Fuzzing Hardware Like Software

Timothy Trippel[*], Kang G. Shin
Computer Science & Engineering
University of Michigan
Ann Arbor, MI
{trippel,kgshin}@umich.edu

Alex Chernyakhovsky,
Garret Kelly, Dominic Rizzo
OpenTitan
Google, LLC
Cambridge, MA
{achernya,gdk,domrizzo}@google.com

Matthew Hicks
Computer Science
Virginia Tech
Blacksburg, VA
mdhicks2@vt.edu

Abstract

Hardware flaws are permanent and potent: hardware cannot be patched once fabricated, and any flaws may undermine even formally verified software executing on top. Consequently, verification time dominates implementation time. The gold standard in hardware Design Verification (DV) is dynamic random testing, due to its scalability to large designs. However, given its undirected nature, this technique is inefficient.

Instead of making incremental improvements to existing dynamic hardware verification approaches, we leverage the observation that existing *software fuzzers* already provide such a solution, and hence adapt them for hardware verification. Specifically, we translate RTL hardware to a software model and fuzz that model directly. The central challenge we address is how to mitigate the differences between the hardware and software execution models. This includes: 1) how to represent test cases, 2) what is the hardware equivalent of a crash, 3) what is an appropriate coverage metric, and 4) how to create a general-purpose fuzzing harness for hardware.

To evaluate our approach, we design, implement, and open-source a *Hardware Fuzzing Pipeline* that enables fuzzing hardware at scale, using only open-source tools. Using our pipeline, we fuzz five IP blocks from Google's OpenTitan Root-of-Trust chip, four SiFive TileLink peripherals, three RISC-V CPUs, and an FFT accelerator. Our experiments reveal a two orders-of-magnitude reduction in run time to achieve similar Finite State Machine coverage over traditional dynamic verification schemes, and 26.70% better HDL line coverage than prior work. Moreover, with our bus-centric harness, we achieve over 83% HDL line coverage in four of the five OpenTitan IPs we study—without any initial seeds—and are able to detect all bugs (four synthetic from Hack@DAC and one real) implanted across all five OpenTitan IPs we study, with less than 10 hours of fuzzing.

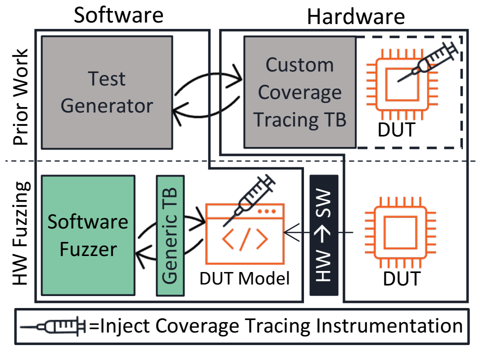

Figure 1: **Fuzzing Hardware Like Software.** Unlike prior Coverage Directed Test Generation (CDG) techniques [6, 22, 39, 65], we advocate for fuzzing software models of hardware directly, with a generic harness (testbench) and feature-rich software fuzzers. This way, we address the barriers to realizing widespread adoption of CDG in hardware DV: 1) efficient coverage tracing, and 2) design-agnostic testing.

1 Introduction

As Moore's Law [48] and Dennard scaling [19] come to a crawl, hardware engineers must tailor their designs for specific applications in search of performance gains [14, 25, 33, 45, 51]. As a result, hardware designs become increasingly unique and complex. For example, the Apple A11 Bionic System-on-Chip (SoC), released over four years ago in the iPhone 8, contains over 40 specialized Intellectual Property (IP) blocks, a number that doubles every four years [62]. Unfortunately, due to the state-explosion problem, **increasing design complexity increases Design Verification (DV) complexity, and therefore, the probability for design flaws to percolate into products.** Since 1999, 247 total Common Vulnerability Exposures (CVEs) have been reported for Intel products, and of those, over 77% (or 191) have been reported in the last four years [18]. While this may come as no surprise, given the onslaught of speculative execution attacks over the past few years [11, 37, 42, 75, 76], it highlights the correlation between hardware complexity and design flaws.

Even worse, hardware flaws are *permanent* and *potent*.

[*] Now at Google (ttrippel@).

Unlike software, there is no general-purpose patching mechanism for hardware. Repairing hardware is both costly, and reputationally damaging [36]. Moreover, hardware flaws subvert even formally verified software that sits above [86]. Therefore, detecting flaws in hardware designs *before* fabrication and deployment is vital. Given these incentives, it is no surprise that hardware engineers often spend more time verifying their designs, than implementing them [21, 83].[1] Unfortunately, the multitude of recently-reported hardware vulnerabilities [11, 37, 42, 47, 75, 76] suggests current efforts are insufficient.

To address the threat of design flaws in hardware, engineers deploy two main DV strategies: 1) *dynamic* and 2) *formal*. At one extreme, *dynamic* verification involves driving concrete input sequences into a Design Under Test (DUT) during simulation, and comparing the DUT's behavior with a set of invariants, or golden model. The most popular dynamic verification technique in practice today is known as Constrained Random Verification (CRV) [1, 16, 30, 88]. CRV attempts to decrease the manual effort required to develop simulation test cases by randomizing input sequences in the hopes of *automatically* maximizing exploration of the DUT state-space. At the opposite extreme, *formal* verification involves proving/disproving properties of a DUT using mathematical reasoning like (bounded) model checking and/or deductive reasoning. While (random) *dynamic* verification is effective at identifying surface flaws in even complex designs, it struggles to penetrate deep into the design state-space. In contrast, formal verification is effective at mitigating even deep flaws in small hardware designs, but fails, in practice, against larger designs.

In search of a hybrid approach to bridge these DV extremes, researchers have ported software testing techniques to the hardware domain in hopes of improving hardware test generation to maximize coverage. In the hardware domain, these approaches are referred to as Coverage Directed Test Generation (CDG) [6, 16, 21, 24, 30, 39, 72, 80, 92, 93]. Like their software counterparts, CDG techniques deploy coverage metrics—e.g., Hardware Description Language (HDL) line, Finite State Machine (FSM), functional, etc.—in a feedback loop to generate tests that further increase state exploration.

While promising, CDG has not seen widespread adoption in hardware DV. As Laeufer *et al.* point out [39], this is likely fueled by several **key technical challenges, resulting from dissimilarities between software and hardware execution models**. First, unlike software, Register Transfer Level (RTL) hardware is not inherently executable. Hardware designs must be simulated, after being translated to a software model and combined with a design-specific testbench and simulation engine, to form a Hardware Simulation Binary (HSB) (Fig. 2). This level of indirection, increases both the complexity and computational effort in tracing test coverage of the hardware. Second, unlike most software, hardware requires *sequences* of structured inputs to drive meaningful state transitions, that must be tailored to each DUT. For example, while software often accepts input in the form of a fixed set of file(s) that contain a loosely-structured set of bytes (e.g., a JPEG or PDF), hardware often accepts input from an ongoing stream of bus transactions. Together, these challenges have resulted in CDG approaches that implement DUT-specific: 1) coverage-tracing techniques [30, 39], and 2) test generators [6, 65, 92].

To supplement traditional dynamic verification methods, we propose an alternative CDG technique we call *Hardware Fuzzing*. **Rather than translating software testing methods to the hardware domain, we advocate for translating hardware designs to software models** and fuzzing those translated models directly (Fig. 1). While fuzzing hardware in the software domain eliminates the need for alternative coverage-tracing mechanisms required by prior CDG techniques [30, 39, 65], since software can be instrumented at compile time to trace coverage, it does not inherently solve the design compatibility issue. Moreover, it creates other challenges we must address. Specifically, to fuzz hardware like software, we must adapt software fuzzers to:

1. interface with HSBs that: a) contain other components besides the DUT, and b) require unique initialization;
2. account for differences between how hardware and software process inputs, and its impact on exploration depth; and
3. design a general-purpose fuzzing harness and a suitable grammar that ensures meaningful mutation.

To address these challenges, we first propose and evaluate strategies for interfacing software fuzzers with HSBs that optimize performance and trigger the HSB to crash upon detection of incorrect hardware behavior. Second, we show that maximizing code coverage of the DUT's software model, by construction, maximizes hardware code coverage. Third, we design an interface to map fuzzer-generated test-cases to hardware input ports. Our interface is built on the observation that unlike most software, hardware requires piecing together a sequence of inputs to effect meaningful state transitions. Lastly, we propose a new interface for fuzzing hardware in a design-agnostic manner: the *bus interface*. Moreover, we design and implement a generic harness, and create a corresponding grammar that ensures meaningful mutations to fuzz bus transactions. Fuzzing at the bus interface solves the final hurdle to realizing widespread deployability of CDG in hardware DV, as it enables us to reuse the same testbench harness to fuzz any RTL hardware that speaks the same bus protocol, irrespective of the DUT's design or implementation.

To demonstrate the effectiveness of our approach, we design, implement, and open-source a Hardware Fuzzing Pipeline (HWFP), inspired by Google's OSS-Fuzz [61], capable of fuzzing RTL hardware at scale (Fig. 5). Using our

[1] It is estimated that up to 70% of hardware development time is spent verifying design correctness [21].

HWFP we: 1) compare Hardware Fuzzing against a conventional CRV technique when verifying over 480 variations of a sequential FSM circuit, 2) compare Hardware Fuzzing against RFUZZ [39] when fuzzing four SiFive TileLink peripherals [63], three RISC-V CPUs [59], and an FFT accelerator [58], and 3) detect five bugs (four synthetic from Hack@DAC [20], and one real) across five commercial IP blocks from Google's OpenTitan silicon Root-of-Trust [44].

To summarize our main results, we demonstrate Hardware Fuzzing:

- provides two orders-of-magnitude reduction in run time to achieve similar FSM coverage than current state-of-the-art CRV schemes (§5.4),
- achieves 24.76% better HDL line coverage (on average) after 24 hours of fuzzing compared with similar hardware fuzzing approaches, i.e., RFUZZ [39] (§6.1),
- identifies all five RTL bugs (both synthetic and real) in five OpenTitan IPs; four in less than 10 minutes, the remaining in less than 10 hours (§6.2).

2 Background

There are two main hardware verification methods: 1) *dynamic* and 2) *formal*. While there have been significant advancements in deploying formal methods in DV workflows [35, 44, 92], dynamic verification remains the gold standard due to its scalability towards complex designs [39]. Therefore, we focus on improving *dynamic* verification by leveraging advancements in the software fuzzing community. Below, we provide a brief overview of the current state-of-the-art in dynamic hardware verification, and software fuzzing.

2.1 Dynamic Verification of Hardware

Dynamic verification of hardware typically involves three steps: 1) **test generation**, 2) **hardware simulation**, and 3) **test evaluation**. First, during *test generation*, a sequence of inputs are crafted to stimulate the DUT. Next, the DUT's behavior—in response to the input sequence—is simulated during *hardware simulation*. Lastly, during *test evaluation*, the DUT's simulation behavior is checked for correctness. These three steps are repeated until all interesting DUT behaviors have been explored. To determine if all interesting behaviors have been explored, verification engineers measure coverage of both: 1) manually defined functional behaviors (functional coverage) [74] and 2) the HDL implementation of the design (code coverage) [32, 56, 70].

2.1.1 Test Generation

To maximize efficiency, DV engineers aim to generate as few test vectors as possible that still close coverage. To achieve this goal, they deploy two main test generation strategies: 1) constrained-random and 2) coverage-directed. The former is typically referred to holistically as *Constrained Random Verification (CRV)*, and the latter as *Coverage Directed Test Generation (CDG)*. CRV is a partially automated test generation technique where manually-defined input sets are randomly combined into transaction sequences [1, 88]. While better than an *entirely* manual approach, CRV still requires some degree of manual tuning to avoid inefficiencies, since the test generator has no knowledge of test coverage. Regardless, CRV remains a popular dynamic verification technique today, and its principles are implemented in two widely deployed (both commercially and academically) hardware DV frameworks: 1) Accellera's Universal Verification Methodology (UVM) framework (SystemVerilog) [1] and 2) the open-source cocotb (Python) framework [77].

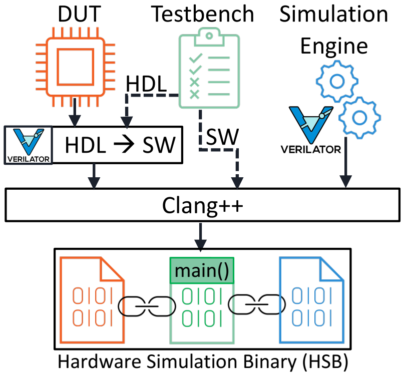

Figure 2: **Hardware Simulation Binary (HSB).** To simulate hardware, the DUT's HDL is first translated to a software model, and then compiled/linked with a testbench (written in HDL or software) and simulation engine to form a *Hardware Simulation Binary (HSB)*. Executing this binary with a sequence of test inputs simulates the behavior of the DUT.

To overcome CRV shortcomings, researchers have proposed CDG [6, 16, 21, 22, 24, 30, 39, 65, 72, 80, 92, 93], or using test coverage feedback to drive future test generation. Unlike CRV, CDG does not randomly piece input sequences together in hopes of exploring new design state. Rather, it *mutates* prior input sequences that explore uncovered regions of the design to iteratively expand the coverage boundary. Unfortunately, due to deployability challenges, CDG has not seen widespread adoption in practice [39]. In this paper, we recognize that existing software fuzzers provide a solution to many of these deployability challenges, and therefore advocate for verifying hardware using software verification tools. The central challenges in making this possible are adapting software fuzzers to verify hardware, widening the scope of supported designs, and increasing the automation of verification.

2.1.2 Hardware Simulation

While there are several commercial [10, 46, 69] and open-source [64, 85] hardware simulators, most work in the same general manner, as shown in Fig. 2. First, they translate hardware implementations (described in HDL) into a software

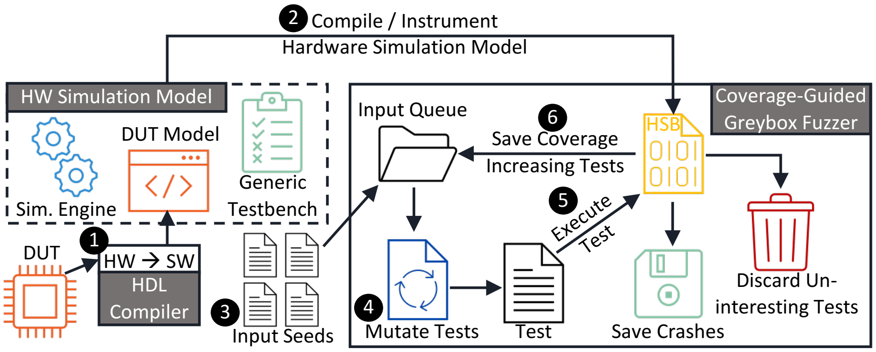

Figure 3: **Hardware Fuzzing.** Fuzzing hardware in the software domain involves: translating the hardware DUT to a functionally equivalent software model (1) using a SystemVerilog compiler [64], compiling and instrumenting a Hardware Simulation Binary (HSB) to trace coverage (2), crafting a set of seed input files (3) using our design-agnostic grammar (§ 4.1.2), and fuzzing the HSB with a coverage-guided greybox software fuzzer [43, 68, 91] (4–6).

model, usually in C/C++. Next, they compile the software model and a testbench—either translated from HDL, or implemented in software (C/C++)—and link them with a simulation engine. Together, all three components form an *Hardware Simulation Binary (HSB)* (Fig. 2) that can be executed to simulate the design. Lastly, the HSB is executed with the inputs from the testbench to capture the design's behavior. Ironically, even though commercial simulators convert the hardware to software, they still **rely on hardware-specific verification tools, likely because software-oriented tools fail to work on hardware models—without the lessons in this paper**. To fuzz hardware in the software domain, we take advantage of the transparency in how an open-source hardware simulator, Verilator [64], generates an HSB. Namely, we *intercept* the software model of the hardware after translation, and instrument/compile it for coverage-guided fuzzing (Fig. 3).

2.1.3 Test Evaluation

After simulating a sequence of test inputs, the state of the hardware (both internally and its outputs) are evaluated for correctness. There are two main approaches for verifying design correctness: 1) invariant checking and 2) (gold) model checking. In invariant checking, a set of assertions (e.g., SystemVerilog Assertions (SVAs) or software side C/C++ assertions) are used to check properties of the design have not been violated. In model checking, a separate model of the DUT's correct behavior is emulated in software, and compared to the DUT's simulated behavior. We support such features and adopt both invariant violations and golden model mismatches as an analog for software crashes in our hardware fuzzer.

2.2 Software Fuzzing

Software fuzzing is an automated testing technique designed to identify security vulnerabilities in software [67]. Thanks to its success, it has seen widespread adoption in both industry [7] and open-source [61] projects. In principle, fuzzing typically involves the following three main steps [50]: 1) **test generation**, 2) **monitoring test execution**, and 3) crash triaging. During test generation, program inputs are synthesized to exercise the target binary. Next, these inputs are fed to the program under test, and its execution is monitored. Lastly, if a specific test causes a crash, that test is further analyzed to find the root cause. This process is repeated until all, or most, of the target binary has been explored. Below we categorize fuzzers by how they implement the first two steps.

2.2.1 Test Generation

Most fuzzers generate test cases in one of two ways, using: 1) a grammar, or 2) mutations. Grammar-based fuzzers [2, 31, 49, 54, 81, 82] use a human-crafted grammar to constrain tests to comply with structural requirements of a specific target application. Alternatively, mutational fuzzers take a correctly formatted test as a seed, and apply mutations to the seed to create new tests. Moreover, mutational fuzzers are tuned to be either: 1) *directed*, or 2) *coverage-guided*. Directed mutational fuzzers [3, 5, 13, 52, 84, 87, 94] favor mutations that explore specific region within the target binary, i.e., prioritizing exploration *location*. Conversely, coverage-guided mutational fuzzers [43, 57, 60, 68, 79, 91] favor mutations that explore as much of the target binary as possible, i.e., prioritizing exploration *completeness*. For this work, we favor the use of mutational, coverage-guided fuzzers, as they are both design-agnostic, and regionally generic.

2.2.2 Test Execution Monitoring

Fuzzers monitor test execution using one of three approaches: 1) blackbox, 2) whitebox, or 3) greybox. Fuzzers that only monitor program inputs and outputs are classified as *blackbox* fuzzers [49, 54, 78]. Alternatively, fuzzers that track detailed execution paths through programs with fine-grain program analysis (source code required) and constraint solving are known as *whitebox* fuzzers [9, 12, 15, 23, 27, 66, 84, 89]. Lastly, *greybox* fuzzers [2, 5, 26, 52, 55, 57, 60, 68, 79, 81, 82, 87, 91, 94] offer a trade-off between black- and whitebox fuzzers by deploying lightweight program analysis techniques, such as code-coverage tracing. Since Verilator [64] produces raw C++ source code from RTL hardware, our approach can leverage *any* software fuzzing technique—white, grey, or blackbox. In our current implementation, we deploy greybox fuzzing, due to its popularity in the software testing community.

3 Approach

To take advantage of advancements in software fuzzing for hardware DV, we propose translating hardware designs to software models, and then fuzzing the model directly. We call this approach, **Hardware Fuzzing**, and illustrate it in Fig. 3. Below we explain the three key components of our approach, including how: 1) RTL hardware is translated to executable software (step 1 in Fig. 3), 2) software fuzzers trace hardware coverage (step 2 in Fig. 3), and 3) fuzzer-generated test cases are interpreted to effectively stimulate the DUT (step 5 in Fig. 3).

3.1 Translating Hardware to Software

Today, simulating RTL hardware involves translating HDL into a functionally equivalent software (C/C++) model that can be compiled and executed (§2.1.2). To accomplish this, most hardware simulators [64, 85] contain an RTL compiler to perform the translation. Therefore, we leverage a popular open-source hardware simulator, Verilator [64], to translate SystemVerilog HDL into a cycle-accurate C++ model for fuzzing.

Like many compilers, Verilator first performs lexical analysis and parsing (of the HDL) with the help of Flex [53] and Bison [73], to generate an Abstract Syntax Tree (AST). Then, it performs a series of passes over the AST to resolve parameters, propagate constants, replace *don't cares* (Xs) with random values, eliminate dead code, unroll loops/generate statements, and perform several other optimizations. Finally, Verilator generates C++ (or SystemC) code representing a cycle-accurate model of the hardware. It creates a C++ class for each Verilog module, and organizes classes according to the original HDL module hierarchy [92].

To interface with the model, Verilator exposes public member variables for each input/output to the top-level module, and a public `eval()` method (to be called in a loop) in the top C++ class. Each input/output member variable is mapped to single/arrayed `bool`, `uint32_t`, or `uint64_t` data types, depending on the width of each signal. Each call to `eval()` updates the model based on the current values assigned to top-level inputs and internal state variables. Two calls represent a single clock cycle (one call for each rising and falling clock edges).

3.2 Hardware Coverage Tracing

To efficiently explore a DUT's state space, CDG techniques rely on tracing coverage of past test cases to generate future test cases. There are two main categories of coverage metrics used in hardware verification [32, 56, 70]: 1) *code coverage*, and 2) *functional coverage*. The coarsest, and most widely-used, code coverage metric is *line coverage*. Line coverage measures the percentage of HDL lines that have been exercised during simulation. Alternatively, *functional coverage* measures the percentage of various high-level design functionalities—defined using special HDL constructs like SystemVerilog Coverage Points/Groups—that are exercised during simulation. Regardless of the coverage metric used, tracing HDL coverage during simulation is often slow, since coverage traced in the software (simulation) domain must be mapped back to the hardware domain [32].

In an effort to compute DUT coverage efficiently prior CDG techniques (RFUZZ [39] and DifuzzRTL [28]) develop custom coverage metrics, e.g., *multiplexer coverage*, that can be monitored by instrumenting the RTL directly. To insert the instrumentation HDL into the design, these techniques implement a custom FIRRTL compiler optimization pass. However, this limits their approach to designs that are implemented in a high-level HDL like Chisel [4] or FIRRTL [41], since their instrumentation compiler can only process designs in HDLs that are *translateable* to FIRRTL.[2]

Rather than make incremental improvements to existing CDG techniques, we recognize that: 1) software fuzzers already provide an efficient mechanism—e.g., ***binary instrumentation*** automatically inserted by compiler optimization passes—to trace coverage of compiled C++ hardware models (HSBs), and 2) the way Verilator translates RTL hardware to software makes mapping software coverage to hardware coverage ***implicit***. On the software side, there are three main code coverage metrics of increasing granularity: 1) basic block, 2) basic block edges, and 3) basic block paths [50]. The most popular coverage-guided fuzzers—AFL [91], libFuzzer [43], and honggfuzz [68]—all trace *edge* coverage. On the hardware side, Verilator conveniently generates straight-line C++ code for both blocking and non-blocking[3] SystemVerilog statements [92], and injects conditional code blocks (basic blocks) for SystemVerilog Assertions and Coverage Points. Therefore, **optimizing test-generation for *edge* coverage of the software model of the hardware during simulation, translates to optimizing for *code*, FSM, and *functional***

[2] The RFUZZ paper states: *"Our tool is language-agnostic since it can work on arbitrary RTL designs expressed in the FIRRTL IR. Once a target design is translated into FIRRTL IR from its source HDL, we can apply compiler passes for the target RTL regardless of its source HDL"* [39]. This implies, the DUT must be described in an HDL that is translatable to FIRRTL (e.g., Chisel). If a design is written in (System)Verilog, as most are, this translation is experimental at best [8].

[3] Verilator imposes an order on the non-blocking assignments since C++ does not have a semantically equivalent assignment operator [64, 92]. Regardless, this ordering does not effect code coverage.

coverage of the RTL hardware itself. We demonstrate this artifact in §5.4, §6.1–6.2, and Appendix B.3.

3.3 Interpreting Fuzzer-Generated Tests

For most software, a single input often activates an entire set of state transitions within the program. Consequently, the most popular software fuzzers assume the target binary reads a single dimensional input—e.g., a single image or document—from either a file, `stdin`, or a byte array [43, 68, 91]. As Laeufer *et al.* point out [39], the execution model of hardware is different. In an HSB, a *sequence* of inputs is required to activate state transitions within the DUT. For example, a 4-digit lock (with a keypad) only has a *chance* of unlocking if a sequence of four inputs (test cases) are provided. Fuzzing this lock with single test cases (digits), will fail. Likewise, fuzzing HSBs with software fuzzers that employ a *single-test-case-per-file* model will also fail. Therefore, to stimulate hardware with software fuzzers, we interpret single dimensional fuzzer-generated tests in two dimensions: space and time. We implement this interface in the form of a generic fuzzing harness (testbench), which we describe in §4.1.

4 Implementation

While Verilator and fuzzer-provided compilers already provide solutions to the first two components of our approach, *hardware to software translation* and *coverage tracing*, the remaining component, *interpreting fuzzer-generated tests* (§3.3) requires a more tailored solution. Therefore, below we describe how to implement a generic fuzzing testbench harness to interpret fuzzer-generated tests. Additionally, we briefly describe the open-source infrastructure we implement to fuzz hardware at scale on Google Cloud Platform (GCP).

4.1 Generic Fuzzing Testbench Harness

To adapt software fuzzers to the hardware execution model, we implement a generic fuzzing harness (testbench) that transforms one-dimensional test inputs, into a two-dimensional *sequence* of inputs (§3.3). Our fuzzing harness—shown in Algo. 1—continuously: 1) reads byte-level portions of fuzzer-generated test files, 2) maps these bytes to hardware input ports, and 3) advances the simulation clock by calling the model's `eval()` method twice, until there are no remaining bytes to process.

4.1.1 Bus-Centric Harness

While the multi-dimensional fuzzing interface we develop enables fuzzer-generated tests to effect state transitions in hardware, it is not design-agnostic. Specifically, the ports of a hardware model are not iterable (Algo. 1: line 4). A DV engineer would have to create a unique fuzz harness (testbench) for each DUT they verify. To facilitate DUT portability, we take inspiration from how hardware engineers interface IP cores within an SoC [17]. Specifically, we propose fuzzing IP cores at the bus interface using a bus-centric harness.

Algorithm 1: Generic Hardware Fuzzing harness (testbench) that maps one-dimensional fuzzer-generated test files to both spatial and temporal dimensions.

```
   Input: fuzz_test_file.hwf
1  $dut \leftarrow Vtop()$;
2  $tf \leftarrow open(fuzz\_test\_file.hwf)$;
3  while tf not empty do
4      foreach $port \in dut.inputs$ do
5          tf.read((uint_8t*) port, $sizeof$(port));
6      end
7      for $k \leftarrow 1$ to 2 do
8          $clock \leftarrow (clock + 1) \% 2$;
9          $dut.eval()$;
10     end
11 end
```

To implement this harness, we could alter our prior harness (Algo. 1) by mapping bytes from fuzzer-generated test files to temporal values for specific signals of a bus-protocol of our choice. However, this would create an exploration barrier since bus-protocols require structured syntax, and most mutational fuzzers lack syntax awareness [90]. In other words, the fuzzer would likely get stuck trying to synthesize a test file, that when mapped to spatio-temporal bus signal values, produces a valid bus-transaction. Instead, we implement a harness that decodes fuzzer-generated test files into sequences of properly structured bus transactions using a bus-centric grammar we describe below. Our current bus-centric harness is implemented around the TileLink Uncached Lightweight (TL-UL) bus protocol [29] with a 32-bit data bus, and illustrated in Fig. 13.

4.1.2 Bus-Centric Grammar

To translate fuzzer-generated test files into valid bus transactions we construct a Hardware Fuzzing grammar. We format our grammar in a compact binary representation to facilitate integration with popular greybox fuzzers that produce similar formats [43, 68, 91]. To match our bus-centric harness, we implement our grammar around the same TL-UL bus protocol [29]. Our grammar consists of *Hardware Fuzzing instructions* (Fig. 4), that contain: 1) an 8-bit opcode, 2) 32-bit address field, and 3) 32-bit data field. The opcode within each instruction determines the bus transaction the harness performs. We describe the mappings between opcodes and TL-UL bus transactions in Table 1.

Note, there are two properties of our grammar that leave room for various harness (testbench) implementations, which we study in Appendix B. First, while we define only three opcodes in our grammar, we represent the opcode with an entire byte, leaving it up to the harness to decide how to map

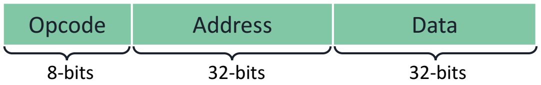

Figure 4: **Hardware Fuzzing Instruction.** A bus-centric harness (testbench) reads binary *Hardware Fuzzing Instructions* from a fuzzer-generated test file, decodes them, and performs TL-UL bus transactions to drive the DUT (Fig.13). Our *Hardware Fuzzing Instructions* comprise a grammar (Tbl. 1) that aid syntax-blind coverage-guided greybox fuzzers in generating valid bus-transactions to fuzz hardware.

Table 1: Hardware Fuzzing Grammar.

Opcode	Address Required?	Data Required?	Testbench Action
wait	no	no	advance the clock one period
read	yes	no	TL-UL Get (read)
write	yes	yes	TL-UL PutFullData (write)

Hardware Fuzzing opcode values to testbench actions. We do this for two reasons: 1) a byte is the smallest addressable unit in most software, facilitating the development of utilities to automate generating compact binary seed files (that comply with our grammar) from high-level markdown languages, and 2) choosing a larger opcode field enables adding more opcodes in the future, should we need to support additional operations in the TileLink bus protocol [29]. Second, of the three opcodes we include, not all require address and data fields. Therefore, it is up to the harness to decide how it should process Hardware Fuzzing instructions. While different implementations may choose to read *fixed* size instruction frames, from our empirical analysis in Appendix B, we decide to implement a harness that processes *variable* size instructions frames, depending on the opcode (Table 1).

4.2 Hardware Fuzzing at Scale

To fuzz hardware at scale we design, implement, and open-source a Hardware Fuzzing Pipeline (HWFP) modeled after Google's OSS-Fuzz (Fig. 5). First, our pipeline builds a Docker image (from the Ubuntu 20.04 base image) containing a compiler (LLVM version 12.0.0), RTL simulator (Verilator [64] version 4.0.4), software fuzzer, the target RTL hardware, and a generic fuzzing harness (§4.1.1). From the image, a container is instantiated on a GCP VM that:

1. translates the DUT's RTL to a software model with Verilator [64],
2. compiles/instruments the DUT model, and links it with the generic fuzzing harness (§4.1.1) and simulation engine to create an HSB (Fig. 2),
3. launches the fuzzer for a set period of time, using the `timeout` utility,
4. traces final HDL coverage of fuzzer-generated tests with Verilator [64],
5. saves fuzzing and coverage data to a Google Cloud Storage (GCS) bucket, and lastly
6. tears down the VM.

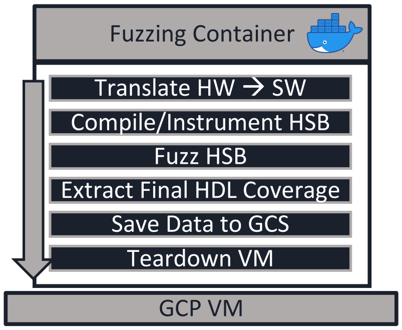

Figure 5: **Hardware Fuzzing Pipeline (HWFP).** We design, implement, and open-source a HWFP that is modeled after Google's OSS-Fuzz [61]. Our HWFP enables us to verify RTL hardware at scale using only open-source tools, a rarity in hardware DV.

Note, for benchmarking, all containers are instantiated on their own GCP `n1-standard-2` VM with two vCPUs, 7.5 GB of memory, 50 GB of disk, running Google's Container-Optimized OS. In our current implementation, we use AFL [91] (version 2.57b) as our fuzzer, but our HWFP is designed to be fuzzer-agnostic.

Unlike traditional hardware verification toolchains, our HWFP uses *only* open-source tools, allowing DV engineers to save money on licenses, and spend it on compute. This not only enhances the deployability of our approach, but makes it ideal for adopting alongside existing hardware DV workflows. This is important because rarely are new DV approaches adopted without some overlap with prior (proven) techniques, since mistakes during hardware verification have costly repercussions.

5 Feasibility Evaluation

In the first part of our evaluation, we address two technical questions around fuzzing software models of RTL hardware with software fuzzers. First, *how should we interface coverage-guided software fuzzers with HSBs?* Unlike most software, HSBs contain other components—a testbench and simulation engine (Fig. 2)—that are *not* the target of testing, yet the fuzzer must learn to manipulate in order to drive the DUT. Second, *how does Hardware Fuzzing compare with traditional dynamic verification methods, i.e., CRV, in terms of time to coverage convergence?* To address this first set of questions, we perform several End-to-End (E2E) fuzzing analyses on over 480 digital lock hardware designs with varying state-space complexities.

5.1 Digital Lock Hardware

In this half of our evaluation, we fuzz various configurations of a digital lock, whose FSM and HDL are shown in Fig. 6

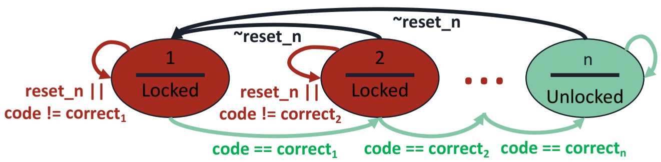

Figure 6: **Digital Lock FSM.** We use a configurable digital lock (FSM shown here) to demonstrate: 1) how to interface software fuzzers with hardware simulation binaries, and 2) the advantages of Hardware Fuzzing (vs. traditional CRV). The digital lock FSM can be configured in two dimensions: 1) total number of states and 2) width (in bits) of input codes.

and List. 1 (Appendix A), respectively. We choose to study this design since the complexity of its state space is configurable, and therefore, ideal for stress testing various DV methodologies. Specifically, the complexity is configurable in two dimensions: 1) the total number of states is configurable by tuning the size, N, of the single state register, and 2) the probability of choosing the correct unlocking code sequence is adjustable by altering the size, M, of the comparator/mux that checks input codes against hard-coded (random) values (List. 1). We develop a utility in Rust, using the `kaze` crate [71], to auto-generate 480 different lock state machines of various complexities, i.e., different values of N, M, and random correct code sequences.

5.2 Digital Lock HSB Architectures

To study these designs, we construct two HSB architectures (Fig. 7) using two hardware DV methodologies: CRV and Hardware Fuzzing. The CRV architecture (Fig. 7A) attempts to unlock the lock through a brute-force approach, where random code sequences are driven into the DUT until the *unlocked* state is reached. If the random sequence fails to unlock the lock, the DUT is reset, and a new random sequence is supplied. If the sequence succeeds, an SVA is violated, which terminates the simulation. The random code sequences are *constrained* in the sense that only valid code sequences are driven into the DUT, i.e., 1) each code in the sequence is in the range $[0, 2^M)$ for locks with M-bit code comparators, and 2) sequences contain exactly $2^N - 1$ input codes for locks with 2^N states. The CRV testbench is implemented with the cocotb [77] framework and simulations are run with Verilator [64].

Alternatively, the Hardware Fuzzing HSB (Fig. 7B) takes input from a software fuzzer that generates code sequences for the DUT. The fuzzer initializes and checkpoints, a process running the HSB (Fig. 2), and repeatedly forks this process and tries various code sequence inputs. If an incorrect code sequence is supplied, the fuzzer forks a new process (equivalent to resetting the DUT) and tries again. If the correct code sequence is provided, an SVA is violated, which the fuzzer registers as a program crash. The difference between CRV and Hardware Fuzzing is that the fuzzer traces coverage during hardware simulation, and will *save* past code sequences

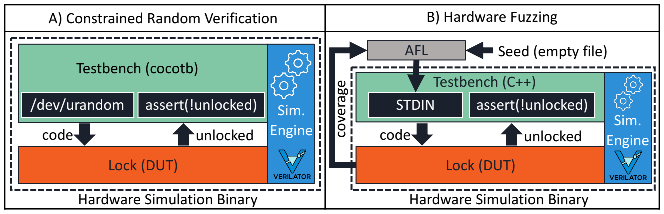

Figure 7: **Digital Lock HSB Architectures.** (A) A traditional CRV architecture: random input code sequences are driven into the DUT until the unlocked state is reached. (B) A software fuzzer generates tests to drive the DUT. The fuzzer monitors coverage of the DUT during test execution and uses this information to generate future tests. Both HSBs are configured to terminate execution upon unlocking the lock using an SVA in the testbench that signals the simulation engine (Fig. 2) to abort.

that get closer to unlocking the lock. These past sequences are then mutated to generate future sequences. Thus, past inputs are used to craft more *intelligent* inputs in the future. To interface the software fuzzer with the HSB, we:

1. implement a C++ testbench harness from Algo. 1 that reads fuzzer-generated bytes from `stdin` and feeds them directly to the `code` input of the lock, and

2. instrument the HSB containing the DUT by compiling it with `afl-clang-fast++`.

5.3 Interfacing Software Fuzzers with Hardware

There are two questions that arise when interfacing software fuzzers with HSBs. First, unlike most software applications, software models of hardware are not standalone binaries. They must be combined—typically by either static or dynamic linking—with a testbench and simulation engine to form an HSB (§2.1.2). Of these three components—DUT, testbench, and simulation engine—we seek to maximize coverage of *only* the DUT. We do not want to waste fuzzing cycles on the testbench or simulation engine. Since coverage tracing instrumentation provides an indirect method to coarsely steer the fuzzer towards components of interest [5], it would be considered good practice to instrument just the DUT portion of the HSB. However, while the DUT is ultimately what we want to fuzz, the fuzzer must learn to use the testbench and simulation engine to manipulate the DUT. Therefore, *what components of the HSB should we instrument to maximize fuzzer performance, yet ensure coverage convergence?*

Second, when simulating hardware, the DUT must be reset to a clean state *before* it can start processing inputs. Traditionally, the testbench portion of the HSB performs this reset by asserting the DUT's global reset signal for a set number of clock cycles. Since the fuzzer instantiates, and repeatedly forks the process executing the HSB, this reset process will happen hundreds, or (potentially) thousands of times per second as each test execution is processed. While some software

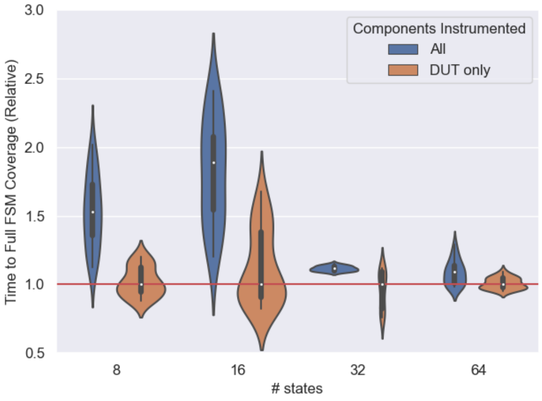

Figure 8: **Instrumentation Level vs. Coverage Convergence Rate.** Distribution of fuzzer run times required to *unlock* various sized digital locks (code widths are fixed at four bits), i.e., achieve ≈ full FSM coverage. For each HSB, we vary the components we instrument for coverage tracing. Run times are normalized to the median DUT-only instrumentation level (orange) across each lock size (red line). While the fuzzer uses the testbench and simulation engine to manipulate the DUT, instrumenting only the DUT *does not* hinder the coverage convergence rate of the fuzzer. Rather, it improves it when DUT sizes are small, compared to the simulation engine and testbench (Fig. 9).

fuzzers [43, 91] enable users to perform initialization operations *before* the program under test is forked—meaning the DUT reset could be performed once, as each forking operation essentially sets the HSB back to a clean state—-this may not always be the case. Moreover, it complicates fuzzer–HSB integration, which contradicts the whole premise of our approach, i.e., low-overhead, design-agnostic CDG. Therefore, we ask: *is this fuzzing initialization feature required to fuzz HSBs?*

5.3.1 Instrumenting HSBs for Fuzzing

To determine the components of the HSB we should instrument, we measure the fuzzing run times to achieve approximate full FSM coverage[4] of several lock designs, i.e., the time it takes the fuzzer to generate a sequence of input codes that *unlocks each lock*. We measure this by modifying the fuzzer to terminate upon detecting the first crash, which we produce using a single SVA that monitors the condition of the *unlocked* signal (List. 1). Specifically, using lock designs with 16, 32, and 64 states, and input codes widths of four bits, we construct HSBs following the architecture shown in Fig. 7B. For each HSB, we vary the components we instrument by using different compiler settings for each component.[5] First, we (naïvely) instrument **all** components, then only the **DUT**.

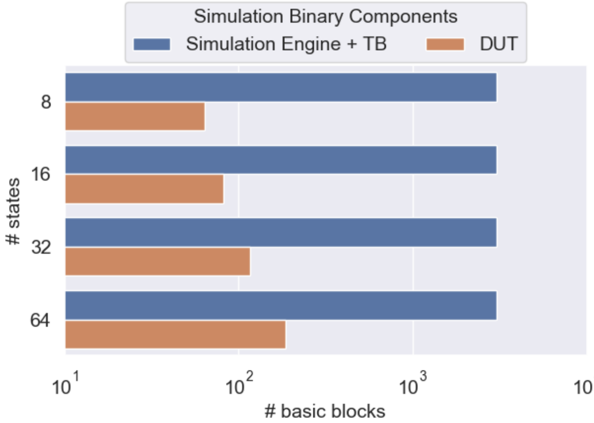

Figure 9: **Basic Blocks per Simulation Binary Component.** We break down the number of basic blocks that comprise the three components within HSBs of different size locks (Fig. 6 & List. 1), generated by Verilator [64]: simulation engine and testbench (TB), and DUT. As locks increase in size, defined by the number of FSM states (code widths are fixed to 4 bits), so do the number of basic blocks in their software model.

Next, we fuzz each HSB 50 times, seeding the fuzzer with an empty file in each experiment.

We plot the distribution of fuzzing run times in Fig. 8. Since fuzzing is an inherently random process, we plot only the middle third of run times across all instrumentation levels and lock sizes. Moreover, all run times are normalized to the median DUT-only instrumentation run times (orange) across each lock size. In addition to plotting fuzzing run times, we plot the number of basic blocks within each component of the HSB in Fig. 9. Across all lock sizes, we observe that only instrumenting the DUT does not handicap the fuzzer, but rather *improves the rate of coverage convergence*! In fact, we perform a Mann-Whitney U test, with a 0.05 significance level, and find all the run-time improvements to be statistically significant. Moreover, we observe that even though the run-time improvements are less significant as the DUT size increases compared to the simulation engine and testbench (Fig. 9), instrumenting only the DUT never handicaps the fuzzer performance.

5.3.2 Hardware Resets vs. Fuzzer Performance

To determine if DUT resets present a performance bottleneck, we measure the degradation in fuzzing performance due to the repeated simulation of DUT resets. We take advantage of a unique feature of a popular greybox fuzzer [91] that enables configuring the exact location of initializing the *fork server*.[6] This enables the fuzzer to perform any program-specific initialization operations *once*, prior to forking children processes to fuzz. Using this feature, we repeat the same fuzzing run

[4] We use the term *approximate* when referring to *full FSM coverage*, since we are not excising the lock's reset state transitions (Fig. 6) in these experiments.

[5] Verilator conveniently contains each component—DUT, testbench, and simulation engine—in separate C++ files, so each file can be compiled with separate settings (i.e., with or without coverage tracing instrumentation).

[6] By default, AFL [91] instantiates a process from the binary under test, pauses it, and repeatedly forks it to create identical processes to feed test inputs to. The component of AFL that performs process forking is known as the *fork server*.

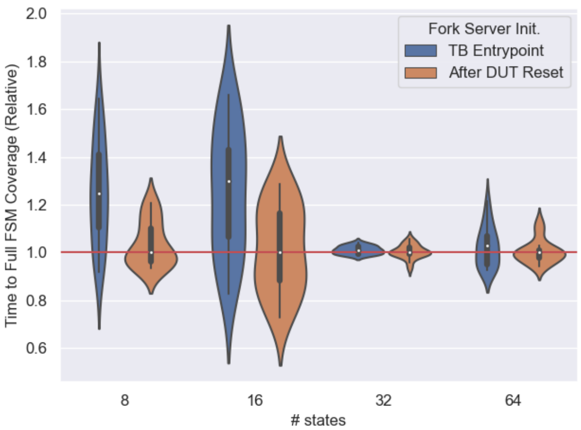

Figure 10: **Hardware Resets vs. Fuzzer Performance.** Fuzzing run times across across digital locks (similar to Fig. 8) with different fork server initialization locations in the testbench to eliminate overhead due to the repeated simulation of hardware DUT resets. DUT resets are only a fuzzing bottleneck when DUTs are small, reducing fuzzer–HSB integration complexity.

time analysis performed in §5.3.1, except we instrument all simulation binary components, and compare two variations of the digital lock HSB shown in Fig. 7B. In one testbench, we use the default fork server initialization location: at the start of `main()`. In the other testbench, we initialize the fork server *after* the point where the DUT has been reset.

Fig. 10 shows our results. Again, we drop outliers by plotting only the middle third of run times across all lock sizes and fork server initialization points. Additionally, we normalize all run times to the median "after DUT reset" run times (orange) across each lock size. From these results, we apply the Mann-Whitney U test (with 0.05 significance level) between run times. This time, only locks with 8 and 16 states yield p-values less than 0.05. This indicates the overhead of continuously resetting the DUT during fuzzing diminishes as the DUT increases in complexity.[7] Additionally, we note that even the largest digital locks we study (64 states), are smaller than the smallest OpenTitan core, the RISC-V Timer, in terms of number of basic blocks in the software model (Fig. 9 & Table 2).

5.4 Hardware Fuzzing vs. CRV

Using the techniques we learned from above, we perform a run-time comparison analysis between Hardware Fuzzing and CRV,[8] the current state-of-the-art hardware dynamic verification technique. We perform these experiments using digital locks of various complexities, from 2 to 64 states, and code widths of 1 to 8 bits. The two HSB architectures we compare are shown in Fig. 7, and discussed in §5.2. Note the fuzzer was again seeded with an empty file to align its starting state with the CRV tests.

Similar to our instrumentation and reset experiments (§5.3) we measure the fuzzing *run times* required to achieve ≈ full FSM coverage of each lock design, i.e., the time to *unlock each lock*. We illustrate these run times in heatmaps shown in Fig. 11. We perform 20 trials for each experiment and average these run times in each square of a heatmap. While the difference between the two approaches is indistinguishable for extremely small designs, the advantages of Hardware Fuzzing become apparent as designs increase in complexity. For medium to larger lock designs, Hardware Fuzzing achieves full FSM coverage faster than CRV by over two orders-of-magnitude, even when the fuzzer is seeded with an empty file. Moreover, many CRV experiments were terminated early (after running for five days) to save money on GCP instances.

6 Practicality Evaluation

In the second part of our evaluation, we address two remaining questions. First, *how does Hardware Fuzzing compare with prior RTL fuzzing schemes, e.g., RFUZZ [39], in terms of HDL code coverage?* While Laeufer *et al.* were the first to demonstrate fuzzing RTL with `RFUZZ` [39], we argue for an entirely different approach (Fig. 1), fuzzing software models of RTL hardware, rather than the RTL hardware itself. Lastly, *how does Hardware Fuzzing perform in practice commercial-grade hardware IP?* To address these questions, we perform E2E fuzzing analyses on several open-source hardware designs, including five commercial-grade cores from Google's OpenTitan [44] SoC, four SiFive TileLink peripherals, three RISC-V CPUs, and an FFT accelerator.

6.1 Hardware Fuzzing vs. RFUZZ

Unlike our approach, `RFUZZ` instruments RTL hardware directly by injecting coverage-tracing hardware into the RTL when it is compiled from a high-level HDL, like FIRRTL, to Verilog. Moreover, `RFUZZ` does not exploit any bus-specific harnesses, rather, it generates design-specific harnesses that are fed fuzzer-generated bit-vectors to hardware input ports, as described in Algo. 1 and demonstrated in the fuzzing harness built for the digital lock in Fig. 7b.

To demonstrate the differences between our approach and `RFUZZ`, we compare the HDL line coverage achieved by both approaches over the course of fuzzing eight different hardware designs for 24 hours. Specifically, we fuzz the same eight hardware designs in the original `RFUZZ` paper [39], including the I2C, SPI, PWM, and UART SiFive TileLink IP blocks [63], three RISC-V Sodor CPUs [59], and an FFT accelerator [58].

[7] While the appearance of Fig. 10 may allude that when the number of states is 32, forking after DUT resets takes longer than forking at the testbench entry point, deeper analysis reveals the variance makes this appear so (the "forking after reset" median is still lower). Regardless, the main takeaway from Fig. 10 is for large designs, the DUT reset overheads are not a bottleneck.

[8] CRV is widely deployed in any DV testbenches built around the cocotb [77] or UVM [1] frameworks, e.g., all OpenTitan [44] IP core testbenches.

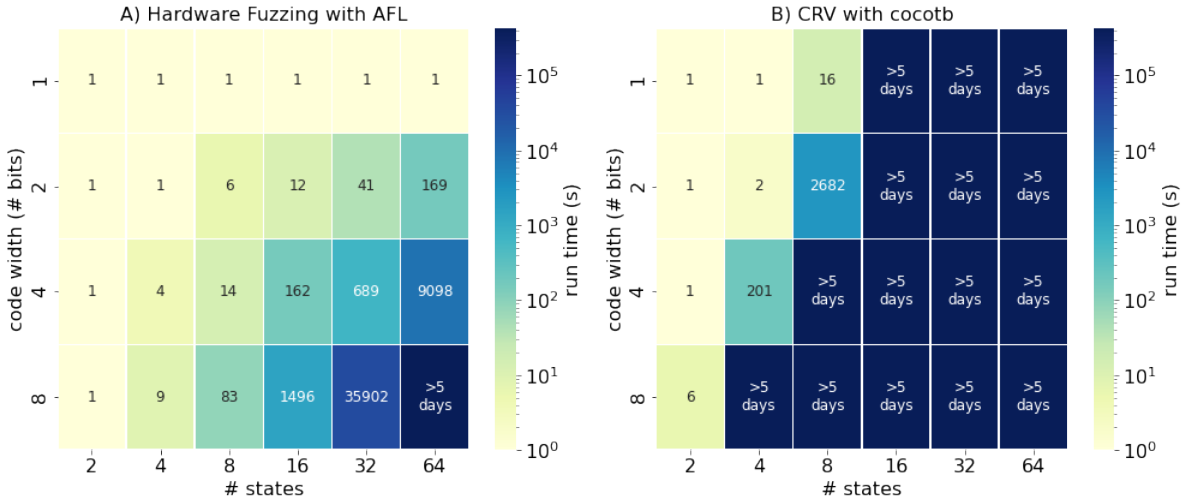

Figure 11: **Hardware Fuzzing vs. CRV.** Run times for both Hardware Fuzzing (A) and CRV (B) to achieve ≈ full FSM coverage of various digital lock (Fig. 6) designs—i.e., time to unlock the lock—using the testbench architectures shown in Fig. 7. Run times are averaged across 20 trials for each lock design—defined by a (# states, code width) pair—and DV method combination. Across these designs, Hardware Fuzzing achieves full FSM coverage faster than traditional CRV approaches, by over two orders of magnitude.

For each core, we use the same RFUZZ-generated test harness across both approaches, but use different fuzzing mechanisms, as highlighted in Fig. 1. Specifically, RFUZZ uses a custom fuzzer (very similar to AFL[12]) that directly measures RTL coverage using Verilog-level instrumentation, while our (Hardware Fuzzing) approach uses a software fuzzer (i.e., AFL) that measures RTL coverage using HSB-level instrumentation.

Since RFUZZ provides a command-line option to seed the fuzzer with zero-level input signals for a provided number of clock cycles, we perform several microbenchmarks to compare the effects of varying this parameter across both fuzzing setups. Specifically, for each core, we perform five trials with both fuzzing techniques, using seed inputs that translate to holding all DUT input signals at a logical zero for one, three, and five clock cycles, and compare the results. To measure our worst case performance vs. RFUZZ, we select the *best-case* RFUZZ results (i.e., highest coverage across all trials), and compare them with the *worst-case* Hardware Fuzzing results (i.e., lowest coverage across all trials). Our results are plotted in Fig. 12. After 24 hours of fuzzing, across all cores and seeds, the average HDL line coverage improvement using our Hardware Fuzzing approach over RFUZZ was 26.70%, while the minimum and maximum improvements are 14.82% and 42.64%, respectively. Lastly, we apply the Mann-Whitney U test (with 0.05 significance level) between all fuzzing trials across all cores, and observe p-values less than 0.05.

6.2 Fuzzing OpenTitan IP

To address the last question—*How does Hardware Fuzzing perform in practice on commercial-grade hardware?*—we fuzz five IP blocks from Google's OpenTitan silicon root-of-trust SoC [44], including the: AES, HMAC, KMAC, RISC-V Timer, and Alert Handler cores. While each core performs different functions,[9] they all conform to the OpenTitan *Comportability Specification* [17], implying **they are all controlled via reads and writes to memory-mapped registers over a TL-UL bus**. By adhering to a uniform bus protocol, we are able to re-use a generic fuzzing harness (Fig. 13), facilitating the deployability of our approach. Below, we highlight the functionality of each IP core. Additionally, in Table 2, we report the complexity of each IP core in both the hardware and software domains, in terms of Lines of Code (LOC), number of basic blocks, and number of SVAs provided in each core's HDL. Software models of each hardware design are produced using Verilator, as we describe in §3.1.

Table 2: OpenTitan IP Core Complexity in HW and SW Domains.

IP Core	HW LOC	SW LOC	# Basic Blocks*	# SVAs†
AES	4,562	38,036	3,414	53
Alert Handler	4,198	16,260	2,920	34
HMAC	2,695	18,005	1,764	30
KMAC	4,585	119,297	6,996	44
RV Timer	677	3,111	290	8

* # of basic blocks in compiled software model with O3 optimization.
† # of SystemVerilog Assertions included in IP HDL at time of writing.

6.2.1 Fuzzing OpenTitan IP with Empty Seeds

Unlike most software applications that are fuzzed [61], we observe that software models of hardware are quite small (Table 2). So, we decided to experiment fuzzing OpenTitan

[9] For more information on the functionalities of each OpenTitan IP block, see https://docs.opentitan.org/hw/ip/.

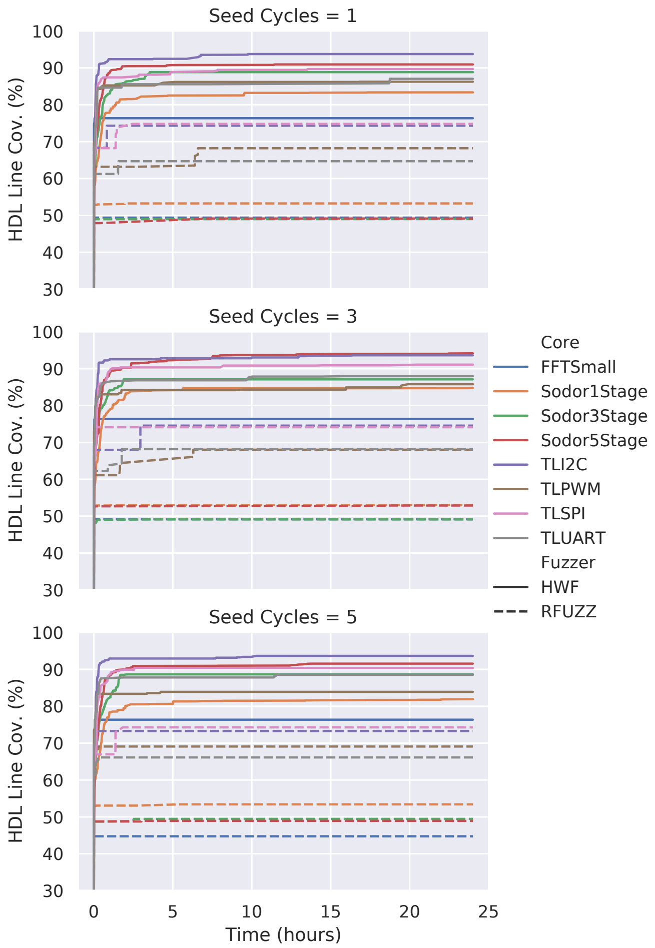

Figure 12: **Hardware Fuzzing vs. RFUZZ.** We fuzz eight different hardware designs, including an FFT accelerator, RISC-V CPUs, and TileLink peripherals, with our Hardware Fuzzing approach vs. RFUZZ [39] (Fig. 1), using input seeds that provide zero-level input signals to the DUTs for one, three, and five clock cycles. Across all cores and input seed configurations, our approach yields 26.70% better HDL coverage on average (than RFUZZ) after 24 hours of fuzzing.

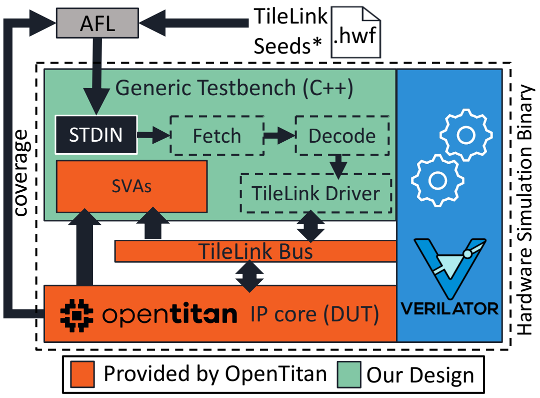

Figure 13: **OpenTitan HSB Architecture.** A software fuzzer learns to generate fuzzing *instructions* (Fig. 4)—from .hwf seed files—based on a hardware fuzzing grammar (§4.1.2). It pipes these instructions to stdin where a generic C++ fuzzing harness fetches/decodes them, and performs the corresponding TileLink bus operations to drive the DUT. SVAs are evaluated during execution of the HSB, and produce a program crash (if violated), that is caught and reported by the software fuzzer.

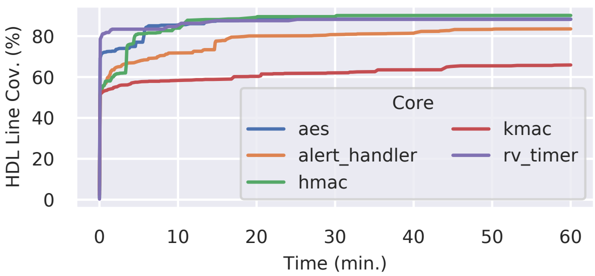

Figure 14: **Coverage vs. Time Fuzzing with Empty Seeds.** Fuzzing five OpenTitan [44] IP cores for one hour, seeding the fuzzer with an empty file in each case, yields over 83% HDL line coverage in three out of four designs.

cores using a single empty seed file as starting input, this time for only one hour. We plot the results of this experiment in Fig. 14. After only one hour of fuzzing with no proper starting seeds, we achieve over 83% HDL line coverage across AES, Alert Handler, HMAC, and RV Timer cores, and over 65% coverage of the KMAC core.

6.2.2 Fuzzing for Bugs in OpenTitan IPs

While coverage is an important metric, the ultimate goal of fuzzing hardware is to automatically uncover bugs, before they percolate into fabricated silicon. Therefore, in our final evaluation, we demonstrate the effectiveness of Hardware Fuzzing at finding five RTL bugs, one in each OpenTitan IP block we study. Specifically, in the AES, Alert Handler, HMAC, and RV Timer IPs, we implant four of the same synthetic bugs used in the Hack@DAC competition [20], and for the KMAC IP, we attempt to re-detect a bug found in the wild, that was reported on the OpenTitan public GitHub (Issue #6408).[10] Across all IPs, we craft generic SVAs to produce HSB crashes upon encountering out-of-spec hardware behaviors. These include SVAs to check FSM transitions, lock registers, and other functional behaviors of an IP. In Table 3, we describe the bugs in each IP, and for each IP except the KMAC,[11] we list the corresponding Hack@DAC bug number [20]. Additionally, in Table 3, we list the time it took our fuzzer to detect each bug when seeded with a set of inputs that simply resets and initializes each DUT to perform its prescribed tasks. Namely, for the AES, our seed configures the device to operate in CTR mode, for the Alert Handler, our seed does nothing after resetting the device, for the HMAC, our seed configures the device to perform SHA256 hashes, for the KMAC, our seed configures the device to perform KMAC operations in cSHAKE hashing mode, and lastly, for the RV

[10] https://github.com/lowRISC/opentitan/issues/6408

[11] The KMAC bug was not a Hack@DAC bug that was intentionally implanted, rather, it was a *real* bug. See link above.

Table 3: Hardware Fuzzing RTL Bug Discovery Times.

IP	Bug Description	Hack@DAC Bug #	Discovery Time (s)
AES	FSM bug causes DoS.	18	65.1
Alert Handler	Lock writable by software.	4	396.5
HMAC	Incorrect padding produces insecure hash.	19	46.9
KMAC	FSM bug skips entropy state refresh.	N/A[11]	35548
RV Timer	Faulty logic causes inaccurate time interrupt tick.	15	5.1

Timer, our seed arms the timer. Across each core we study, we are able to detect four out of five bugs in less than 10 minutes, and all bugs in less than 10 hours, with initialization seeds that are orders-of-magnitude less complex than conventional dynamic verification testbenches.

7 Discussion

Detecting Bugs During Fuzzing. The focus of Hardware Fuzzing is to provide a scalable yet flexible solution for integrating CDG with hardware simulation. However, *test generation* and *hardware simulation* comprise only two-thirds of the hardware verification process (§2.1). The final, and arguably most important, step is detecting incorrect hardware behavior, i.e., *test evaluation* in §2.1.3. For this there are two approaches: 1) invariant checking and 2) (gold) model checking. In both cases, we trigger HSB crashes upon detecting incorrect hardware behavior, which software fuzzers log. For invariant checks, we use SVAs that send the HSB process the `SIGABRT` signal upon assertion violation (demonstrated in §6.2.2). Likewise, for gold model checking testbenches any mismatches between models results in a `SIGABRT`.

Developing Additional Bus Protocols. To provide a design-agnostic interface to fuzz RTL hardware, we develop a design-agnostic testbench harness (Fig. 13). Our harness decodes fuzzer-generated tests using a bus-specific grammar (§4.1.2), and produces corresponding TL-UL bus transactions that drive a DUT. In our current implementation, our generic testbench harness conforms to the TL-UL bus protocol [29]. As a result, we can fuzz any IP core that speaks the same bus protocol (e.g., all OpenTitan cores [44]). To fuzz cores that speak other bus protocols (e.g., Wishbone, AMBA, Avalon, etc.), users can simply write a new harness for the bus they wish to support.

Writing a new bus harness requires developing an API with two (or more) functions that represent possible bus transactions that the fuzzing harness (testbench) can initiate. These functions toggle the bus interface signals in the correct order, to complete a bus transaction, e.g., reading/writing to/from a specific address. For the TL-UL bus protocol we study, only a *read* (*Get* in TL-UL terms) and *write* (*Put* in TL-UL terms) function are required. Implementing these functions took 380 lines of C++ code (including supporting debug code). For more details on how to implement your own fuzzing harness based off our TL-UL harness, we refer you to our open-source codebase.

Hardware without a Bus Interface. For hardware designs that perform I/O over a generic set of ports that do not conform to any bus protocol, we provide a generic testbench harness that maps fuzzer-generated input files across spatial and temporal domains by interpreting each fuzzer-generated file as a sequence of DUT inputs (Algo. 1). We demonstrate this Hardware Fuzzing configuration when fuzzing various digital locks (Fig. 7B) and the RFUZZ cores (Fig. 12). Specifically, RFUZZ automatically generates a testbench harness for each DUT that takes as input a byte array each clock cycle, and maps the bits in the array to the inputs of the DUT. Our generic testbench harness then wraps the RFUZZ-generated testbench, reading input bytes generated by the fuzzer, and routing them directly to the (auto-generated) RFUZZ testbench byte array.

Note, if any DUT inputs require structural dependencies, we recommend developing a grammar and corresponding testbench—similar to our bus-specific grammar (§4.1.2)—to aid the fuzzer in generating valid test cases. Designers can use the lessons in this paper to guide their core-specific grammar designs.

Limitations. While Hardware Fuzzing is both efficient and design-agnostic, there are some limitations. First, unlike software, there is no notion of a *hardware sanitizer*, that can add safeguards against generic classes of hardware bugs for the fuzzer to sniff out. While we envision hardware sanitizers being a future active research area, for now, DV engineers must create invariants or gold models to check design behavior against for the fuzzer to find crashing inputs. Second, there is no notion of analog behavior in RTL hardware, let alone in translated software models. In its current implementation, Hardware Fuzzing is not effective against detecting side-channel vulnerabilities that rely on information transmission/leakage through analog domains.

8 Related Work

There are two categories of prior CDG approaches: 1) design-agnostic and 2) design-specific.

Design-Agnostic. Laeufer *et al.* 's `RFUZZ` [39] is the most relevant prior work, which attempts to build a full-fledged design-agnostic RTL fuzzer. To achieve their goal, they propose a new RTL coverage metric—*mux toggle coverage*—that measures if the control signal to a 2:1 multiplexer expresses both states (0 and 1). Like our approach, they use a fuzzer very similar to AFL.[12] Unlike our approach, they instrument

[12] According to their GitHub repository, the RFUZZ fuzzer, called *kfuzz*, is mostly a re-implementation of AFL in the Rust programming language [38].

the RTL directly, by injecting additional HDL into the design. Unfortunately, this has two drawbacks. First, to instrument the RTL for coverage tracing, the authors of RFUZZ develop a custom optimization pass on top of the FIRRTL compiler. However, the FIRRTL compiler, takes as input FIRRTL code, and translates it to Verilog. This implies that their coverage-tracing instrumentation tooling is only compatible with hardware designs described in FIRRTL, or an HDL that is easily translated to FIRRTL, e.g., Chisel.[13] Unfortunately, translating (System)Verilog designs to FIRRTL is non-trivial, and experimental at best [8]. Second, RFUZZ requires some designs be modified to have reset times on the order of one to two clock cycles, since designs must be reset between test executions, and slow resets can lead to poor fuzzing performance.

Similarly, Gent *et al.* [22] also propose an automatic test pattern generator based on custom coverage coverage monitors injected into the RTL. However, given their coverage tracing methods, Laeufer *et al.* [39] question the scalability of their approach to larger designs.

Design-Specific. Unlike the *design-agnostic* approaches, several researchers propose CDG techniques exclusively for processors. Zhang *et al.* [92] propose Coppelia, a tool that uses a custom symbolic execution engine (built on top of KLEE [9]) on software models of the RTL. Coppelia's goal is to target specific security-critical properties of processors; Hardware Fuzzing enables combining such static methods with fuzzing (i.e., concolic execution [66]) for free, overcoming the limits of symbolic execution alone. Hur *et al.* [28] propose DIFUZZRTL that combines RFUZZ with golden model checking to find bugs in CPUs. However, Hardware Fuzzing produces better coverage than RFUZZ (§6.1), and can be combined with invariant *or* with golden model checking to detect bugs. Lastly, two other processor-specific CDG approaches are Squillero's *MicroGP* [65] and Bose *et al.* 's [6] that use a genetic algorithms to generate random assembly programs that maximize RTL code coverage of a processor. Unlike Hardware Fuzzing, these approaches require custom DUT-specific grammars to build assembly programs from.

9 Conclusion

Hardware Fuzzing is an effective solution to CDG for hardware DV. Unlike prior work, we take advantage of feature-rich software testing methodologies and tools, to solve a long-standing problem in hardware DV. To make our approach attractive to DV practitioners, we solve several key deployability challenges, including developing generic interfaces (grammar & testbench) to fuzz RTL in a design-agnostic manner. Using our generic grammar and testbench, we show that our Hardware Fuzzing approach can achieve over 83% HDL code coverage across four of the five OpenTitan IPs we study in only one hour, with no knowledge of the DUT design or implementation. Moreover, we demonstrate that approach can also detect various real and implanted bugs in the same designs, in less than 10 hours. Finally, compared to standard dynamic verification practices and prior RTL fuzzing techniques [39], with Hardware Fuzzing, we achieve over two orders-of-magnitude and 26.70% coverage convergence improvements, respectively.

[13] This is further confirmed by reviewing their GitHub project [38], which only contains hardware designs written in FIRRTL.

Acknowledgment

We thank Scott Johnson, Srikrishna Iyer, Rupert Swarbrick, Pirmin Vogel, Philipp Wagner, and other members of the OpenTitan project for their technical expertise that enabled us to demonstrate our approach on OpenTitan silicon IP.

The work reported in this paper was partially supported by the National Science Foundation under Grant CNS-1646130, the Army Research Office under Grant W911NF-21-1-0057, and the Defense Advanced Research Projects Agency. The content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. Approved for public release; distribution is unlimited.

References

[1] Accellera. Universal Verification Methodology (UVM). https://www.accellera.org/downloads/standards/uvm.

[2] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for deep bugs with grammars. In *Network and Distributed Systems Security Symposium (NDSS)*, 2019.

[3] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[4] 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 *Design Automation Conference (DAC)*. IEEE, 2012.

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 2017.

[6] Mrinal Bose, Jongshin Shin, Elizabeth M Rudnick, Todd Dukes, and Magdy Abadir. A genetic approach to automatic bias generation for biased random instruction generation. In *Proceedings of the Congress on Evolutionary Computation*. IEEE, 2001.

[7] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013.

[8] Jean Bruant, Pierre-Henri Horrein, Olivier Muller, Tristan Groleat, and Frédéric Pétrot. (system) verilog to chisel translation for faster hardware design. In *International Workshop on Rapid System Prototyping (RSP)*. IEEE, 2020.

[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[10] Cadence Design Systems. Xcelium Logic Simulation. https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html.

[11] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant CPUs. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[12] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[13] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[14] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[15] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[16] Marek Cieplucha. Metric-driven verification methodology with regression management. *Journal of Electronic Testing*, 2019.

[17] lowRISC Contributors. OpenTitan: Comportability Definition and Specification, November 2020. https://docs.opentitan.org/doc/rm/comportability_specification/.

[18] MITRE Corporation. Cve details: Intel: Vulnerability statistics, August 2019. https://www.cvedetails.com/vendor/238/Intel.html.

[19] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 1974.

[20] 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 Security Symposium*, 2019.

[21] Shai Fine and Avi Ziv. Coverage directed test generation for functional verification using bayesian networks. In *the 40th annual Design Automation Conference (DAC)*, 2003.

[22] Kelson Gent and Michael S Hsiao. Fast multi-level test generation at the rtl. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2016.

[23] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. *Queue*, 2012.

[24] Onur Guzey and Li-C Wang. Coverage-directed test generation through automatic constraint extraction. In *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2007.

[25] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *the 37th annual International Symposium on Computer Architecture (ISCA)*, 2010.

[26] Jesse Hertz and Tim Newsham. ProjectTriforce: AFL/QEMU fuzzing with full-system emulation. https://github.com/nccgroup/TriforceAFL.

[27] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. PANGOLIN: Incremental hybrid fuzzing with polyhedral path abstraction. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[28] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. DIFUZZRTL: Differential fuzz testing to find cpu bugs. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[29] SiFive Inc. SiFive: TileLink Specification, November 2020. Version 1.8.0.

[30] Charalambos Ioannides, Geoff Barrett, and Kerstin Eder. Introducing xcs to coverage directed test generation. In *IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2011.

[31] James Johnson. gramfuzz. https://github.com/d0c-s4vage/gramfuzz.

[32] Jing-Yang Jou and Chien-Nan Jimmy Liu. Coverage analysis techniques for hdl design validation. *Proceedings of Asia Pacific CHip Design Languages*, 1999.

[33] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. Motivation for and evaluation of the first tensor processing unit. *IEEE Micro*, 2018.

[34] Simon Kagstrom. kcov. https://github.com/SimonKagstrom/kcov.

[35] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, et al. Replacing testing with formal verification in intel core i7 processor execution engine validation. In *International Conference on Computer Aided Verification (CAV)*. Springer, 2009.

[36] Tae Kim. Intel's alleged security flaw could cost chipmaker a lot of money, Bernstein says. https://www.cnbc.com/2018/01/03/intels-alleged-security-flaw-could-cost-chipmaker-a-lot-of-money-bernstein.html.

[37] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[38] Kevin Laeufer. rfuzz: coverage-directed fuzzing for rtl research platform. https://github.com/ekiwi/rfuzz.

[39] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. RFUZZ: coverage-directed fuzz testing of RTL on FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018.

[40] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization (CGO)*, San Jose, CA, USA, 2004.

[41] Patrick S Li, Adam M Izraelevitz, and Jonathan Bachrach. Specification for the FIRRTL language. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-9*, 2016.

[42] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security'18)*, 2018.

[43] LLVM Project. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

[44] lowRISC. Opentitan. https://opentitan.org/.

[45] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. ASIC clouds: Specializing the datacenter. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 178–190. IEEE, 2016.

[46] Mentor Graphics. ModelSim. https://www.mentor.com/products/fv/modelsim/.

[47] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. TPM-FAIL: TPM meets Timing and Lattice Attacks. In *29th USENIX Security Symposium (USENIX Security'20)*, 2020.

[48] Gordon E Moore. Cramming more components onto integrated circuits, 1965.

[49] Mozilla Security. Dharma: A generation-based, context-free grammar fuzzer. https://www.overleaf.com/project/5e163844e63c070001079faa.

[50] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Symposium on Security and Privacy (S&P)*. IEEE, 2019.

[51] Tony Nowatzki, Vinay Gangadhan, Karthikeyan Sankaralingam, and Greg Wright. Pushing the limits of accelerator efficiency while retaining programmability. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[52] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided greybox fuzzing. In *USENIX Security Symposium*, 2020.

[53] Vern Paxson, Will Estes, and John Millaway. Lexical analysis with flex. *University of California*, 2007.

[54] Peach Tech. Peach Fuzzing Platform. https://www.peach.tech/products/peach-fuzzer/.

[55] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: fuzzing by program transformation. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[56] Andrew Piziali. *Functional verification coverage measurement and analysis*. Springer Science & Business Media, 2007.

[57] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *Network and Distributed Systems Security Symposium (NDSS)*, 2017.

[58] UC Berkeley Architecture Research. Pipelined FFT. https://github.com/ucb-art/fft.

[59] UC Berkeley Architecture Research. RISC-V Sodor. https://github.com/ucb-bar/riscv-sodor.

[60] Kosta Serebryany. Continuous fuzzing with libfuzzer and addresssanitizer. In *Cybersecurity Development (SecDev)*. IEEE, 2016.

[61] Kostya Serebryany. OSS-Fuzz - google's continuous fuzzing service for open source software. In *USENIX Security Symposium*, 2017.

[62] Sophia Shao and Emma Wang. Die photo analysis. http://vlsiarch.eecs.harvard.edu/research/accelerators/die-photo-analysis/.

[63] SiFive. SiFive Blocks. https://github.com/sifive/sifive-blocks.

[64] Wilson Snyder. verilator. https://www.veripool.org/wiki/verilator.

[65] Giovanni Squillero. Microgp—an evolutionary assembly program generator. *Genetic Programming and Evolvable Machines*, 2005.

[66] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed Systems Security Symposium (NDSS)*, 2016.

[67] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[68] Robert Swiecki. honggfuzz. https://honggfuzz.dev/.

[69] Synopsys. VCS. https://www.synopsys.com/verification/simulation/vcs.html.

[70] Serdar Tasiran and Kurt Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design & Test of Computers*, 2001.

[71] Jake Taylor. Kaze: an HDL embedded in Rust, November 2020. https://docs.rs/kaze/0.1.13/kaze/.

[72] Marat Teplitsky, Amit Metodi, and Raz Azaria. Coverage driven distribution of constrained random stimuli. In *Proceedings of the Design and Verification Conference (DVCon)*, 2015.

[73] The GNU Project. Bison. https://www.gnu.org/software/bison/.

[74] Timothy Trippel, Kang G. Shin, Kevin B. Bush, and Matthew Hicks. Bomberman: Defining and defeating hardware ticking timebombs at design-time. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[75] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security'18)*, 2018.

[76] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[77] Potential Ventures. cocotb. https://github.com/cocotb/cocotb.

[78] Martin Vuagnoux. Autodaf´e: an act of software torture. http://autodafe.sourceforge.net/tutorial/index.html.

[79] Dmitry Vyukov. syzkaller. https://github.com/google/syzkaller.

[80] Fanchao Wang, Hanbin Zhu, Pranjay Popli, Yao Xiao, Paul Bodgan, and Shahin Nazarian. Accelerating coverage directed test generation for functional verification: A neural network-based framework. In *Proceedings of the Great Lakes Symposium on VLSI*, 2018.

[81] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2017.

[82] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019.

[83] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Tim Cheng. *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009.

[84] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[85] Stephen Williams. Icarus Verilog. http://iverilog.icarus.com/.

[86] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. A2: Analog malicious hardware. In *IEEE Symposium on Security and Privacy (SP)*, 2016.

[87] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[88] Jun Yuan, Carl Pixley, Adnan Aziz, and Ken Albin. A framework for constrained functional verification. In *International Conference on Computer Aided Design (ICCAD)*. IEEE, 2003.

[89] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *{USENIX} Security Symposium*, 2018.

[90] Michael Zalewski. afl-fuzz: making up grammar with a dictionary in hand. https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html.

[91] Michael Zalewski. American fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[92] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. End-to-end automated exploit generation for validating the security of processor designs. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018.

[93] Rui Zhang and Cynthia Sturton. A recursive strategy for symbolic execution to find exploits in hardware designs. In *ACM SIGPLAN International Workshop on Formal Methods and Security (FSM)*, 2018.

[94] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *USENIX Security Symposium*, 2020.

A Digital Lock HDL

Listing 1: SystemVerilog of Lock with $N = log2(\#states)$ and M-bit codes.

```
module lock(
  input  reset_n ,
  input  clk ,
  input  [M−1:0] code,
  output  unlocked
);
logic  [N−1:0]  state ;
logic  [M−1:0] correct_codes  [N];

//  Secret  codes  set  to  random values
for  (genvar  i  =  0;  i  < N;  i++)  begin  :  secret_codes
  assign  correct_codes [ i ]  = <random value>;
end

assign  unlocked = ( state  == ’1)  ?  1’b1  :  1’b0;

always  @(posedge clk)  begin
  if  (! reset_n )  begin
    state  <= ’0;
  end  else  if  (!unlocked  && code == correct_codes[ state  ])
       begin
    state  <=  state  +  1’b1;
  end  else  begin
    state  <=  state ;
  end
end
endmodule
```

B Optimizing the Hardware Fuzzing Grammar

Recall, to facilitate widespread adoption of Hardware Fuzzing we design a generic testbench fuzzing harness that decodes a grammar and performs corresponding TL-UL bus transactions to exercise the DUT (Fig. 13). However, there are implementation questions surrounding how the grammar should be decoded (§4.1.2):

1. *How should we decode 8-bit opcodes when the opcode space defines less than 2^8 valid testbench actions?*
2. *How should we pack Hardware Fuzzing instruction frames that conform to our grammar?*

B.1 Opcode Formats

In its current state, we define three opcodes in our grammar that correspond to three actions our generic testbench can perform (Table 1): 1) **wait** one clock cycle, 2) TL-UL **read**, and 3) TL-UL **write**. However, we chose to represent these opcodes with a single byte (Fig. 4). Choosing a larger field than necessary has implications regarding the fuzzability of our grammar. In its current state, 253 of the 256 possible opcode values may be useless depending on how they are decoded by the testbench. Therefore we propose, and empirically study, two design choices for decoding Hardware Fuzzing opcodes into testbench actions:

- **Constant**: *constant* values are used to represent each opcode corresponding to a single testbench action. Remaining opcode values are decoded as *invalid*, and ignored.
- **Mapped**: equal sized ranges of opcode values are *mapped* to valid testbench actions. No invalid opcode values exist.

B.2 Instruction Frame Formats

Of the three actions our testbench can perform—wait, read, and write—some require additional information. Namely, the TL-UL read action requires a 32-bit address field, and the TL-UL write action requires 32-bit data and address fields. Given this, there are two natural ways to decode Hardware Fuzzing instructions (Fig. 4):

- **Fixed**: a *fixed* instruction frame size is decoded regardless of the opcode. Address and data fields could go unused depending on the opcode.
- **Variable**: a *variable* instruction frame size is decoded. Address and data fields are only appended to opcodes that correspond to TL-UL read and write testbench actions. No address/data information goes unused.

B.3 Results

To determine the optimal Hardware Fuzzing grammar, we fuzz four OpenTitan IP blocks—the AES, HMAC, KMAC, and RV-Timer—for 24 hours using all combinations of opcode and instruction frame formats mentioned above. For each core we seed the fuzzer with 8–12 binary Hardware Fuzzing seed files (in the corresponding Hardware Fuzzing grammar) that correctly drive each core, with the exception of the RV-Timer core, which we seed with a single wait operation instruction due to its simplicity. For each experiment, we extract and plot three DUT coverage metrics over fuzz times in Fig. 15. These metrics include: 1) line coverage of the DUT software model,

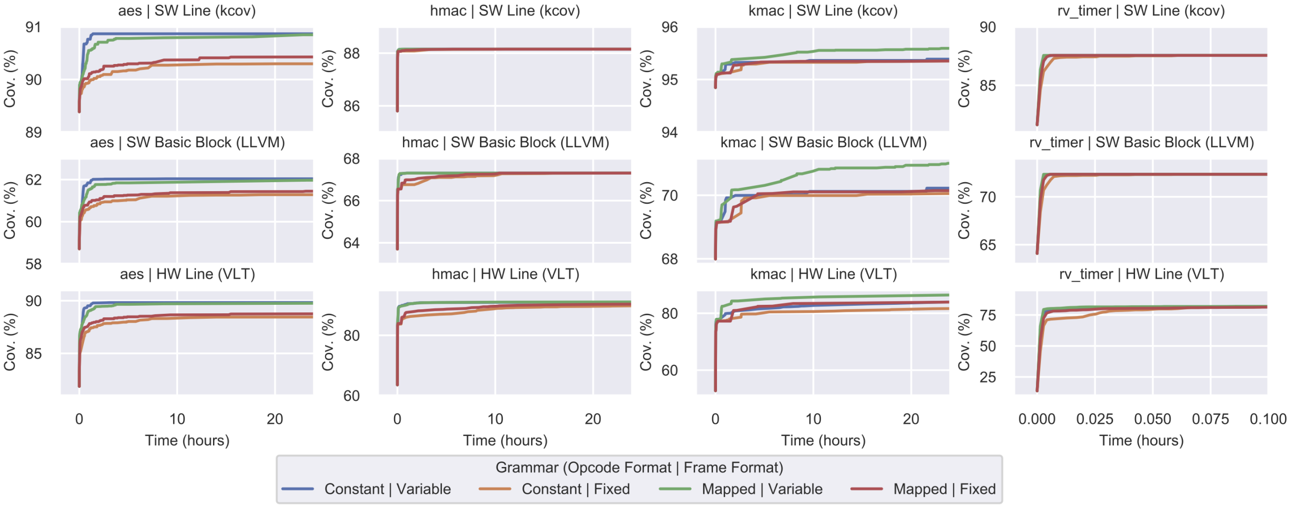

Figure 15: **Coverage Convergence vs. Hardware Fuzzing Grammar.** Various software and hardware coverage metrics over fuzzing time across four OpenTitan [44] IP cores and hardware fuzzing grammar variations (§B). In the first row, we plot *line coverage* of the software models of each hardware core computed using `kcov`. In the second row, we plot *basic block coverage* computed using `LLVM`. In last row, we plot HDL line coverage (of the hardware itself) computed using `Verilator` [64]. From these results we formulate two conclusions: 1) coverage in the software domain correlates to coverage in the hardware domain, and 2) the Hardware Fuzzing grammar with *variable* instruction frames is best for greybox fuzzers that prioritize small test files.

2) basic block coverage of the same, and 3) line coverage of the DUT's HDL. Software line coverage is computed using `kcov` [34], software basic block coverage is computed using `LLVM` [40], and hardware line coverage is computed using `Verilator` [64]. Since we perform 10 repetitions of each fuzzing experiment, we average and consolidate each coverage time series into a single trace.

From these results we draw two conclusions. First, *variable* instruction frames seem to perform better than fixed frames, especially early in the fuzzing exploration. Since AFL prioritizes keeping test files small, we expect variable sized instruction frames to produce better results, since this translates to longer hardware test sequences, and therefore deeper possible explorations of the (sequential) state space. Second, the opcode type seems to make little difference, for most experiments, since there are only 256 possible values, a search space AFL can explore very quickly. Lastly, we point out that for simple cores, like the RV-Timer, Hardware Fuzzing is able to achieve ≈85% HDL line coverage in less than a minute (hence we do not plot the full 24-hour trace).