [49]. On the FPGA side, Yosys is used by popular flows such as SymbiFlow [66] and PRGA [60]. On the simulation side, open-source simulators are nowadays used for complex SoCs [20] and diverse designs such as BLE MAC [43].

9 Related Work

We first discuss compiler fuzzing. We then cover other work that aims at compromising pre-silicon hardware designs.

Compiler fuzzing and bugs. Compilers share similarities with synthesizers. The confused deputy problem was initially demonstrated in a compiler and some backdoors were demonstrated using some of their bugs [6, 22, 31, 44]. Previous work advertises multi-variant execution against such attacks [13], but we show the limitation of this proposal by compromising three simulators at a time in Section 7. Compiler fuzzing has been shown to be effective at finding bugs in compilers [18, 54, 62, 64, 68, 106], mostly by relying on differential fuzzing [35, 65, 76]. Yet fuzzing and differential testing differ in the context of EDA applications. We show in Section 6 that inputs of non-trivial lengths boost performance, while compiler outputs are generally executed only once [62, 106]. Additionally, fuzzing with software programs shares no clear similarity with fuzzing with hardware circuits.

Fuzzing simulators. From GitHub issues [27], we observe that there have been simulator fuzzing campaigns with traditional approaches of HDL file content manipulations. While this approach could find crashes in the simulator's frontend, it is unlikely to find bugs in the simulator's backend where the translation bugs may lurk.

10 Conclusion

MIRTL is a new class of confused deputy attacks on EDA software that relies on translation bugs. We presented TRANSFUZZ, the first fuzzer dedicated to finding such translation bugs in simulators and synthesizers using pairs of hardware designs and stimuli. By creating netlists of macrocells, TRANSFUZZ creates particularly complex hardware designs. In addition to some new unsafe crashes, TRANSFUZZ found 20 new translation bugs causing wrong runtime values in three major open-source RTL simulators and in the Yosys synthesizer, among the 31 new bugs that it found. We showed concrete MIRTL attacks that exploit these translation bugs by corrupting the behavior of the CVA6 CPU to leak kernel memory to user mode and bypassing the white-box analysis techniques. Facing these newly demonstrated security risks based on yet undiscovered EDA bugs, we encourage further research in the direction of discovering and fixing bugs in EDA software.

Ethical considerations. We reported all bugs to their respective maintainers and provided support when required.

Acknowledgements

The authors would like to thank the anonymous reviewers and Katharina Ceesay-Seitz for their valuable feedback. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

References

[1] Alif Ahmed, Farimah Farahmandi, Yousef Iskander, and Prabhat Mishra. Scalable hardware trojan activation by interleaving concrete simulation and symbolic execution. In *ITC*, 2018.

[2] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. Register transfer level information flow tracking for provably secure hardware design. In *DATE*, 2017.

[3] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, et al. The rocket chip generator. *UC Berkeley*, 2016.

[4] K. Asanovic, D. A. Patterson, and C. Celio. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. *UC Berkeley*, 2015.

[5] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a scala embedded language. In *DAC*, 2012.

[6] Scott Bauer, Pascal Cuoq, and John Regehr. Deniable backdoors using compiler bugs. https://mcfp.felk.cvut.cz/publicDatasets/pocorgtfo/contents/articles/08-03.pdf. [Online; accessed 6-Feb-2024].

[7] Shivam Bhasin and Francesco Regazzoni. A survey on hardware trojan detection techniques. In *ISCAS*, 2015.

[8] Swarup Bhunia, Michael S Hsiao, Mainak Banga, and Seetharam Narasimhan. Hardware trojan attacks: Threat analysis and countermeasures. *Proceedings of the IEEE*, 2014.

[9] Swarup Bhunia and M Tehranipoor. The hardware trojan war. *Springer*, 2018.

[10] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz. Grimoire: Synthesizing structure while fuzzing. In *USENIX Sec*, 2019.

[11] Jérémy Bonvoisin, Robert Mies, Jean-François Boujut, and Rainer Stark. What is the "source" of open source hardware? *Journal of Open Hardware*, 2017.

[12] Jérémy Bonvoisin, Jenny Molloy, Martin Häuer, and Tobias Wenzel. Standardisation of practices in open source hardware. *arXiv:2004.07143*, 2020.

[13] Cristian Cadar, Luís Pina, and John Regehr. Multi-version execution defeats a compiler-bug-based backdoor. https://blog.regehr.org/archives/1282. [Online; accessed 6-Feb-2024].

[14] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. B. Taylor, M. Egele, and A. Joshi. Processorfuzz: Processor fuzzing with control and status registers guidance. In *HOST*, 2023.

[15] Rajat Subhra Chakraborty, Seetharam Narasimhan, and Swarup Bhunia. Hardware trojan: Threats and emerging solutions. In *HLDVT*, 2009.

[16] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. Hypfuzz:formal-assisted processor fuzzing. In *USENIX Sec*, 2023.

[17] H. Chen, Y. Li, B. Chen, Y. Xue, and Y. Liu. Fot: A versatile, configurable, extensible fuzzing framework. In *ACM FSE*, 2018.

[18] Junjie Chen and Chenyao Suo. Boosting compiler testing via compiler optimization exploration. *TOSEM*, 2022.

[19] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *IEEE SP*, 2018.

[20] Yuan Chi, Xian Lin, and Xin Zheng. Design of high-performance soc simulation model based on verilator. In *ACAI*, 2022.

[21] Sanjay Churiwala and I Hyderabad. Designing with xilinx® fpgas. In *Circuits & Systems*. 2017.

[22] Tim Clifford, Ilia Shumailov, Yiren Zhao, Ross Anderson, and Robert Mullins. Impnet: Imperceptible and blackbox-undetectable backdoors in compiled neural networks. *arXiv:2210.00108*, 2022.

[23] Synopsys Design Compiler. Synopsys design compiler. *Pages/default. aspx*, 2016.

[24] Clash contributors. Clash: A modern, functional, hardware description language. https://clash-lang.org/. [Online; accessed 25-Jan-2024].

[25] PULP contributors. Bender dependency management tool. https://github.com/pulp-platform/bender. [Online; accessed 16-May-2024].

[26] Qflow contributors. Qflow 1.3: An open-source digital synthesis flow. http://opencircuitdesign.com/qflow/. [Online; accessed 30-January-2024].

[27] Verilator contributors. Fuzzer-related verilator issues. https://github.com/verilator/verilator/issues?q=Fuzzer. [Online; accessed 30-January-2024].

[28] C. Courbet. Nsan: a floating-point numerical sanitizer. In *ACM SIGPLAN CC*, 2021.

[29] Ruochen Dai and Tuba Yavuz. A symbolic approach to detecting hardware trojans triggered by don't care transitions. *ACM Transactions on Design Automation of Electronic Systems*, 28(2), 2022.

[30] Fisher Daniel K and Gould Peter J. Open-source hardware is a low-cost alternative for scientific instrumentation and research. *Modern instrumentation*, 2012.

[31] Baptiste David. How a simple bug in ml compiler could be exploited for backdoors? *arXiv:1811.10851*, 2018.

[32] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. {HardFails}: Insights into {Software-Exploitable} hardware bugs. In *USENIX Sec*, 2019.

[33] LLVM Developers. Undefinedbehaviorsanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. [Online; accessed 4-June-2023].

[34] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. Evolutionary grammar-based fuzzing. In *SSBSE*, 2020.

[35] Robert B Evans and Alberto Savoia. Differential testing: a new approach to change detection. In *ESEC/FSE*, 2007.

[36] Farimah Farahmandi, Yuanwen Huang, and Prabhat Mishra. Trojan localization using symbolic algebra. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017.

[37] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. Afl++ combining incremental steps of fuzzing research. In *WOOT*, 2020.

[38] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *ICSE*, 2009.

[39] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *ASPLOS*, 2008.

[40] Google. American fuzzy lop. https://github.com/google/AFL. [Online; accessed 25-Jan-2024].

[41] Yann Herklotz Grave. Fuzzing verilog. https://yannherklotz.com/docs/fpga2020/verismith_thesis.pdf. [Online; accessed 30-January-2024].

[42] Syed Kamran Haider, Chenglu Jin, Masab Ahmad, Devu Manikantan Shila, Omer Khan, and Marten van Dijk. Hatch: A formal framework of hardware trojan design and detection. *University of Connecticut Cryptology ePrint Archive Technical Report*, 943, 2014.

[43] Eunkyung Ham, Yujin Jeon, Jaeyun Lim, and Ji-Hoon Kim. Verilator-based fast verification methodology for ble mac hardware. In *ICEIC*, 2023.

[44] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 1988.

[45] J Piet Hausberg and Sebastian Spaeth. Why makers make what they make: motivations to contribute to open source hardware development. *R&D Management*, 2020.

[46] ITS Heikkinen, Hele Savin, Jouni Partanen, Jukka Seppälä, and Joshua M Pearce. Towards national policy for open source hardware research: The case of finland. *Technological Forecasting and Social Change*, 2020.

[47] Yann Herklotz. Expression evaluates to 1'bx instead of expected 1'b0. https://github.com/steveicarus/iverilog/issues/283. [Online; accessed 16-May-2024].

[48] Yann Herklotz and John Wickerson. Finding and understanding bugs in fpga synthesis tools. In *FPGA*, 2020.

[49] Sarah Hesham, Mohamed Shalan, M Watheq El-Kharashi, and Mohamed Dessouky. Digital asic implementation of risc-v: Openlane and commercial approaches in comparison. In *MWSCAS*, 2021.

[50] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. Hardware information flow tracking. *CSUR*, 2021.

[51] Wei Hu, Dejun Mu, Jason Oberg, Baolei Mao, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Gate-level information flow tracking for security lattices. *TODAES*, 2014.

[52] Zhao Huang, Quan Wang, Yin Chen, and Xiaohong Jiang. A survey on machine learning against hardware trojan attacks: Recent advances and challenges. *IEEE Access*, 2020.

[53] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *IEEE SP*, 2021.

[54] Raphael Isemann, Cristiano Giuffrida, Herbert Bos, Erik van der Kouwe, and Klaus von Gleissenthall. Don't look ub: Exposing sanitizer-eliding compiler optimizations. *PACMPL*, 2023.

[55] Akira Ito, Rei Ueno, and Naofumi Homma. A formal approach to identifying hardware trojans in cryptographic hardware. In *ISMVL*, 2021.

[56] Shunning Jiang, Peitian Pan, Yanghui Ou, and Christopher Batten. Pymtl3: A python framework for open-source hardware modeling, generation, simulation, and verification. *MICRO*, 2020.

[57] R. Kande, A. Crump, G. Persyn, P. Jauernig, A. R. Sadeghi, A. Tyagi, and J. Rajendran. {TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities. In *USENIX Sec*, 2022.

[58] Christian Krieg, Clifford Wolf, Axel Jantsch, and Tanja Zseby. Toggle mux: How x-optimism can lead to malicious hardware. In *DAC*, 2017.

[59] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen. Rfuzz: Coverage-directed fuzz testing of rtl on fpgas. In *ICCAD*, 2018.

[60] Ang Li and David Wentzlaff. Prga: An open-source framework for building and using custom fpgas. In *The First Workshop on Open-Source Design Automation; Florence, Italy*, 2019.

[61] He Li, Qiang Liu, and Jiliang Zhang. A survey of hardware trojan threat and defense. *Integration*, 2016.

[62] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpgen. *PACMPL*, 2020.

[63] Faiq Khalid Lodhi, Syed Rafay Hasan, Osman Hasan, and Falah Awwad. Formal analysis of macro synchronous micro asychronous pipeline for hardware trojan detection. In *NORCAS*, 2015.

[64] Michaël Marcozzi, Qiyi Tang, Alastair F Donaldson, and Cristian Cadar. Compiler fuzzing: How much does it matter? *PACMPL*, 2019.

[65] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 1998.

[66] Kevin E Murray, Mohamed A Elgammal, Vaughn Betz, Tim Ansell, Keith Rothman, and Alessandro Comodi. Symbiflow and vpr: An open-source design flow for commercial and novel fpgas. *MICRO*, 2020.

[67] Adib Nahiyan, Mehdi Sadi, Rahul Vittal, Gustavo Contreras, Domenic Forte, and Mark Tehranipoor. Hardware trojan detection through information flow security verification. In *ITC*, 2017.

[68] Yannic Noller, Corina S Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. Hydiff: Hybrid differential software analysis. In *ICSE*, 2020.

[69] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Information flow isolation in i2c and usb. In *DAC*, 2011.

[70] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *USENIX Sec*, 2020.

[71] Joshua M Pearce. Building research equipment with free, open-source hardware. *Science*, 2012.

[72] Joshua M Pearce. Quantifying the value of open source hardware development. *Modern Economy*, 2015.

[73] Lisa Piper and Jin Zhang. Don't let the x-bugs bite: Conquer elusive x-propagation issues early! get them before they get you! In *ASICON*, 2011.

[74] Kushal Kumar Ponugoti. *Formal Verification for Hardware Trojan Detection*. PhD thesis, North Dakota State University, 2023.

[75] eInfochips Priyambada Mishra. Understanding logic equivalence check (lec) flow and its challenges and proposed solution. https://www.design-reuse.com/articles/51622/understanding-logic-equivalence-check-lec-flow-and-its-challenges-and-proposed-solution.html. [Online; accessed 8-Feb-2024].

[76] Rong Qu, Jiangang Huang, Long Zhang, Tianlu Qiao, and Jian Zhang. Scope-based compiler differential testing. In *QRS*, 2023.

[77] Michael Rathmair, Florian Schupfer, and Christian Krieg. Applied formal methods for hardware trojan detection. In *ISCAS*, 2014.

[78] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, 2017.

[79] Katharina Ruep and Daniel Große. Spinalfuzz: Coverage-guided fuzzing for spinalhdl designs. In *ETS*, 2022.

[80] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. A novel technique for improving hardware trojan detection and reducing trojan activation time. *VLSI*, 2011.

[81] S. Sargsyan, S. Kurmangaleev, M. Mehrabyan, M. Mishechkin, T. Ghukasyan, and S. Asryan. Grammar-based fuzzing. In *IVMEM*, 2018.

[82] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. https://clang.llvm.org/docs/AddressSanitizer.html. [Online; accessed 25-Jan-2024].

[83] Mohamed Shalan and Tim Edwards. Building openlane: a 130nm openroad-based tapeout-proven flow. In *ICCAD*, 2020.

[84] Lixiang Shen, Dejun Mu, Guo Cao, Maoyuan Qin, Jeremy Blackstone, and Ryan Kastner. Symbolic execution based test-patterns generation algorithm for hardware trojan detection. *computers & security*, 78, 2018.

[85] Zachary Snow. sv2v: Systemverilog to verilog. https://github.com/zachjs/sv2v. [Online; accessed 16-May-2024].

[86] Wilson Snyder. Verilator and systemperl. In *NASCUG*, 2004.

[87] RISC-V Software. riscv-tests. https://github.com/riscv-software-src/riscv-tests. [Online; accessed 5-Feb-2024].

[88] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Cascade: Cpu fuzzing via intricate program generation. In *USENIX Sec*, 2024.

[89] Flavien Solt, Ben Gras, and Kaveh Razavi. {CellIFT}: Leveraging cells for scalable and precise dynamic information flow tracking in {RTL}. In *USENIX Sec*, 2022.

[90] Flavien Solt, Patrick Jattke, and Kaveh Razavi. Rememberr: Leveraging microprocessor errata for design testing and validation. In *MICRO*, 2022.

[91] Flavien Solt and Kaveh Razavi. Hybridift: Scalable memory-aware dynamic information flow tracking for hardware. In *ICCAD*, 2024.

[92] SpinalHDL. Vexriscv. https://github.com/SpinalHDL/VexRiscv. [Online; accessed 25-Jan-2024].

[93] Suriya Srinivasan and Ranga Vemuri. Trojan localization using information flow tracking properties in soc designs. In *VLSID*, 2024.

[94] P. Srivastava and M. Payer. Gramatron: Effective grammar-aware fuzzing. In *ISSTA*, 2021.

[95] Stuart Sutherland. I'm still in love with my x! In *DVCon*, 2013.

[96] Mohammad Tehranipoor and Farinaz Koushanfar. A survey of hardware trojan taxonomy and detection. *IEEE design & test of computers*, 2010.

[97] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks. Fuzzing hardware like software. In *USENIX Sec*, 2022.

[98] Mike Turpin. Solving verilog x-issues by sequentially comparing a design with itself. you'll never trust unix diff again! *SNUG*, 2005.

[99] Haoyi Wang, Chenguang Wang, Yici Cai, and Qiang Zhou. A high-level information flow tracking method for detecting information leakage. *Integration*, 69, 2019.

[100] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: Grammar-aware greybox fuzzing. In *ICSE*, 2019.

[101] P. Wang, X. Zhou, K. Lu, T. Yue, and Y. Liu. Sok: The progress, challenges, and perspectives of directed greybox fuzzing. *arXiv:2005.11907*, 2020.

[102] Stephen Williams and Michael Baxter. Icarus verilog: open-source verilog more than a year later. *Linux Journal*, 2002.

[103] C. Wolf, J. Glaser, and J. Kepler. Yosys-a free verilog synthesis suite. In *Austrochip*, 2013.

[104] Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. {MorFuzz}: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation. In *USENIX Sec*, 2023.

[105] Xiao-lang Yan, Long-li Yu, and Jie-bing Wang. A front-end automation tool supporting design, verification and reuse of soc. *Journal of Zhejiang University-SCIENCE A*, 2004.

[106] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011.

[107] YosysHQ. Vloghammer. https://github.com/YosysHQ/VlogHammer. [Online; accessed 30-January-2024].

[108] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. In *VLSI*, 2019.

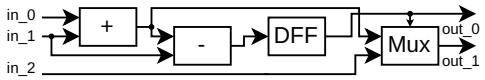

Figure 21: Concrete subnet example. DFF represents a flip-flop (the clock signal is implicit) and Mux is a multiplexer.

Algorithm 1: Non-combinational loop insertion.

Data: An input port of a given cell C
Result: An eligible cell C' whose output will be connected to C's input

```
reds = ∅;
redfanout = {C} greens = ∅;
greenfanout = ∅;
whites = allCells - {C};
currcolor = red;
while redfanout ≠ ∅ or greenfanout ≠ ∅ do
    if redfanout ≠ ∅ then
        D = redfanout.pop();
        currcolor = red;
    else
        D = greenfanout.pop();
        currcolor = green;
    whites.remove(D);
    if stateful(D) or currcolor == green then
        greens.add(D);
        greenfanout.addmultiple(D.fanout ∩ whites);
    else
        reds.add(D);
        redfanout.add(D.fanout ∩ whites);
return pick(greens) if greens ≠ ∅ else ⊥
```

A Sound Subnet Construction

Figure 21 shows a concrete subnet example. To enforce the single-driver rule, we ensure that a wire only takes one of the following roles: a design input, a cell output, or a design output. To enforce the loop-free rule, we build subnets as sequences of cells, in which each cell can only be driven by the subnet input ports or by the output of a previous cell. For maximizing relational diversity, TRANSFUZZ inserts non-combinational loops through the following algorithm (Algorithm 1): for a cell C, color all cells in its subnet in white (non-successor), red (combinational successor) or green (non-combinational successor). All cells start white. Only green cells are eventually eligible for loop insertion.

B Cell interconnection

Algorithm 2 specifies the input selection algorithm that connects the cells together, as described in Section 4.

Algorithm 2: Input selection algorithm sketch. The function *pickcell* selects some previous cell, with some bias toward the yet unused output bits.

Data: port_width
Result: Inputs for the next macrocell input port

```
in_offset = 0; remaining_bits = input_bits;
while remaining_bits > 0 do
    C = pickcell(cell_id);
    out_offset = C.out.width; conn_width =
      min(remaining_bits, C.out.width-out_offset);
    connect(port_width-remaining_bits, out_offset,
      conn_width);
return connections
```

Table 4: CVEs assigned for bugs found by TRANSFUZZ.

Id	CVE-2024-	CVSS	Id	CVE-2024-	CVSS
I1	25470	7.1	V6	25493	7.1
I2	25471	7.1	V7	25485	7.1
I3	-	-	V8	-	-
C1	26522	7.1	V9	25486	7.1
C2	28720	7.5	V10	25488	7.1
C3	25472	7.1	V11	25491	7.1
C4	-	7.1	V12	25490	7.1
C5	28719	7.1	V13	28721	7.1
C6	25478	7.5	V14	25489	7.1
C7	-	-	V15	25492	7.1
C8	-	-	V16	25495	7.1
V1	25481	7.5	V17	25494	7.5
V2	25480	7.5	V18	25496	7.1
V3	25482	7.5	Y1	25479	7.1
V4	-	-	Y2	25477	7.1
V5	25484	7.5	-	-	-

C Consistency Across EDA Applications

While Verilog specifies 4 levels of logic (0, 1, X: "don't care", Z: "high impedance"), we use the special `bit` construct that only supports 2 levels (0, 1). However, the Verilog standard specifies divisions and modulo by zero as a special case where X propagates through `bit` signals, and then propagates through the design. We therefore enforce that all divisors and modulo divisors have at least one bit constantly tied to VCC. We cannot exclude that propagating X would reveal bugs, but no existing bug was related. Yet as shown in Section 6.5, some unused bits in wires are X and revealed bug I2.

D Common Vulnerabilities and Exposures

In this Appendix, we summarize the CVEs corresponding to bugs found by TRANSFUZZ in Table 4.