Fuzzing Soft IPs for Fun & Profit

Raghudeep Kannavara, Meta Platforms
Steven Larson, Intel Corp

1 Introduction

The growing complexity of systems coupled with the advent of increasingly sophisticated security attacks highlights a dire need for advanced automated vulnerability analysis tools. Fuzzing is an effective proven technique to find security critical issues in systems, often without needing to fully understand the internals of the system under test. The main purpose of fuzzing is not to test the correct functionality, but to explore and test that undefined area. Fuzzing is a form of negative testing in which an interface is tested with a large quantity of randomized, invalid input, with the goal of triggering a failure. While fuzzing has been widely adopted in software testing, fuzzing tools and techniques to find security bugs in hardware are fast gaining traction in both academia and industry [1][2][3][7].

In electronic design, a semiconductor intellectual property (IP) core is a reusable unit of logic, cell, or integrated circuit layout design. Soft IPs (SIP) are IP cores generally offered as synthesizable RTL modules. These are developed in Hardware description language like Verilog. A SIP implements a specific logic that is concise enough to be tested using fuzzing techniques while not running into scalability issues often encountered while testing an SoC. Hence, we focus on SIP interface fuzzing to constrain our Design Under Test (DUT) to a specific logic implementation.

Fuzzing stimuli to SIP Verilog modules in a simulation environment is challenging largely because simulation is slow. Even if test cases are generated in parallel, simulation needs to happen in parallel to avoid a bottleneck, along with enabling coherency between different stimulus generators and simulations to avoid repetition of testcases. This is not easily scalable. On the other hand, FPGA models are very useful in enabling faster verification but require efforts to set up emulation or FPGA prototype. Converting C++ or SystemC to Verilog has been well researched with many commercial and opensource tools. At the same time, compiling Verilog into C++ executables or SystemC models is fast gaining traction with tools such as Verilator [4] in the forefront. While this opens exciting new opportunities to harness state of the art coverage driven scalable fuzzers such as AFL++ [5] to test Verilog modules by converting them to C++ binaries, it also comes with the perils of treading uncharted territory. While the technique of applying software fuzzers to verilated hardware modules has been proposed earlier [1][2][3][7], we believe there are fundamental questions that are yet to be answered and solutions to success that are not discussed in previous literature. Previous works have primarily focused on coverage metrics and not on identifying specific CWEs.

With this in view, to complement existing verification methods, in this paper we present a newer approach to fuzzing SIP modules, i.e., using AFL++ to fuzz interfaces of “verilated” SIPs. In the next section, we describe the proposed methodology and tooling required to enable SIP fuzzing. We follow that up with a discussion on the challenges we encountered, recommend solutions, and demonstrate this approach on selected hardware Common Weakness Enumerations (CWE), both of which are novel contributions of our work. Finally, we summarize the paper with conclusions and future plans.

2 Methodology & Tooling

Identify/Isolate the logic & interface to fuzz: This initial step is critical to a successful fuzzing outcome. As with any verification methodology, we need to clearly identify the boundaries of the logic we intend to test. Although the SIP interface at the top-level module is an excellent fuzzing target, it may require modelling behavior of other logic that interacts with the SIP. In many instances, these interactions may already be defined previously to support SIP verification, e.g., Bus Functional Models (BFM). Additionally, debugging the internals of complex IPs can become problematic to identify failure scenarios. On the other hand, we can drill further down in the SIP logic to isolate sections of code that implement a security critical functionality, e.g., SAI based access controls, PCIE packet parsing, address decoding, opcode handling etcetera and focus on fuzzing the interfaces of such logic in isolation.

Write a C++ test harness: Next, we write a C++ test harness, which defines the standard main() that instantiates the cycle-accurate behavioral model as a C++ object. The C++ object in this case is the verilated SIP DUT. This test harness must read stimulus from a file and input it to the DUT. The goal is to malform this input file using fuzzers such as AFL++ and use it to drive DUT stimuli. To limit the interfaces to fuzz, we can hold input signals that do not influence DUT behavior as constant while feeding the input signals that do influence the DUT behavior with the output of the fuzzer.

Compile Verilog / C++ test harness into C++ executable (Verilation): The synthesizable Verilog code is then converted to cycle-accurate C++ behavioral model using Verilator which then, along with C++ test harness is compiled into a simulation executable by Verilator. Verilator is a free and open-source software tool which converts synthesizable Verilog to a cycle-accurate behavioral model in C++ or SystemC. The models typically offer higher performance than the more widely used event-driven simulators, which can process the entire Verilog language and model behavior within the clock cycle. The verilated SIP model in essence is a C++ executable that can run in the user space and reads inputs from a file on disk. This is an ideal target for coverage driven fuzzers such as AFL++. To instrument the executables for AFL++, we can redirect Verilator to use afl-g++ instead of g++ to enable coverage driven unique test case generation (avoiding redundant tests).

Fuzz the interfaces (using AFL++): This is a simple command that launches AFL++ to fuzz the inputs to the C++ binary. Generally, fuzzers log testcases that generate a crash or a hang. But we note that fuzzing is not always about finding crashes and memory corruption issues in the system under test. It is a holistic approach to automate generation of malformed inputs to drive the system under test to undefined or unexpected spaces. One can write assertions or define rules to monitor and log malformed inputs that drive the DUT to such unexpected states.

Techniques to automatically generate assertions or rules including the test harness by parsing design specification documents and Verilog modules along with tainting inputs to understand how inputs flow during simulation can be explored to improve the capabilities of this fuzzer.

Review/Replay/Reuse testcases that cause failures: The goal here is to retrieve the testcases that were logged by the fuzzer and replay those as directed tests in a simulation or emulation environment to ascertain the correctness or incorrectness of the DUT. Moreover, these testcases can be reused in a regression environment to verify fixes are in place. From writing a C++ test harness to replaying the testcases in another verification environment, there are automation opportunities to streamline this process and mainstream adoption of the proposed SIP fuzzing methodology.

3 Challenges & Proposed Solutions

Before we dive into the results of our case studies, we illustrate the limitations of state-of-the-art fuzzers such AFL++ to support SIP fuzzing. To address these limitations, we propose specific solutions or workarounds.

Fuzzers are not built to handle hardware state machines. Finite State Machine (FSM) outputs depend on current inputs, past inputs, current states, and past states (are sequential vs combinatorial), e.g., Moore machines, Mealy machines. An out of box fuzzer is not built to remember past states and past inputs between fuzz iterations, so state transitions need to be accurately handled between each fuzz iteration. The intermediate state of a Verilated model may be saved on disk, so that it may later be restored. This save/restore operation can be called from the C++ test harness. *Leveraging this save/restore ability between each fuzz iteration enables the fuzzer to drive the DUT to the next state based on previous state, previous input, and current input to generate the output signal.* It also allows the fuzzer to not only log test inputs causing failures but also save a snapshot of the state during failure for later introspection.

Generating clock and reset signals is not a native fuzzer capability. Without internal modifications, out of the box software fuzzers cannot generate a clock signal to drive state transitions in the DUT. Similarly, instructing a software fuzzer to assert or deassert a reset signal after "n" clock cycles or when specific conditions are met is challenging. The challenge with clock signal is twofold, we need to toggle the clock between each fuzz iteration, while also monitor the clock context, i.e., for a rising edge or negative edge, to perform state transitions. *Both these issues can be solved using the state save/restore ability provided by Verilator.* On the first fuzz iteration, current state is saved to disk. On subsequent fuzz iterations, previous sate is restored from the disk, previous state of the clock is read, toggled, and input to the DUT in the current state. This new state is now saved to the disk including the clock and the process repeats until a failure case is encountered and logged. A reset signal can similarly be asserted or deasserted via the test harness at random, after 'n' clock cycles or by monitoring state transitions and checking to see if a specific condition is met (e.g., stuck states).

Hangs and crashes are unlike software in hardware. Buffer overflows causing crashes or application resource exhaustion leading to hangs are software centric; these can have a completely different context in hardware fuzzing. Hardware hangs can be due to and not limited to transitions to invalid states, states stuck without transitions, memory or I/O reads that do not return, waiting indefinitely for completions during non-posted transactions etcetera. On the other hand, *a crash occurs when the fuzzer encounters runtime assertions that include $fatal and $error in the verilated DUT.* Similarly, a runtime warning or runtime information message is generated when the fuzzer encounters a $warning or $info assertion respectively. This ability of the fuzzer to comprehend Verilog assertions and behave accordingly is powerful and allows us to monitor for interesting behaviors that may have security impact.

Non-asserted failures are risky. Verilog assertions are great if they are written to catch those corner cases. In security critical flows, there can be undetectable scenarios where assertions are missed but it can still result in an unexpected hardware behavior, invalid (potentially dangerous) output or undefined state. For example, a privilege escalation bug leading to asset access by an untrusted agent can go undetected during fuzzing because an assertion is missing, consequently the fuzzer is made to believe this is indeed a valid scenario and no failure testcase is logged. Fuzzers are generally not good at catching these types of failures. Hence, one must ensure that *assertions are correctly written to capture all potential failure scenarios or the test harness itself can incorporate these checks as a library of issues* to evaluate for during SIP fuzzing.

Fuzzing multiple stimuli can get complex. Fuzzing multiple interfaces at the same time can easily get complicated when using a file fuzzer approach. To drive multiple stimuli accurately, we need *grammar-based test case generation* combined with coverage driven fuzzing. Not many fuzzers combine these two capabilities in one package and even if they exist, they are poorly maintained. Of late, there is a growing interest in custom mutators for AFL++ to handle highly structured inputs, with opensource contributions. We are starting to explore these custom mutators for AFL++ for SIP interface fuzzing. While this is ongoing, currently we are resorting to splitting the fuzzed input blob bitwise to drive multiple input signals. This requires us to clearly understand the bit widths of each input signal, parse the fuzzed input data blob bitwise, *use the bitwise data to drive each bit of the input signal*, and thereby continue to leverage the fuzzer to generate tests to drive multiple stimuli.

Fuzzing hierarchical designs. Hierarchical designs facilitate design of complex architectures and promote design reuse. Fuzzing such designs can get complex and requires a good understanding of the design hierarchy. Verilator supports inclusion of ***/*verilator hier_block*/*** metacomment in the Verilog code of instantiated submodules and nested hierarchical blocks to enable compilation of the entire design hierarchy into a single executable for testing purposes.

Not All Verilog is Synthesizable. Synthesizable Verilog translates into gates, registers, RAMs, etcetera. But not all Verilog should be synthesized, e.g., delays are implemented using clocks and flipflops or loops end up as multiple hardware instances, which may not be the ideal expected outcome, especially for someone who is accustomed to writing and compiling C/C++ code. Although this is not really a fuzzer issue, we think it might be worth noting here since we are in essence discussing fuzzing C++ binaries using AFL++ in this paper.

4 Case Study Results

We evaluate the proposed methodology and tooling to identify certain hardware weaknesses in Verilog code snippets provided as examples to demonstrate the CWE. We selected these CWE

[6] based on the availability of sample code in the online CWE documentation and the suitability for behavioral modeling to demonstrate the efficacy of the proposed approach. This is by no means a comprehensive analysis of all hardware weaknesses for fuzzing, rather a proving ground for future work. We have not included the vulnerable Verilog code snippets in this paper for the sake of readability, but the reader is encouraged to review the code snippets accessible via online CWE documentation linked below and in the reference section [6].

CWE-1311 - Improper Translation of Security Attributes by Fabric Bridge: This weakness arises when a bridge IP block incorrectly translates security attributes from either trusted to untrusted or from untrusted to trusted when converting from one fabric protocol to another. In this CWE example, the OCP2AHB bridge interfaces between Open Core Protocol (OCP) and AMBA-Advanced High-Performance Bus (AHB) end points. OCP uses MReqInfo signal to indicate security attributes, whereas AHB uses HPROT signal to indicate the security attributes. The IP internal logic converts the incoming 5-bit identity (MReqInfo) to output a 2-bit identity (HPROT).

The values 5’h11, 5’h10, 5’h0F, 5’h0D, 5’h0C, 5’h0B, 5’h09, 5’h08, 5’h04, and 5’h02 in MReqInfo indicate that the request is coming from a trusted state of the OCP bus controller. Values 5’h1F, 5’h0E, and 5’h00 indicate untrusted state. HPROT values 2’b00 and 2’b10 are considered trusted, and 2’b01 and 2’b11 are considered untrusted. By fuzzing the incoming 5-bit MReqInfo, we trigger scenarios where trusted identities are translated to untrusted identities and vice versa. We catch these issues by including an allowed list in the test harness, which is evaluated every fuzz iteration and violations are logged.

CWE-1245 - Improper Finite State Machines (FSMs) in Hardware Logic: FSMs can be used to indicate the current security state of the system. Faulty FSM designs that do not account for all states, either through undefined states or through incorrect implementation, can drive the system to an unstable state from which the system cannot recover without a reset, thus causing a DoS. Depending on the FSM use case, an attacker may also gain additional privileges to launch further attacks and compromise system security.

In this CWE example, the FSM assigns the output based on the value of a register, which is determined based on externally provided input. A case statement in the implementation is missing a default statement and does not handle the scenario where certain inputs can push the system to an undefined state leading to an unexpected outcome such as denial of service by being stuck at an undefined state. We trigger such scenarios by fuzzing the input values. Since the state is internal to SIP and not exposed externally as a top-level parameter, we are required to declare such internal signals or variables public by including an inline comment /*verilator public_flat_rw*/ next to the signal or variable declaration in the Verilog code and use Verilator’s Verification Procedural Interface (VPI) in the test harness to access those public values during runtime. By doing so, we can perform runtime state introspection to monitor for states that are stuck across multiple clock cycles and terminate fuzzing by logging a snapshot of the incorrect state for offline analysis.

CWE-1298 - Hardware Logic Contains Race Conditions: A race condition in logic circuits typically occurs when a logic gate gets inputs from signals that have traversed different paths while originating from the same source. Such inputs to the gate can change at slightly different times in response to a change in the source signal. This results in a timing error or a glitch (temporary or permanent) that causes the output to change to an unwanted state before settling back to the desired state. If such timing errors occur in access control logic or finite state machines that are implemented in security sensitive flows, an attacker might exploit them to circumvent existing protections.

In this CWE example, we evaluate a 2x1 multiplexor using logic gates. The output signal 'z' periodically changes to an unwanted state. Thus, any logic that references signal 'z' when it is in an unwanted state ends up in an undefined state. We tried to trigger such glitching scenarios by fuzzing the multiplexer inputs (input0, input1 and select signals). We found that Verilator does not model time delays. Any timing issues such as race conditions cannot be evaluated using Verilator.

CWE-1280 - Access Control Check Implemented After Asset is Accessed: The logic we consider implements a hardware-based access control check. The asset should be accessible only after the check is successful. If, however, this operation is not atomic and the asset is accessed before the check is complete, the security of the system may be compromised, i.e., an untrusted agent can have access to the asset since the access control check is completely bypassed.

The logic implements a protected register. The register content is the asset. Only transactions made by a specific agent (indicated by signal usr_id) 0x4 are allowed to modify the register contents. The signal grant_access is used to provide asset access to the requester via data_out. This buggy implementation uses Verilog blocking assignments for data_out and grant_access. Therefore, these assignments happen sequentially (i.e., data_out is updated to new value first, and grant_access is updated the next) and not in parallel. Therefore, data_out is allowed asset access even before the access control check is complete and grant_access signal is set. Since grant_access does not have a reset value, it will randomly go to either 0 or 1. We trigger this scenario by fuzzing the input usr_id and monitoring for the specific condition where data_out equals data_in value while grant_access is not 1. Since grant_access is an internal signal and not exposed at the top level, we declare it public as explained in the CWE-1245 section and use Verilator’s VPI for runtime access to internal signals. We verify the correctness of the fuzzer behavior by tying grant_access to 1 and observing there are no failures.

CWE-1271 - Uninitialized Value on Reset for Registers Holding Security Settings: In this CWE example, a positive clock edge triggered flip-flop implements a lock bit for test and debug interface. When the circuit is first brought out of reset, the state of the flip-flop will be unknown until the enable signal and D-input signals update the flip-flop state. Before the registers are initialized, there will be a window during which the device is in an insecure state and may be vulnerable to attack. In this example, an attacker can reset the device until the test and debug interface is unlocked and access the test interface until the lock signal is driven to a known state by the logic.

To simulate this attack scenario, we fuzz the D-input while we toggle the reset signal randomly and allow enable signal to be uninitialized, thus enable will randomly go to either 0 or 1. We then monitor for scenarios where the enable signal becomes 1 and attacker-controlled D-input feeds a 0 to disable the lock bit. While we did observe failure scenarios, we believe further investigation of reset flow fuzzing is required, given that Verilator is a two-state simulator and does not support metastability scenarios.

The following table summarizes the results of the CWE case study. Additionally, all failure scenarios were detected within the first minute of fuzzing, it is fast.

CWE #	CWE Focus	Similar Evaluation Areas	SIP Fuzzing Evaluation Results
CWE-1311	Improper Translation of Security Attributes by Fabric Bridge	Address decoding, packet parsing, message routing, format conversions, opcode handling.	Good target for SIP fuzzing, applicable for detecting these types of issues.
CWE-1245	Improper Finite State Machines (FSMs) in Hardware Logic	States stuck at certain values (deadlocks), Unreachable states, metastable states, livelocks.	Deadlocks can be detected. Unreachable states, metastable states or livelocks cannot be easily detected.
CWE-1298	Hardware Logic Contains Race Conditions	Timing issues, signal glitching, concurrency issues.	These issues are not detectable by SIP fuzzing.
CWE-1280	Access Control Check Implemented After Asset Access	Monitoring asset access implemented using Verilog blocking statements (sequential).	Good target for SIP fuzzing, applicable for detecting these types of issues.
CWE-1271	Uninitialized Value on Reset for Registers Holding Security Settings	Reset flows, values of registers or signals during reset, metastability.	Needs further investigation. Prone to false positives or missing real issues. Mostly experimental at this time.

Verilated models are behavioral and cycle accurate, they do not represent synthesized logic. Hence, we are not able to accurately evaluate gate level logic leading to certain side channels for example. Since, SIPs are delivered as RTL modules, we do not consider fuzzing synthesized designs in scope for our work. But it does permit reusing the tests generated during fuzzing on synthesized logic. Although Verilator supports assigning 1'bz and 1'bx to signals, being a two-state simulator, it seems incapable of driving a signal to these values during simulations, so explicit 1'bz or 1'bx assignments may get converted to a binary representation internally.

5 Conclusion & Future Plans

Fuzzing hardware in pre-silicon environment is gaining a lot of attention in the industry. SIP fuzzing leveraging Verilog to C++ conversion is an emerging technique that can provide a powerful complementary capability to drive security assurance of our hardware designs. We believe this approach is promising and can achieve acceptance in the SIP verification community in due course. We propose exploring how the SIP fuzzing framework can utilize existing verification collateral via automation to scale adoption. Another key area to explore is how to implement a reusable library of assertions for a broader set of hardware weaknesses, i.e., attack patterns, that the SIP fuzzing framework can leverage. The tools themselves are free, opensource, community supported and the barrier to entry is considerably low.

6 References

[1] Yu Zhang, Wenlong Feng and Mengxing Huang, Automatic Generation of High-Coverage Tests for RTL Designs using Software Techniques and Tools, *arXiv:1602.06038.*

[2] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach and Koushik Sen, RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs, *IEEE/ACM International Conference on Computer-Aided Design 2018: 1-8.*

[3] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly and Matthew Hicks, Fuzzing Hardware Like Software, *arXiv:2102.02308.*

[4] Verilator [Online], Available: *https://www.veripool.org/verilator/*

[5] AFL++ Overview [Online], Available: *https://aflplus.plus/*

[6] Common Weakness Enumeration [Online], Available: *https://cwe.mitre.org/index.html*

[7] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele and Ajay Joshi, DirectFuzz: Automated Test Generation for RTL Designs using Directed Graybox Fuzzing, *58th ACM/IEEE Design Automation Conference (DAC) 2021.*