BugsBunny: Hopping to RTL Targets with a Directed Hardware-Design Fuzzer

Hany Ragab*, Koen Koning*, Herbert Bos, and Cristiano Giuffrida

hany.ragab@vu.nl, koen.koning@vu.nl, herbertb@cs.vu.nl, giuffrida@cs.vu.nl

Vrije Universiteit Amsterdam
The Netherlands

**Equal contribution joint first authors*

***Abstract*—Recent attacks on modern processors have demonstrated the severe consequences of discovering and exploiting hardware vulnerabilities. Simultaneously, the increasing complexity of modern chip designs and the ever-limited testing time presents numerous challenges to existing pre-silicon hardware-design verification tools.**

Fuzzing is increasingly the technique of choice for discovering software vulnerabilities, but the same cannot be said about fuzzing for hardware designs vulnerabilities. Due to the data-flow nature of how hardware is designed, existing software fuzzing solutions cannot be readily applied in the hardware context, and the performance of the proposed hardware fuzzing solutions suffers from state explosion when applied on complex hardware designs.

In this work, we present BugsBunny, a feedback-guided directed hardware-design fuzzer which aims to reduce the costs of pre-silicon validation. BugsBunny focusses the testing resources only on the relevant parts of the design-under-test (DUT), by fuzzing towards a certain target state of the DUT and eliminating irrelevant parts of the design. We propose a novel distance-to-target feedback metric, capable of directing and guiding the fuzzer towards the desired target state, which is based on lightweight data-flow analysis and instrumentation of the DUT. By running the DUT on an FPGA, BugsBunny achieves high fuzzing throughput, outperforming existing simulation-based solutions.

We perform an end-to-end evaluation of BugsBunny on complex SoC designs (e.g., the RISC-V BOOM), where preliminary experiments demonstrate a significant reduction in the number of fuzzing seeds that are required before the DUT reaches the target state.

***Index Terms*—hardware design, RTL, directed fuzzing**

1. Introduction

Electronic design automation is the process of designing, simulating, and testing electronic circuits using software prior to fabrication. Chip manufacturers save millions of dollars in non-recoverable engineering costs by simulating designs to detect flaws and correct them prior to the silicon device fabrication. However, as hardware designs become more complex, the difficulty of testing increases proportionately. As a result, numerous hardware vulnerabilities have been uncovered, putting at risk millions of users [1], [2], [3]. Therefore, we need ways to test hardware early on, ideally at the level of a hardware description language (HDL) such as Verilog or VHDL [1].

On the software side, modern (*greybox*) fuzzing has grown into the de-facto standard technique for detecting vulnerabilities at scale [4]. By automatically mutating and testing inputs, guided by feedback from the executions such as code coverage, modern fuzzers can effectively explore programs to trigger crashes and uncover bugs. Unfortunately, hardware designs are distinctly different from software and the existing software-fuzzing solutions cannot be directly applied to testing hardware designs. Recent work [5], [6], [7], [8], [9] has investigated how to model and express certain hardware properties required in a design fuzzing process, e.g., how to express coverage on (mainly data-flow oriented) hardware and how to deal with complex system on chip (SoCs) while avoiding state explosion, but these problems remain far from solved.

One way to reduce the complexity of hardware fuzzing is to draw from another well-known technique in the software fuzzing world: *directed fuzzing*, that is directing the fuzzer to reach a particular place in the code [10], [11], [12], [13]. By focussing on a specific target, the fuzzer can avoid wasting time and resources exploring less interesting parts of the program. Modern directed greybox fuzzers for software revolve around the ability to measure and minimize the *distance* between program inputs and the fuzzing target location. Thanks to the program control-flow graph (CFG), this distance is relatively easy to calculate for software fuzzers. However, this is not the case for RTL fuzzers due to the data-flow nature of RTL.

In this paper, we argue that applying directed fuzzing on hardware designs can be an important step to accelerate the process of hardware verification. In particular, our hypothesis is that directed fuzzing can help efficiently handle large hardware designs by minimizing the time needed to validate specific properties of the design under test (DUT) and focusing the testing resources towards specific design targets in specific modules. This target can either be inferred manually (e.g., existing assertions in the RTL) or manually chosen by an analyst. For example, an analyst may want to find ways of triggering a transient execution windows. A target can also be semi-automated: an analyst may want to know if variants of a mitigated

1. The "source code" of hardware designs, expressed at the register transfer level (RTL).

```
1  module Counter (
2      input clk,
3      input reset,
4      input enabled,
5      input [7:0] incr,
6      input [7:0] max,
7      output [7:0] out
8  );
9      reg [7:0] cnt;
10     wire [7:0] cnt_next;
11
12     assign cnt_next = enabled ? cnt + incr :
       ↪  cnt;
13     assign out = cnt;
14
15     always @(posedge clk)
16         if (reset)
17             cnt <= 0;
18         else if (cnt_next >= max)
19             cnt <= 0;
20         else
21             cnt <= cnt_next;
22 endmodule
```

Listing 1: Verilog code of an example hardware module called Counter.

hardware vulnerability still exist, and thus direct the fuzzer to the mitigated vulnerability.

We present BugsBunny, a work-in-progress RTL directed fuzzing framework that can be used in the validation of arbitrary and complex hardware designs, including the RISC-V BOOM core [14]. We propose a design for calculating the distance based on approximating the data flow from the input to the target location in the DUT. We first statically analyze the RTL of the DUT to extract a dependency tree of the fuzzing target. Then, we instrument the DUT RTL to gather the information needed to calculate the distance during the fuzzing loop. Our framework features a fuzzer that operates on RTL signals, is clock-cycle aware, and can isolate subsets of the modules in large designs. Finally, BugsBunny runs inputs on the instrumented RTL using an FPGA, to optimize the executions per second.

To summarize, we present the following contributions:

- A scalable fuzzing framework for testing complex hardware designs, including a method to approximate distance for use in directed fuzzing.
- A prototype end-to-end implementation of BugsBunny, including FPGA-acceleration.
- A preliminary evaluation showing promising results and a practical solution that can operate on complex SoCs including the RISC-V BOOM core.

2. Background

2.1. Hardware Design

Hardware is typically described in languages such as Verilog or VHDL, called hardware description languages (HDLs). These describe the circuitry at the register transfer level (RTL), and can either be simulated or synthesized for hardware such as field programmable gate arrays (FPGAs) or application-specific integrated circuit (ASICs). While hardware descriptions may look like typical software programs, their design is distinctly different. In RTL, the design is described in sequential and combinational logic. Sequential logic is clock-sensitive, and includes registers (flip-flops) that update every clock cycle and retain their value between clock cycles. Combinational logic, on the other hand, is simply a series of wires and logic gates (e.g., AND, XOR, etc.) that update immediately once their input changes (modulo the propagation delay of electronic signals). This logic is contained within *modules*, which have a list of input and output ports. One module can instantiate other modules to create a hierarchy.

```
1  module Counter:
2  input clock : Clock
3  input reset : UInt<1>
4  input enabled : UInt<1>
5  input incr : UInt<8>
6  input max : UInt<8>
7  output out : UInt<8>
8
9  reg cnt : UInt<8>, clock
10
11 node cnt_add = add(cnt, incr)
12 node cnt_next = mux(enabled, cnt_add, cnt)
13 node cnt_wrap = mux(geq(cnt_next, max),
   ↪  UInt<8>("h0"), cnt_next)
14 cnt <= mux(reset, UInt<8>("h0"), cnt_wrap)
15
16 out <= cnt
```

Listing 2: FIRRTL intermediate representation of the Counter module.

Listing 1 shows an example of a Verilog counter module. Lines 1–8 define the module and its inputs and outputs. The sequential logic of this module is at lines 15–21, and runs on the positive edge of the clock signal. Here the `cnt` register is updated to either 0 (if the `reset` signal is high or the counter wraps around) or `cnt_next`. `cnt_next` is combinational logic defined at line 12: it is either the addition of the old value of `cnt` plus the `incr`, or the old value, depending on the `enabled` input.

FIRRTL [15] is an intermediate representation (IR) for RTL. The FIRRTL IR is part of the Chisel [16] project, which is used by RISC-V cores such as Rocket [17] and BOOM [14]. Additionally, existing Verilog code can be lifted to FIRRTL. This IR, in turn, can easily be analyzed and instrumented by compiler passes, and can finally be lowered to Verilog. This Verilog can then be simulated or synthesized. Listing 2 shows the FIRRTL representation of the above Verilog Counter module. This demonstrates well that most hardware designs use a data-flow rather than control-flow model. For example, while Verilog had an `if-else` (at lines 16–21), FIRRTL shows explicitly that in hardware this is a combinational multiplexer (`mux`, lines 13 and 14).

2.2. Directed Software Fuzzing

A software fuzzer is a tool that tests a program by generating inputs that explore its code in the hope of finding "crashes", such as assertion failures or segmentation faults. Mutational fuzzers take an initial input and *mutate* this seed to generate new inputs. Modern mutational fuzzers are *greybox* [4], in that they select their mutants based on execution feedback such as code coverage: information from the program to identify "good" mutations, e.g., those that explored code which was not seen before.

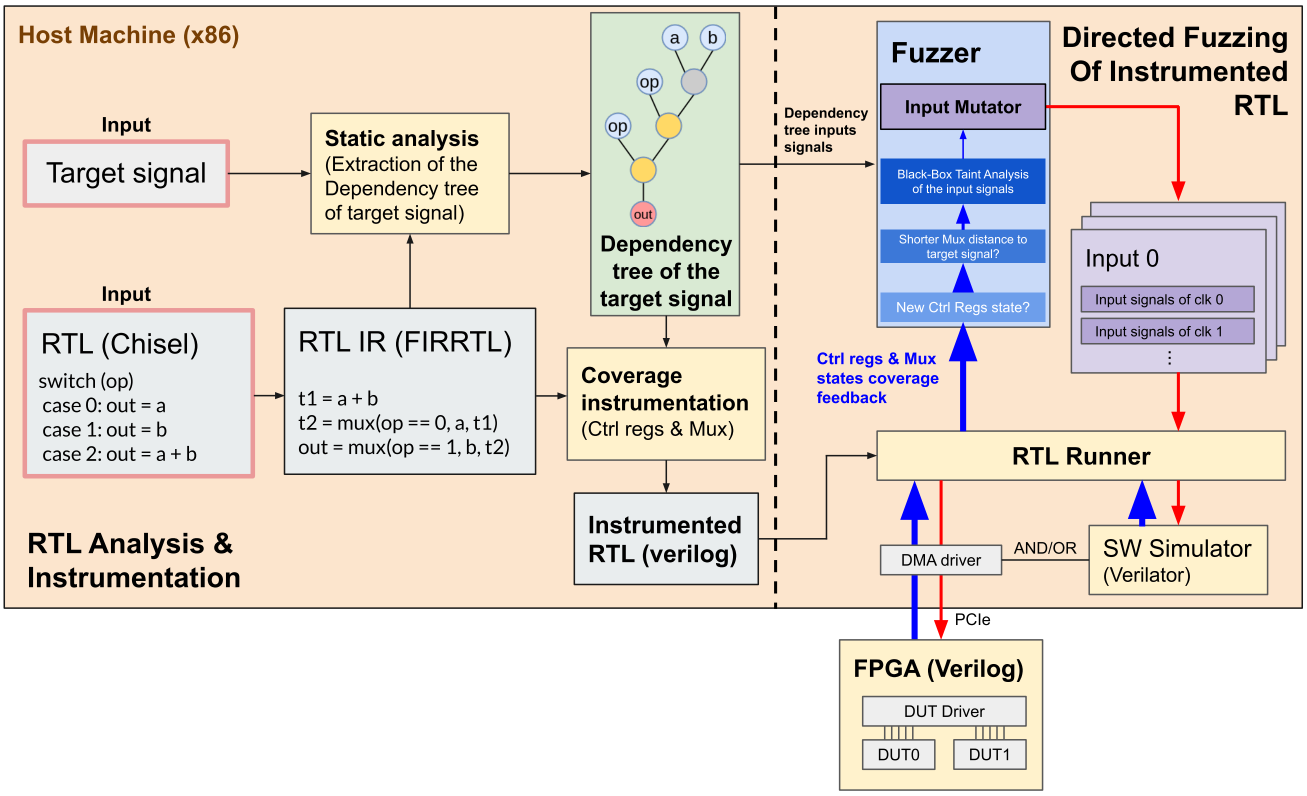
Host Machine (x86)
Input
Target signal
Static analysis (Extraction of the Dependency tree of target signal)
a
b
op
op
out
Dependency tree of the target signal
Input
RTL (Chisel)
switch (op)
case 0: out = a
case 1: out = b
case 2: out = a + b
RTL IR (FIRRTL)
t1 = a + b
t2 = mux(op == 0, a, t1)
out = mux(op == 1, b, t2)
Coverage instrumentation (Ctrl regs & Mux)
Instrumented RTL (verilog)
RTL Analysis & Instrumentation
Dependency tree inputs signals
Fuzzer
Input Mutator
Black-Box Taint Analysis of the input signals
Shorter Mux distance to target signal?
New Ctrl Regs state?
Directed Fuzzing Of Instrumented RTL
Input 0
Input signals of clk 0
Input signals of clk 1
Ctrl regs & Mux states coverage feedback
RTL Runner
DMA driver
AND/OR
SW Simulator (Verilator)
PCIe
FPGA (Verilog)
DUT Driver
DUT0
DUT1

Figure 1: Overview of the fuzzer stages, components and fuzzing feedback loop.

A large body of works exists that explore different aspects of software fuzzing, but most state-of-the-art fuzzers [4], [18] use some variation of *branch coverage*: the target program is instrumented to record which branches are taken. Mutated inputs that hit new coverage are added back to the queue of inputs, so that they will later be used again for further mutations.

A fuzzer operating in this manner will explore the program as broadly as possible, since the exploration is rewarded by the coverage feedback. *Directed* fuzzers [10], [11], [12], [13], on the other hand, try to fuzz towards a *specific* target, such as a particular target code location. For this purpose, they use additional feedback from the program to measure a *distance* to the target. The latter estimates how far the input was from reaching the target, a metric useful to prioritize mutants: those with a shorter distance to target are preferred. The distance feedback is typically computed over the control-flow graph (CFG), similar to the coverage feedback.

Because the state and execution of software is expressed through its control flow, the CFG is a logical way to measure feedback (coverage and distance). However, as we have seen, hardware designs are normally based on a data-flow rather than control-flow model. As such, different coverage metrics are required for hardware [8], [7], [5]. One such metric, called the *register coverage* [5], measures unique states seen during execution, by hashing all state registers (i.e., sequential logic such as finite state machines). For our directed fuzzer, we build on top of this existing coverage metric, more about this in Section 3.3. But not only hardware fuzzing does require a different coverage metric than software, it also requires a completely different approach to tracking distance for directed fuzzing.

3. BugsBunny: A Directed RTL Fuzzer

To address the issues with hardware fuzzing, such as state explosion on complex DUTs and the lack of a CFG, we present BugsBunny, a directed hardware-design greybox fuzzer, guided by both state coverage of the DUT and distance-to-target. Figure 1 shows an overview of our framework. In this section, we briefly describe the overall goals and concepts of our design before going into more details, such as how to model the distance on RTL.

Module trimming Our fuzzer can operate on arbitrary portions (i.e., modules) of the SoC, and eliminate all unnecessary modules. The resulting design under test (*DUT*) can then be evaluated more efficiently. To enable sending inputs to arbitrary modules, where the interface (e.g., AXI, TileLink) is unknown, our fuzzer operates at the *signal level*. We analyze the input ports of the DUT, and construct inputs where each signal is a separate value.

Clock awareness As demonstrated in the example of the Counter of Section 2, data propagates through the DUT over the course of several clock cycles, and data must be fed in for each clock cycle. Our fuzzer generates inputs containing a number of *packets*. Each packet represents the signal values for a single clock cycle. An input with three packets would thus start from the reset state of the module, then run the DUT for three clock cycles.

Fuzzing target Our fuzzer is *directed* towards a certain *target state*: the state the fuzzer should trigger (through a single input) for it to finish. This target state could be a single internal signal (wire or register) having a particular value (e.g., an existing exception signal), or a combination of signals (e.g., a pipeline-flush signal and a cache-miss signal being high). The target can also be defined automatically, such as the condition of an assert

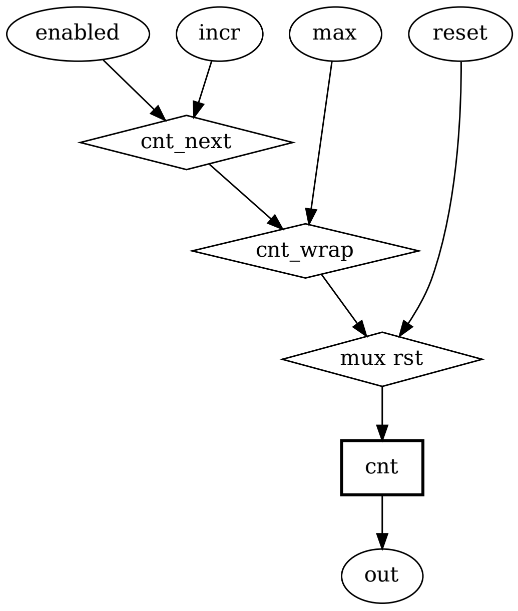

Figure 2: Dependency tree for Counter with target `out`.

statement, or the state of a previous bug when using the fuzzer testing hardware vulnerabilities mitigations.

Dependency tree Based on the fuzzing target, BugsBunny creates a dependency tree (*deptree*) of the data flow of the fuzzing target signal(s). This is done statically during compilation and is a lightweight operation: starting from the target, walk backwards until module input ports are reached. The resulting deptree contains all the information to reconstruct the data flow of the DUT to reach the fuzzing target, such as (clock-sensitive) registers and muxes. The deptree for Counter with target `out` is shown in Figure 2. The deptree serves two purposes: input space reduction and distance modelling. The latter is quite complex and Section 3.3 describes how the deptree is used for distance calculations. The former is simple: the fuzzer only needs to include signals in its packets that can (in any way) influence the target signal.

Fuzzing and input execution It is the fuzzer's job to create inputs based on the deptree and feedback from previous executions. The inputs it generates are then run through the DUT. BugsBunny can execute these tests either in a simulator on the host machine or on a (more scalable) FPGA-hosted environment. From the perspective of the fuzzer, the underlying execution model is simple: it simply feeds in the inputs and expects as feedback the coverage, the distance, and the state of the target signal(s).

Figure 1 demonstrates how all these concepts and components come together. All components to the left are part of the RTL analysis and instrumentation, which are performed once by the compiler. The output of the static analysis (the deptree) is passed to the fuzzer for future distance calculations. The fuzzer then drives the fuzzing loop until the target state is found. In the following sections, we describe these components in more detail.

3.1. RTL Static Analysis and Instrumentation

The goal of the compilation phase is to provide all information required during runtime for the fuzzer, which includes static analysis (deptree) and transformation of the DUT to expose coverage and distance feedback. The result is an instrumented RTL (Verilog) file, which has all irrelevant modules trimmed, exposes signals for feedback during runtime, and has control signals added for the fuzzer to reset the state between inputs.

BugsBunny can compose multiple FIRRTL compiler passes written in Scala, and reuses existing ones alongside newly written ones. For example, for coverage, BugsBunny simply reuses existing state-of-the-art RTL register coverage from DifuzzRTL [5]. Moreover, if future research introduces better coverage metrics for RTL, these passes can easily be plugged in.

3.2. The Fuzzing Loop

At a high level, the fuzzer generates or mutates inputs, executes them on the instrumented DUT, and retrieves feedback used for future iterations of the fuzzer loop. This flow is shown by the red and blue arrows in Figure 1. The fuzzer generates a single *input* that consists of multiple *packets*, each packet representing the value of the *input signals* for a particular clock cycle. The fuzzer feeds these inputs to the RTL runner (red arrows), which returns feedback about the execution (blue arrows). The design of our fuzzer is mainly based on that of software fuzzers such as AFL++ [4], with some custom mutators that operate on the signal and packet level (e.g., duplicate a packet).

In order to maximize the fuzzing throughput, we propose a feedback hierarchy of different filters, each discarding useless inputs as early as possible. Each feedback filter is more expensive to calculate but is required less frequently than the previous one. The first feedback filter is built on top of the register coverage [5] metric, which models the DUT states through its control registers. This feedback filter checks if an input increased the state coverage of the DUT, and if not, the input is not considered useful and is immediately discarded. If the coverage did increase, the second feedback filter is applied, where an approximation of the distance-to-target is calculated and the input is inserted in the priority queue of the fuzzer based on its distance score. In most cases, the fuzzer will then start the loop over again, picking the best-performing input (i.e., the one with the shortest distance-to-target) from the queue to mutate. However, these filters are extensible, and for example a more complex but accurate distance calculation can be used in cases where the fuzzer does not make progress.

3.3. Distance to Target

The distance function models the distance between how far an input has made it (from the input ports of the module) to the target signal. Instead of using costly proper data-flow tracking analysis and instrumentation (e.g., taint analysis), we instead do this heuristically through more light-weight instrumentation that can run on FPGAs. After statically constructing the deptree of the target, we instrument all multiplexers (muxes) along the path of this deptree to expose their state to the fuzzer. At runtime we read the state of these selected muxes, and map this *mux-toggle coverage* [7] onto the deptree. Then, we calculate the shortest path from the target to a toggled mux to obtain the distance-to-target.

Intuitively, a mux can only toggle if data has propagated to the control condition of that mux, giving us some notion of how far data is flowing along the path of the deptree. In practice, however, there are several complications with this scheme that require attention. Firstly, we do not know the "correct" position of each mux, and a mux may already be in the right position at reset to reach the target signal (and flipping it results in getting further away from the target). As such, our distance model does not require all muxes to flip, nor must the distance be zero, it simply has to become less overall.

A more difficult issue is that of the time dimension: our deptree and the mux state give a snapshot of a single clock cycle, but data must often flow through the DUT for several clock cycles. Often it is not possible to toggle all muxes every clock: if there are two muxes controlled by opposite conditions of the same input, then they cannot both be true in the same clock cycle. To account for this, we introduce the concept of *partitions* on top of the deptree. Partitions are subtrees of the deptree that are separated by registers (i.e., clock-sensitive sequential logic). In our example of the Counter, we have two partitions: one containing all muxes before the `cnt` register (P1) and one after (P2, not containing any muxes).

When calculating the distance, each partition calculates a local distance for only that sub-tree (using the shortest path from the root to a toggled mux). Then, the overall distance is calculated as the *weighted* average of these partitions, where the weights are dynamic to stimulate data propagating through the design over multiple clock cycles. For early clock cycles within a single input, partitions close to the inputs of the module are biased (i.e., P1 is favored over P2), whereas for the last clock cycle of an input we favor partitions closer to the target (P2). This algorithm takes into account reachability from the inputs ("can this input-port still influence the target in any way?") and from the target ("could this partition already be influenced by an input early on?").

3.4. RTL Runner

The RTL runner is fed the (instrumented) RTL and a constant stream of inputs from the fuzzer, and is responsible for running these inputs on the DUT. One approach it can take is running them on a simulator: the Verilog is compiled to a software simulation using Verilator [19]. Since everything is in software, this approach is simple: we can feed and inspect any signal of the DUT, and simulate the execution of a single clock cycle at will. However, this approach does not scale well to large designs, leading to a very low throughput of inputs per second.

Therefore the RTL runner can alternatively execute the DUT on an FPGA. Here the DUT can run at much higher speeds (e.g., 100MHz) even for complex designs. However, controlling the DUT is much more difficult. First of all, the inputs must be transferred from the fuzzer to the FPGA, and the feedback must be transferred back. Furthermore, the DUT must be carefully controlled: if an input contains six packets, these packets must be fed in at consecutive clock cycles (at 100MHz speeds); the DUT cannot be paused.

TABLE 1: Fuzzing Targets

Design under test	Target signal
Counter example (Counter)	`cnt == 123`
BOOM-Core Reorder Buffer (ROB)	`is_mini_exception`
BOOM-Core Load-Store Unite (LSU)	`r_xcpt_valid`

To tackle these problems, we have written a custom Verilog *DUT wrapper* connecting the fuzzer on the host with the DUT. On one side, it is connected to the PCIe bus, on the other side it has full control over all signals going in and out of the DUT. It communicates over PCIe to receive input data and respond with the feedback data. This data is cached in memory inside the DUT wrapper. This gives the fuzzer time to prepare all data, and during this time the DUT is held in a reset state. Once ready, the DUT wrapper stops resetting for the specified number of clock cycles (e.g., six, matching the number of packets in the input), and it feeds in the data from memory to the DUT. Once all packets are consumed by the DUT, the current feedback is latched into memory inside the DUT wrapper ready to be read by the fuzzer.

For the FPGA-based executions, we use Vivado with its qdma IP block for PCIe communication. For complex designs, Vivado might take over 2 hours to produce a bitstream file for the FPGA. However, as we will show in the evaluation in Section 4, the performance increase easily amortizes this during fuzzing.

4. Evaluation

To evaluate our prototype fuzzing framework we ran it on several subsets of the BOOMv3 RISC-V core [14]. These results, while preliminary, show that our design can be applied to complex real-world hardware designs. All experiments were performed on a machine with an Intel i9-12900K CPU, 128GB RAM, running Ubuntu 20.04. For RTL simulations, we used Verilator 4.028. For FPGA experiments, we used a Xilinx VCU118 FPGA connected via PCIe x16, with synthesis done by Vivado 2020.2 and the DUT running at 100MHz. For the targets of this evaluation, we ran our toy Counter example and also considered two modules from the BOOM core: the re-order buffer (ROB) and the LSU load-store unit (LSU). As a target, we selected a signal in each that is relatively hard to reach and indicates some form of exception occurred. The LSU represents the single largest module in the BOOM core, whereas the ROB is a medium-sized module. Table 1 summarizes our targets.

To evaluate the execution efficiency of our framework, the effect of using the FPGA, and the impact of distance calculations, we measured the raw fuzzer performance in inputs per seconds. The results of these experiments can be found in Table 2. We can see that while the performance of the FPGA is on-par with that of software simulation for toy programs, the software simulations do not scale at all. Even for medium sized DUTs like the ROB and LSU, the performance of an FPGA is orders of magnitude higher. This difference is unsurprising: for

TABLE 2: Fuzzing performance comparison between SW simulation and FPGA (inputs per second where each input contains 1 packet only)

Design under test	Verilator Cov.	Verilator Cov. + Dist.	FPGA Cov.	FPGA Cov. + Dist.
Counter	1,569,118	1,583,030	1,738,340	1,760,600
ROB	1,208	1,150	16,720	17,769
LSU	24	24	11,453	5,080

software simulations most logic has to be calculated sequentially for each clock cycle, an approach that scales poorly as the complexity increases. On the other hand, the performance of the FPGA-based executions does not suffer from this problem: as long as timing constraints can still be met, the DUT can run at high frequencies. For FPGA performance, the bottleneck is instead the communication overhead. The more bits have to be transferred to the FPGA (packets and signals), the slower execution becomes. Similarly, the more data has to be read back from the FPGA, the more performance suffers. This can be observed when comparing the "Cov." and "Cov. + Dist." results: when using distance, the fuzzer will read back the mux state every time it has to calculate the distance. While some slowdown stems from the distance calculation algorithm in Python, the majority comes from the data reads over PCIe. This also explains why the performance of both versions of Verilator are so close.

Finally, we gathered preliminary results on the effectiveness of our work-in-progress directed fuzzer and its distance algorithm. Table 3 shows two metrics often used to evaluate (directed) fuzzers: *time-to-exposure* and *seeds-to-exposure*. This is simply the (median) time and seeds that was required to reach the target, with lower values indicating faster progress. For the medium-sized ROB module, our fuzzer already finds a satisfying input during its initialization phase (where it generates random seeds for later mutations) in all cases. This shows that on relatively simple modules the search space can easily be explored without the need for (directed) fuzzing. On the other hand, for the more complicated LSU module, we can see the time in seconds is slightly lower when using the distance feedback, albeit not significantly since the module is still relatively small. More notably, the seeds to exposure do differ by an order of magnitude. As we close the performance gap between the two configurations (as we will discuss in Section 5), this will result in far better results in time as well. Because BugsBunny does not currently support directed fuzzing on multiple modules, we can only present results for complex single modules, with the LSU being the most complicated module of the BOOM core. With more work, BugsBunny should also support larger DUTs, where the effect should also be more noticeable. Nevertheless, these results show BugsBunny can have a significant impact on the fuzzing speed when aiming for certain targets.

5. Discussion and Future Work

While BugsBunny shows promising results in the area of directed RTL fuzzing, there is still a lot of room for improvement:

TABLE 3: Distance metric evaluation results (median over 100 runs, time measured on FPGA)

Design under test	Seeds-to-exposure Cov.	Seeds-to-exposure Cov. + Dist.	Time-to-exposure Cov.	Time-to-exposure Cov. + Dist.
ROB	n/a	n/a	0.1 s	0.1 s
LSU	486.0	20.5	24 s	19 s

Inter-module distance The current prototype only accurately models distance on a single module. While BugsBunny supports DUTs with multiple modules, no feedback is present to guide to fuzzer towards this module. While recent work has investigated this area [20], a more efficient and scalable metric is required, in the form of coarse-grained inter-module data-flow information.

Distance accuracy Currently the feedback loop of the fuzzer only uses a single approximate distance-to-target calculation, where efficiency is favored over accuracy. As an additional filter, more accurate but expensive distance calculations can be added for situations when the fuzzer is stuck. For example, an option is to perform dynamic taint analysis [11] and measure the influence of the input on the mux state (for every input signal of every packet). This approach can thus more accurately measure how far the input affects the internal DUT state, but negatively impacts fuzzing throughput.

On-FPGA distance Currently our performance suffers from transferring the required data from the FPGA to the host to calculate distance. We can eliminate this overhead by implementing the distance function on the FPGA itself, bringing the performance of using distance to the same levels as that of using only coverage.

On-FPGA mutators BugsBunny implements its entire fuzzer, including the mutation engine, on the host in Python. This means after every mutation the input data needs to be transferred to the FPGA, a relatively slow process. By implementing the mutators on the FPGA itself, a significant speedup can be achieved.

Improved coverage metrics As more work is published in the area of RTL fuzzing, more advanced and accurate coverage metrics are discovered [8]. Due to the nature of our framework, these can easily be plugged in to enhance both the baseline and directed fuzzing performance.

6. Related Work

Hardware design verification has been a topic of interest for long due to the costly nature of errors in hardware, and is typically done through formal verification [21], [22] or (semi-)automated input generation such as constrained random verification (CRV) [23], [24] or coverage directed test generation (CDG) [25], [26], [27]. Recently, due the an in increase in hardware vulnerabilities [1], [2] and the rise of software fuzzing, we have seen a renewed interest in hardware fuzzing [7], [5], [8], [6], [9], [28]. Many of the commonly used techniques (e.g., formal verification and CRV) require significant manual work from verification engineers, something fuzzers aim to address. Many of the existing techniques only work in a simulation environment, due to tight coupling with the simulator (e.g., for coverage feedback) [6], [8], [25], [28] or cannot scale well to accelerator FPGA environments [7]. In contrast, BugsBunny has synthesizable feedback instrumentation for efficient test execution.

Our work builds on top of recent hardware fuzzing developments [7], [5], and adds the concept of directed fuzzing. Recent work in the same direction [20] only focuses on guiding inputs to the right module overall, whereas BugsBunny focusses on a specific target state.

On the software side, directed fuzzers show their effectiveness in finding specific targets [13], [12], [11], but typically operate on the program's CFG, a concept that does not easily transfer to hardware fuzzing. Recent work to apply conventional software fuzzers to hardware designs (e.g., by applying them on the simulator [6]) shows only limited effectiveness.

7. Conclusion

In this paper, we presented a work-in-progress design and implementation of a directed fuzzing framework for RTL. By using module trimming and directed fuzzing based on a distance-to-target fitness function, our preliminary results show this is a promising direction of research in the area of hardware design fuzzing.

Acknowledgements

We thank Marco Spaziani Brunella for his input, and the anonymous reviewers for their valuable feedback. This work was supported by Intel Corporation through the Side Channel Vulnerability ISRA, and the Netherlands Organisation for Scientific Research through projects "TROPICS" and "Intersect".

References

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.

[2] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *S&P*, 2019.

[3] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks," in *USENIX Security*, 2022.

[4] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *WOOT*, 2020.

[5] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "DIFUZZRTL: Differential fuzz testing to find CPU bugs," in *S&P*, 2021.

[6] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," *arXiv preprint arXiv:2102.02308*, 2021.

[7] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs," in *ICCAD*, 2018.

[8] A. Tyagi, A. Crump, A.-R. Sadeghi, G. Persyn, J. Rajendran, P. Jauernig, and R. Kande, "TheHuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities," *arXiv preprint arXiv:2201.09941*, 2022.

[9] S. K. Muduli, G. Takhar, and P. Subramanyan, "Hyperfuzzing for SoC security validation," in *ICCAD*, 2020.

[10] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "BEACON: Directed grey-box fuzzing with provable path pruning," in *S&P*, 2022.

[11] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "Parmesan: Sanitizer-guided greybox fuzzing," in *USENIX Security*, 2020.

[12] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *CCS*, 2018.

[13] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *CCS*, 2017.

[14] J. Zhao, A. Gonzalez, B. Korpan, and K. Asanovic, "SonicBOOM: The 3rd generation berkeley out-of-order machine," in *CARRV*, 2020.

[15] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," in *ICCAD*, 2017.

[16] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *DAC*, 2012.

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

[18] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *S&P*, 2018.

[19] W. Snyder, "Verilator," https://www.veripool.org/verilator/, Accessed: 2022-03-23.

[20] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi, "DirectFuzz: Automated test generation for RTL designs using directed graybox fuzzing," in *DAC*, 2021.

[21] M. R. Fadiheh, A. Wezel, J. Muller, J. Bormann, S. Ray, J. M. Fung, S. Mitra, D. Stoffel, and W. Kunz, "An exhaustive approach to detecting transient execution side channels in RTL designs of processors," *IEEE Transactions on Computers*, 2022.

[22] Cadence, "JasperGold."

[23] J. Yuan, C. Pixley, and A. Aziz, *Constraint-based verification*. Springer, 2006.

[24] F. Haedicke, H. M. Le, D. Große, and R. Drechsler, "CRAVE: An advanced constrained random verification environment for SystemC," in *SoC*, 2012.

[25] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *DAC*, 2003.

[26] G. Squillero, "MicroGP—an evolutionary assembly program generator," *Genetic Programming and Evolvable Machines*, 2005.

[27] K. Gent, M. Li, and M. S. Hsiao, "Design validation of RTL circuits using evolutionary swarm intelligence," in *ITC*, 2012.

[28] K. Ruep and D. Groß, "SpinalFuzz: Coverage-guided fuzzing for SpinalHDL designs," in *ETS*, 2022.