2022

Directing greybox fuzzing to discover bugs in hardware and software

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

DIRECTING GREYBOX FUZZING TO DISCOVER BUGS IN HARDWARE AND SOFTWARE

by

SADULLAH CANAKCI

B.S., Sabanci University, 2016

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2022

Approved by

First Reader

Ajay J. Joshi, PhD
Professor of Electrical and Computer Engineering

Second Reader

Manuel Egele, PhD
Associate Professor of Electrical and Computer Engineering

Third Reader

Gianluca Stringhini, PhD
Assistant Professor of Electrical and Computer Engineering

Fourth Reader

Martin C. Herbordt, PhD
Professor of Electrical and Computer Engineering

And I knew exactly what to do. But in a much more real sense, I had no idea what to do. (Michael Garry Scott)

Acknowledgments

First, I would like to express my gratitude to my advisors, Prof. Ajay Joshi and Prof. Manuel Egele, for their continuous support and guidance throughout my PhD. Working with them has always been enlightening and inspiring. They have taught me how to become a successful researcher. Additionally, I like to extend my gratitude to the rest of my thesis committee members, Prof. Gianluca Stringhini and Prof. Martin Herbordt for their precious time and insightful feedback. I also appreciate the guidance from Prof. Michael Taylor in many projects. I would also like to thank my industry mentor Dr. John Kalamatianos at AMD for giving me the opportunity to apply my research ideas to real-world commercial products. I want to thank the funding agency, DARPA and BU Hariri Research, to support all my works.

I am very grateful to all my collaborators and co-authors specially Dr. Leila Delshadtehrani, Chathura Rajapaksha, Furkan Eris, and Dr. Boyou Zhou. I would like to extend a thank you to my lab mates from ICSG and SeclaBU. I truly appreciate the long-lasting friendships with all my friends at Boston University specially with Aselya, Furkan, Leila, Rasoul, Will, and Onur.

Finally, I want to thank my family for their continuous support and unconditional love in my whole life including my PhD journey. This work would not have been possible without them.

DIRECTING GREYBOX FUZZING TO DISCOVER BUGS IN HARDWARE AND SOFTWARE

SADULLAH CANAKCI

Boston University, College of Engineering, 2022

Major Professors: Ajay J. Joshi, PhD
Professor of Electrical and Computer Engineering

Manuel Egele, PhD
Associate Professor of Electrical and Computer Engineering

ABSTRACT

Computer systems are deeply integrated into our daily routines such as online shopping, checking emails, and posting photos on social media platforms. Unfortunately, with the wide range of functionalities and sensitive information stored in computer systems, they have become fruitful targets for attackers. Cybersecurity ventures estimate that the cost of cyber attacks will reach $10.5 trillion USD annually by 2025. Moreover, data breaches have resulted in the leakage of millions of people's social security numbers, social media account passwords, and healthcare information. With the increasing complexity and connectivity of computer systems, the intensity and volume of cyber attacks will continue to increase. Attackers will continuously look for bugs in the systems and ways to exploit them for gaining unauthorized access or leaking sensitive information.

Minimizing bugs in systems is essential to remediate security weaknesses. To this end, researchers proposed a myriad of methods to discover bugs. In the software domain, one prominent method is fuzzing, the process of repeatedly running a pro-

gram under test with "random" inputs to trigger bugs. Among different variants of fuzzing, greybox fuzzing (GF) has especially seen widespread adoption thanks to its practicality and bug-finding capability. In GF, the fuzzer collects feedback from the program (e.g., code coverage) during its execution and guides the input generation based on the feedback. Due to its success in finding bugs in the software domain, GF has gained traction in the hardware domain as well. Several works adapted GF to the hardware domain by addressing the differences between hardware and software. These works demonstrated that GF can be leveraged to discover bugs in hardware designs such as processors.

In this thesis, we propose three different fuzzing mechanisms, one for software and two for hardware, to expose bugs in the multiple layers of systems. Each mechanism focuses on different aspects of GF to assist the fuzzing procedure for triggering bugs in hardware and software. The first mechanism, TargetFuzz, focuses on producing an effective seed corpus when fuzzing software. The seed corpus consists of a set of inputs serving as starting points to the fuzzer. We demonstrate that carefully selecting seeds to steer GF towards potentially buggy code regions increases the bug-finding capability of GF. Compared to prior works, TargetFuzz discovered 10 additional bugs and achieved 4.03× speedup, on average, in the total elapsed time for finding bugs.

The second mechanism, DirectFuzz, adapts a specific variant of GF for software fuzzing, namely directed greybox fuzzing (DGF), to the hardware domain. The main use case of DGF in software is patch testing where the goal is to steer fuzzing towards recently modified code region. Similar to software, hardware design is an incremental and continuous process. Therefore, it is important to prioritize testing of a new component in a hardware design rather than previously well-tested components. DirectFuzz takes several differences between hardware and software (such as clock sensitivity, concurrent execution of multiple code fragments, hardware-specific coverage)

into account to successfully adapt DGF to the hardware domain. DirectFuzz relies on coverage feedback applicable to a wide range of hardware designs and requires limited design knowledge. While this increases its ease of adoption to many different hardware designs, its effectiveness (i.e., bug-finding success) becomes limited in certain hardware designs such as processors. Overall, compared to a state-of-the-work hardware fuzzer, DirectFuzz covers specified targets sites (e.g., modified hardware regions) 2.23× faster.

Our third mechanism named ProcessorFuzz relies on novel coverage feedback tailored for processors to increase the effectiveness of fuzzing in processors. Specifically, ProcessorFuzz monitors value changes in control and status registers which form the backbone of a processor. ProcessorFuzz addresses several drawbacks of existing works in processor fuzzing. Specifically, existing works can introduce significant instrumentation overhead, result in misleading guidance, and have lack of support for widely-used hardware languages. ProcessorFuzz revealed 8 new bugs in widely-used open source processors and identified bugs 1.23× faster than a prior work.

Contents

List of Tables

List of Figures

List of Abbreviations

AU	Architectural Unit
BCSG	Baseline Seed Corpus Generator
BTB	Branch Target Buffer
CDG	Coverage Directed Test Generation
CFG	Control-flow Graph
CFI	Control-flow Instruction
CG	Call Graph
CGF	Coverage-based Greybox Fuzzing
CSR	Control and Status Registers
DGF	Directed Greybox Fuzzing
DUT	Design Under Test
FSM	Finite State Machine
GF	Greybox Fuzzing
HDL	Hardware Description Language
IR	Intermediate Representation
ISA	Instruction Set Architecture
KASLR	Kernel Address Space Layout Randomization
MaxSat	Maximum Satisfiability
MSC	Minimum Set Cover
PoC	Proof of Concept
PTE	Page Table Entry
PUT	Program Under Test
PWM	Pulse Width Modulator
RAW	Read-After-Write
RTL	Register Transfer Level
SRA	Seed Refiner and Accumulator
TM	Transition Map
TSI	Target Sites Identifier
TSS	Target-oriented Seed Selector
TSX	Transactional Synchronization Extensions
TTE	Time-to-Exposure
TU	Transition Unit

Chapter 1

Introduction

1.1 Motivation

Computer systems have become an integral aspect of our lives today. We heavily rely on them in our daily routine such as checking emails, posting photos on social media platforms, and checking bank statements. With the wide range of functionalities and personal information stored in these systems, they have become a lucrative target for attackers. Malicious cyber activity has resulted in billions of dollars of damage to the economy [EOP, 2018], leakage of millions of people's social security numbers [CNET, 2018], a data breach affecting millions of patients [Staff, 2018], and more.

Attackers constantly look for ways to identify bugs in software and hardware so that they can penetrate systems by exploiting bugs. To illustrate the prevalence and significance of bugs, we will name a number of software and hardware bugs that have had a large-scale impact. For instance, in 2014, the Heartbleed [Bug, 2014] bug in the OpenSSL software library allowed adversaries to leak sensitive information (e.g., login credentials) from web servers. Researchers estimated that 24–55% of HTTPS servers in the Alexa Top 1 Million websites has been affected by Heartbleed [Durumeric et al., 2014]. Recently, a bug in the open-source Apache logging library named Log4j [Apache, 2001] allowed attackers to perform remote code execution, the ability to run any code on a victim machine. More than 35,000 Java packages (amounting to over 8% of the most significant Java package repository) have been affected by this particular bug [Google, 2022]. In the last 5 years, researchers showed that pro-

prietary processors, including AMD and Intel processors, are vulnerable to transient execution attacks such as Spectre [Kocher et al., 2019], Foreshadow [Van Bulck et al., 2018], ZombieLoad [Schwarz et al., 2019]. Researchers demonstrated the practicality of leaking security-sensitive information (such as credentials, cryptographic keys, or personal data) to an attacker by exploiting microarchitectural features of processors. For instance, Google's security team exploited Spectre against web users and successfully leaked data from the browser's memory [Google, 2021]. Besides transient execution attacks, researchers exposed many other hardware bugs in the past. For instance, a security researcher showed that systems that run 64-bit Xen hypervisor along with paravirtualization guests on Intel CPUs were exposed to privilege escalation bug [Wojtczuk, 2012]. Specifically, by exploiting this bug, guest administrators could gain full control of the host machine. With Intel's transactional synchronization extensions (TSX) bug [Jang et al., 2016], attackers could bypass the kernel address space layout randomization (KASLR) that allow them to easily predict the kernel memory layout. This way, it became easier for the attacker to exploit a memory corruption bug to gain unauthorized privilege in the target system. Intel's machine check bug [Intel, 2019] allowed adversaries to perform denial-of-service attacks by triggering a machine check that results in crashes and hangs in the processors.

In response to the increasing intensity and volume of cyber attacks, there has been tremendous effort in both academia and industry to develop effective methods for exposing bugs in systems. An effective method can eliminate many bugs in systems and prevent companies from paying the cost of security breaches both in terms of money and time. Besides, it can prevent the leakage of users' confidential information. Due to fundamental differences between software and hardware, the bug-finding methods used in these domains differ. In the following subsections, we provide commonly used methods in the software and hardware domains.

1.1.1 Finding Bugs in Software

Static analysis is a foundational part of discovering bugs in software where the goal is to expose bugs by examining the code of software without executing the program. The static analysis comes in many different forms. For instance, a plethora of works [Balakrishnan and Reps, 2004, Machiry et al., 2017, Clang, 2022, Yamaguchi et al., 2014, Livshits and Lam, 2005, Shankar et al., 2001] rely on analyses using different program representations (e.g., data-flow graph, control-flow graph, abstract syntax tree). These works perform various checks on these representations to discover patterns that signify security issues (e.g., buffer overflows, format string vulnerabilities). Another research line in static analysis is symbolic execution. Symbolic execution-based approaches [Godefroid et al., 2012, Cadar et al., 2008, Cha et al., 2012, Chipounov et al., 2012] perform more in-depth analysis and are able to identify hard to trigger bugs (e.g., complex sanity checks). Symbolic execution-based approaches generally do not scale to complex software because they suffer from a path explosion problem [Baldoni et al., 2018, Shoshitaishvili et al., 2016].

Methods that rely on dynamic analysis execute the program and monitor its behavior at runtime to detect bugs. For instance, dynamic taint analysis [Schwartz et al., 2010, Newsome and Song, 2005] tracks the propagation of labeled data (e.g., untrusted user input) during program execution, and detects when the data is used in a way that signifies a potential attack. Besides, sanitizers (e.g., `ASAN` [LLVM, 2022a], `LSAN` [LLVM, 2022b],`MSAN` [LLVM, 2022c]) can be used to expose memory safety bugs (e.g., buffer overflow). They statically instrument the program with necessary checks using a compiler and determine if a violation occurred at runtime. Tools like `Valgrind` [Valgrind, 2022] and `Pin` [Luk et al., 2005] provide dynamic instrumentation frameworks that allow developers to build a variety of dynamic analysis tools such as a heap profiler, thread error detector, memory error (e.g., access to an array

with an illegal index) detector, etc. Another prominent dynamic analysis method is fuzzing, the process of executing a program under test (PUT) recurrently with generated inputs (seeds) to trigger a software bug. Due to its simplicity, practicality, and capability of discovering security bugs, fuzzing has become highly popular in the last decade with more than two hundred academic works [Wcventure, 2022] and several industry-based platforms such as Google's OSS-Fuzz [Google, 2019] and Microsoft's Security Risk Detection [Microsoft, 2017]. Fuzzing has proved its effectiveness by discovering a plethora of vulnerabilities in widely used applications. For instance, Google's OSS-Fuzz platform exposed 30,000 bugs in 500 open-source projects as of June 2021 when running over more than 25,000 cores [Google, 2019].

The simplest variant of fuzzing is blackbox fuzzing which generates inputs independent of the internal program behavior. Blackbox fuzzing is highly compatible with complex software projects and generally very easy to use for testing applications. However, its bug-finding capability is limited due to the lack of information that is internal to the program. To improve the bug-finding capability of blackbox fuzzing, researchers proposed Greybox Fuzzing (GF) which collects internal program behavior (e.g., data flow [Gan et al., 2020,Google, 2021b], control flow [Google, 2017,Gan et al., 2018], memory accesses [Coppik et al., 2019]) at runtime to guide the input generation process. Widely-used GF tools such as AFL [Google, 2017] and libfuzz [LLVM, 2021] utilize code coverage (i.e., how many basic blocks/edges/lines are executed while a test is running.) of the PUT as the feedback. They use code coverage to determine if an input is "interesting", i.e., whether it results in coverage increase. If so, these GF tools apply a set of mutations to the "interesting" input to generate new inputs which are then fed to the PUT in the next fuzzing rounds. Here, the intuition is that generating new inputs from coverage increasing ones would cover even more unexplored code. Collecting internal program behavior makes GF more effec-

tive in exposing bugs compared to blackbox fuzzing [Böhme et al., 2017b]. To date, researchers proposed various variants of GF techniques. The most popular variants are Coverage-based Greybox Fuzzing (CGF) and Directed Greybox Fuzzing (DGF). CGF aims to maximize the code coverage of the program, thereby exposing bugs anywhere in the program. DGF steers test generation towards specific target sites such as recently modified code regions, and therefore, aims to discover bugs in a certain region of the program.

1.1.2 Finding Bugs in Hardware

Broadly, the methods for finding bugs in hardware can be divided into two categories - formal and dynamic. Formal methods (e.g., symbolic execution, model checking) use mathematical reasoning to prove that a hardware design conforms to its specification [Chen and Mishra, 2011, Mukherjee et al., 2015, Cadence, 2019]. Unfortunately, formal methods have a well-known state explosion problem, and therefore, do not scale well for complex hardware designs such as a processor [Dessouky et al., 2019].

Dynamic methods involve simulating a Design Under Test (DUT) with a test input and analyzing the behavior of the DUT during or after simulation to identify bugs. It is challenging and time-consuming to achieve high coverage with manually crafted inputs, especially for large hardware designs with millions of gates. To automate the test generation process, researchers proposed a variety of Coverage Directed Test Generation (CDG) mechanisms [Fine and Ziv, 2003, Wagner et al., 2005, Tasiran et al., 2001, Nativ et al., 2001, Gal et al., 2021, Bose et al., 2001, Squillero, 2005] . These mechanisms obtain coverage feedback from the DUT to automatically fine-tune the constraints of test generators, for example tuning the parameters of a Bayesian network used for test generation [Fine and Ziv, 2003]. However, these mechanisms are usually either DUT-specific or require in-depth design knowledge for the initial setup. Over the past few years, fuzzing has gained traction in the hardware domain due to

its bug-finding success in the software domain [Google, 2019] and also its ease of adoption to many different hardware designs. Several works [Hur et al., 2021, Laeufer et al., 2018, Trippel et al., 2021] demonstrated that GF can be adapted as a dynamic method in the hardware domain if certain differences (detailed in Section 2.4) between software and hardware are addressed.

1.2 Thesis Contribution

In this thesis, we analyze different aspects of fuzzing and propose solutions to improve the effectiveness (i.e., bug-finding capability) and efficiency (i.e., bug-finding speed) of fuzzing in both the software and the hardware domain. In total, we present three different fuzzing mechanisms; one for software and two for hardware. In our first work, we propose TargetFuzz, a mechanism that guides fuzzing towards recently modified code regions in a software program by outputting a target-specific corpus. Here, guiding the fuzzer towards modified regions is important because bugs mostly originate from recent code changes to the existing codebase [Zhu and Böhme, 2021]. In our second work, we present DirectFuzz, a directed hardware fuzzing mechanism that adapts DGF from the software domain to the hardware domain. Similar to software, hardware design is an incremental process, where new components are gradually added instead of designing the entire system in one step. Therefore, fuzzing the whole design after every single change in a hardware design unnecessarily wastes the effort of the fuzzer. DirectFuzz guides fuzzing towards verifying a specific part of the hardware design rather than the whole design, and therefore, spends most of its time budget on regions that need to undergo thorough verification. To adapt DGF from the software domain to the hardware domain, DirectFuzz addresses several challenges. For instance, power scheduling functions designed for the fuzzing software cannot be directly used for fuzzing hardware since the execution of hardware and software dif-

fers. Moreover, hardware designs consist of modules and have typically a concept of clock dependency while software mainly consists of functions and lacks a clock concept. Prior hardware fuzzing works [Laeufer et al., 2018, Li et al., 2021a, Muduli et al., 2020, Trippel et al., 2021] including DirectFuzz rely on coverage feedback types (e.g., branch coverage, multiplexer selection toggle coverage) that extend the general applicability of fuzzing to a variety of hardware designs. Coverage metric is a key factor in CGF when identifying 'interesting' test inputs, and therefore, significantly affects the success of fuzzing. While the general applicability is important, these coverage feedback types are not effective when fuzzing a specific type of hardware design; a processor. Our third work presents ProcessorFuzz tailored for fuzzing processors. ProcessorFuzz presents a novel coverage metric tailored for processor designs based on the registers that characterize the current processor state to guide fuzzing towards buggy hardware states. Overall, we advance four different aspects of fuzzing to systematically guide the fuzzing process towards exposing bugs in software and hardware layers of a computing system. We present the thesis statement as follows:

Effective guiding strategies that direct fuzzing can discover bugs in multiple layers of a computing system. The strategies can influence different aspects of fuzzing with a target-specific seed corpus and target-specific coverage guidance.

The main contributions of this Ph.D. thesis are as follows:

- **Guiding Software Fuzzing with a Target-specific Seed Corpus.** We observe that using seed corpora tailored for CGF limits the bug-finding capability of DGF. This is intuitive because CGF aims to maximize the coverage of the whole program and corpora designed for CGF includes seeds that cover different regions of a program. However, DGF aims to cover a certain program region (e.g., recently modified code region) and is misguided if the corpus in-

cludes irrelevant seeds to the target code region. To mitigate this shortcoming, we propose TargetFuzz, a mechanism that provides a DGF tool with a target-oriented seed corpus. We refer to this corpus as DART corpus, which contains only 'close' seeds to the targets. At a high level, the 'close' seed refers to a seed that mostly exercised basic blocks around the modified basic blocks. By selecting 'close' seeds, DART corpus guides DGF to the targets, thereby exposing bugs under limited fuzzing time. Evaluations on 34 real bugs show that AFLGo (a state-of-the-art directed greybox fuzzer), when equipped with DART corpus, finds 10 additional bugs and achieves 4.03× speedup, on average, in the time-to-exposure compared to a generic corpus tailored for CGF.

- **Guiding Test Generation for Hardware Designs using DGF.** Prior works [Laeufer et al., 2018, Trippel et al., 2021, Hur et al., 2021] on hardware fuzzing demonstrated the applicability of GF as a dynamic verification method on a variety of hardware designs to maximize the overall coverage. Unfortunately, fuzzing the entire hardware design to maximize overall coverage is not viable for a scenario where we just want to verify a newly added feature or recently modified component of the hardware design. To this end, we present DirectFuzz, a directed fuzzing approach that generates targeted test inputs to focus on the verification of specific hardware regions. In a nutshell, DirectFuzz varies from the prior works in two ways. First, DirectFuzz prioritizes test inputs that cover at least one of the target sites in the DUT when choosing the next stimulus. Second, DirectFuzz invests more time into test inputs that are closer to target sites when generating new test inputs. Our analyses show that DirectFuzz covers specified target sites (e.g., a recently modified hardware region) significantly faster than prior work (e.g., RFUZZ [Laeufer et al., 2018]) on a variety of RTL designs.

- **Guiding Processor Fuzzing via Control and Status Registers.** Existing processor fuzzers [Laeufer et al., 2018, Hur et al., 2021] have several drawbacks such as lack of support for widely-used Hardware Description Languages (HDLs), significant instrumentation overhead, and misleading coverage-signals that misidentify "interesting" inputs. Towards overcoming these shortcomings, we present ProcessorFuzz, a processor fuzzer that guides the fuzzer with a novel *CSR-transition coverage* metric. ProcessorFuzz monitors the transitions in Control and Status Registers (CSRs) as CSRs are in charge of controlling and holding the state of the processor. Therefore, transitions in CSRs indicate a new processor state, and guiding the fuzzer based on this feedback enables ProcessorFuzz to explore new processor states. ProcessorFuzz is agnostic of the HDL and does not require any instrumentation in the processor design. Thus, it can be easily adopted to a variety of different hardware languages without requiring language-specific instrumentation. We evaluated ProcessorFuzz with three real-world open-source processors – Rocket, BOOM, and BlackParrot. ProcessorFuzz triggered a set of ground-truth bugs 1.23× faster (on average) than DIFUZZRTL. Moreover, our experiments exposed 8 new bugs across the three RISC-V cores and one new bug in the reference model. All nine bugs were confirmed by the developers of the corresponding projects.

1.3 Organization

The remainder of this thesis is organized as follows. We provide the background on greybox fuzzing, seed selection strategies in software fuzzing, dynamic verification methods for hardware, CGF-based hardware fuzzing, and differential testing in the software domain in Chapter 2. Chapter 3 presents our work on the design and implementation of TargetFuzz to assist DGF tools with a target-specific seed corpus.

In Chapter 4, we introduce our automated test generation mechanism, DirectFuzz, for hardware designs using DGF approach. Chapter 5 presents our work on a processor fuzzer, ProcessorFuzz, to effectively expose bugs in processors using a novel processor-specific coverage metric. In Chapter 6, we discuss the future directions and conclude this thesis.

Chapter 2

Background and Related Work

In this chapter, we first provide a background on GF variants (i.e., CGF and DGF) and present prior works that are related to our first fuzzing mechanism (i.e., TargetFuzz). Then, we explain traditional verification techniques to discover bugs in hardware and how CGF is adapted as a dynamic verification method. Next, we present prior works that use fuzzing to find bugs in hardware. We conclude with prior works that use differential testing for testing software.

2.1 Greybox Fuzzing Variants

In this section, we first provide a background on CGF. Next, we explain the major differences between CGF and DGF.

2.1.1 Coverage-based Greybox Fuzzing

The main goal of CGF is to maximize the code coverage of a PUT. The intuition is that executing different code blocks in the target program with different inputs increases the chance of exposing bugs. At a high level, CGF generates a large number of test inputs by performing a set of mutation operations (such as bit flips) and executes the PUT with the generated inputs. Based on the coverage-feedback recorded from the PUT at runtime, CGF determines the 'interesting' inputs and uses them for generating a new set of inputs. Here an 'interesting' input refers to any input that increases code coverage.

We provide an algorithmic sketch of CGF in Algorithm 1. In stage **S1**, a fuzzer takes a seed corpus and a time limit as inputs. A seed corpus includes an initial set of test inputs that serve as starting points to the fuzzer. Seeds usually have infinite domains. For instance, a png processor tool can be provided with an infinite number of valid png files. To reduce the size of the initial seed corpus, it is a common practice to use a seed selection strategy that selects a subset of seeds from a large seed pool. Besides a seed corpus, stage **S1** takes as input a time limit that determines the total duration for fuzzing. Once fuzzing starts, all the remaining stages (i.e., **S2-S6**) repeat in a loop until the time limit is reached. Specifically, stage **S2** picks a seed from the seed corpus for the next fuzzing iteration. This stage of fuzzing is referred to as seed scheduling. The goal of seed scheduling is to pick an input that is more likely to increase coverage with mutations. Stage **S3** assigns energy to the scheduled seed. The energy of the seed determines the total number of mutations. Fuzzer performs more mutations on seeds with higher energy levels. Stage **S4** mutates the scheduled seed to produce a new set of seeds. This stage is referred to as seed generation. Stage **S5** executes the PUT with a mutated seed and makes an observation. Stage **S6** checks the observation for the mutated seed to determine if the seed increases coverage of the program (i.e., interesting) or results in an anomaly in the PUT behavior such as crashes, logic error, and unexpected exceptions. Mostly, Stage **S6** adds interesting seeds to the seed corpus and crashing seeds to a separate crashing seed set.

2.1.2 Directed Greybox Fuzzing

In a software project, any commit that changes a part of the program source produces a candidate buggy code region. The fuzzer should aim to maximize code coverage of the potential buggy code region rather than the whole program. DGF [Böhme et al., 2017a] is a suitable approach when this target program region is known. When a new commit is submitted by a developer, DGF can be used to generate test cases towards

Algorithm 1: Coverage-based Greybox Fuzzing

```
(S1) Input  : Initial Seed Corpus S, time_limit
Output: Crashing Input Set CI
CI ← ∅;
while time_elapsed < time_limit do
    (S2) s ← ScheduleNext(S);
    (S3) e ← AssignEnergy(s);
    for i = 1 to e do
        (S4) m' = MutateInput(s);
        (S5) o = ExecutePUT(m');
        (S6) if is_CRASHING(o) then
                 add m' to CI;
        (S6) else if is_INTERESTING(o) then
                 add m' to S;
    end
end
return CI
```

stressing target sites modified by the commit. During fuzzing, DGF steers the input generation towards modified code regions to detect any bugs in the target region.

DGF and CGF mainly differ at stage **S3** (i.e., how long to fuzz the chosen seed). The energy value of a seed controls the total number of mutations that need to be applied to the seed (i.e., higher energy results in more seed mutations). CGF [Google, 2017, LLVM, 2021, Yue et al., 2020, Böhme et al., 2017b, Google, 2021b] determines the energy of a seed with the goal of exploring more code regions. For instance, AFL [Google, 2017] assigns more energy to those seeds that exercise a new path. DGF [Böhme et al., 2017a, Chen et al., 2018] takes the program structure into account in its energy assignment formula so that it can spend longer fuzzing time on seeds that are likely to cover certain target sites. To achieve this goal, DGF instruments the source code by analyzing the call graph (CG) and control-flow graphs (CFGs) of the program. First, DGF assigns a value (function-level target distance) to each node in the CG on function-level by finding the shortest-path of each node to the target function (e.g., modified function). Next, DGF computes the basic-block-level

target distance of each basic block to all basic blocks that call a function in the CFG. Function-level target distance and basic-block-level target distance together define a seed distance formula. At run-time, the fuzzer uses the exercised basic blocks in the execution trace along with the seed distance formula to compute the seed distance. DGF determines the energy of a seed based on the seed distance and performs more mutations on a seed that is 'closer' (i.e., has lower seed distance) than on a seed that is 'further away' from target sites. The intuition is that the seeds mutated from a close seed are more likely to reach the target sites. Note that some of the DGF works [Österlund et al., 2020, Chen et al., 2018] utilize seed distances at stage **S2** to prioritize closer seeds in seed scheduling.

2.2 Seed Selection Strategies

In this section, we present prior works that are related to TargetFuzz. Specifically, TargetFuzz provides a target-specific seed corpus to a DGF tool by selecting a subset of seeds from a large seed pool. Therefore, it is highly related to prior seed selection strategies that reduce the size of a large seed pool. In Table 2.1, we provide a brief summary of different seed selection strategies. While the main goal of prior seed selection strategies is to select a subset of seeds that cover all the unique edges in a seed pool, TargetFuzz aims to select only "closer" seeds to the target region. This way, TargetFuzz aims to guide a DGF tool to the target regions.

`afl-cmin` [Zalewski, 2017] is one of the widely-used tools, and it relies on AFL's own notion of edge coverage when selecting a subset of seeds. `afl-cmin` first computes the edge coverage (a.k.a. branch coverage) of each seed in the large seed pool. Next, it finds a set of edges by calculating the union of edge coverage across all the seeds. To reduce the corpus size, `afl-cmin` performs a greedy search that selects the smallest size of seed for each edge in the set.

Abdelnur et al. [Abdelnur et al., 2010] propose a seed selection strategy (`MINSET`) that applies the minimum set cover (MSC) problem to the edge covering. Since MSC is an NP-hard problem, the authors use an approximation greedy algorithm. In Algorithm 2, we provide the greedy algorithm for `MINSET`. Given a union set of covered edges U and a list of sets S where each set contains covered edges from a particular seed, `MINSET` computes a minimum set C that covers all the edges in U. To compute C, at each iteration, `MINSET` greedily searches for the set S_i that contains the greatest number of uncovered elements by C. Later, Rebert et al. [Rebert et al., 2014] extend MSC to weighted set cover problem by considering seed execution times (`TIMEMINSET`) and seed sizes (`SIZEMINSET`). Among these three approaches, `MINSET` has the highest bug-finding detection capability as shown by Rebert et al. [Rebert et al., 2014].

A recent work by Herrera et. al [Herrera et al., 2021] proposes a new seed selection strategy, namely `OPTIMIN`, that implements an optimal corpus minimization tool for AFL. Seed selection strategies like `TIMEMINSET` and `SIZEMINSET` employ some heuristics to compute a set cover `C` since the underlying problem (i.e., weighted set cover) is NP-complete. `OPTIMIN` computes the exact solution to this NP-complete problem in a reasonable time by interpreting the problem as a maximum satisfiability (MaxSAT) problem. The MaxSAT relies on hard and soft constraints where the goal is to satisfy all hard constraints and maximize the number of satisfied soft constraints. By treating edge coverage as a hard constraint while excluding a seed in the solution as a soft constraint, `OPTIMIN` guarantees to cover all edges with a minimal number of seeds.

At its core, all the five aforementioned seed selection strategies aim to maximize the code coverage. We demonstrate (in Section 3.4) that these coverage-based seed selection strategies are not well-suited for DGF (i.e., DGF wastes the fuzzing effort on

Table 2.1: Comparison of different seed selection strategies.

Strategy	Goal	Approach	Target
afl-cmin [Zalewski, 2017]	Cover all edges in the seed pool	Perform greedy search to select the smallest size of seeds	CGF
MINSET [Abdelnur et al., 2010]	Cover all edges in the seed pool	Perform greedy search to select seeds covering greatest number of uncovered edges	CGF
TIMEMINSET [Rebert et al., 2014]	Cover all edges in the seed pool	Extend MINSET's approach using seed execution times	CGF
SIZEMINSET [Rebert et al., 2014]	Cover all edges in the seed pool	Extend MINSET's approach ‘4e using seed sizes	CGF
OPTIMIN [Herrera et al., 2021]	Cover all edges in the seed pool	Solve seed selection as a MaxSat problem	CGF
TargetFuzz	Select 'close' seeds from the seed pool	Select the seeds with lowest seed distances	DGF

unrelated code) and propose a seed selection strategy that accounts for seed distance rather than merely coverage.

Algorithm 2: Greedy Algorithm for MINSET.

Input : A Set of Covered Branches U, A List of Sets S
Output: Set Cover C
$C \leftarrow \emptyset$;
while *U contains elements not covered by C* **do**
    Find the set S_i containing the greatest number of uncovered elements
    Add S_i to C
end
return C

2.3 Traditional Hardware Verification

Both DirectFuzz and ProcessorFuzz aim to identify bugs in hardware designs. In this section, we provide hardware verification methods that aim at the same goal (i.e., finding bugs in hardware). Random instruction generators [Lee and Cook, 2015, Google, 2021c, Futurewei Technologies, 2020, Group, 2018, Herdt et al., 2020] have been commonly used in processor verification since they require limited human expertise and are scalable to large RTL designs. These tools produce random assembly programs based on a set of constraints such as the instruction mix, frequencies, etc., to identify functional bugs in processors. The lack of coverage guidance in these tools leads to the generation of the repetitive inputs that test the same processor functionalities, thereby decreasing the chances of finding bugs [Hur et al., 2021, Laeufer et al., 2018].

A verification engineer can target the uncovered RTL regions by adjusting the constraints that control the random test generator. For instance, if coverage is maximized in the branch prediction unit but not in the load-store unit, the verification engineer can increase the ratio of load and store instructions. However, this method significantly increases engineering effort, and therefore, slows down the verification process. To overcome this problem, researchers proposed several coverage-directed test generation mechanisms [Fine and Ziv, 2003, Wagner et al., 2005, Tasiran et al., 2001, Nativ et al., 2001, Gal et al., 2021, Bose et al., 2001, Squillero, 2005] that automatically direct the next round of test generation that targets the uncovered parts of RTL. These works use RTL simulators to dynamically monitor the behavior of an RTL design and adjust the test generator constraints towards producing tests inputs that target uncovered RTL regions.

CDG is a widely-used technique for the verification of RTL designs. In this technique, the constraints of a test generator are automatically driven by the coverage feedback so that the test input generated in the next round can increase the overall coverage. For instance, MicroGP [Squillero, 2005] aims to verify the whole microprocessor design by generating test inputs using an instruction template based on genetic programming. The fitness value that drives the searching of an instruction sequence is determined by the statement coverage. Fine et al. [Fine and Ziv, 2003] propose a CDG mechanism based on Bayesian networks. Unfortunately, setting up the network is not a straightforward task and requires in-depth expertise in the design specifications of the RTL design. Wagner et al. [Wagner et al., 2005] present a CDG framework that uses a Markov chain model. The weights of the model are fine-tuned based on the collected coverage. This framework relies on the abstract form of the DUT that needs to be crafted manually in the form of a custom template. Therefore, it requires deep domain knowledge. Overall, CDG mechanisms aim to find a balance

between the amount of domain knowledge applied to the framework and the general applicability of the mechanism [Ioannides and Eder, 2012].

Formal verification methods (e.g., symbolic execution, model checking) are also widely used in hardware verification [Chen and Mishra, 2011, Mukherjee et al., 2015, Cadence, 2019]. These methods use mathematical reasoning to prove that a hardware design conforms to its specification. Unfortunately, formal verification methods have a well-known state explosion problem, and therefore, do not scale well for complex RTL designs such as a processor [Dessouky et al., 2019].

2.4 Adapting CGF for Processor Fuzzing

Recent works [Hur et al., 2021, Laeufer et al., 2018, Trippel et al., 2021] show that CGF can be adapted as a dynamic verification method for hardware including processors. In this section, we briefly explain two important aspects when adapting CGF to processor fuzzing.

Hardware Execution. In the case of CGF for software, the fuzzing target is a software program that can be directly executed on a host machine with a test input after compilation. However, hardware (e.g., a processor) is not directly executable on the host machine. A hardware design is implemented with an RTL abstraction and must be simulated with an RTL simulator to evaluate a test input. RTL describes the hardware design in terms of data transfer between registers and the logical operations between the registers. The RTL design is usually expressed with an HDL (e.g., Verilog, VHDL, etc.). The RTL simulator can provide cycle-accurate information regarding the real-time behavior of the RTL design.

Bug Detection. Most software fuzzers focus on bugs that manifest as memory safety violations such as segmentation faults. These types of bugs are relatively easy to detect because they cause an observable anomaly (i.e., crash) in program behavior.

However, fuzzing to find semantic bugs (e.g., logic errors) is harder than discovering memory violations because defining semantic violations is a highly domain-specific task. For these types of bugs, researchers proposed differential testing [Sahin et al., 2018, Martignoni et al., 2009, Brubaker et al., 2014, Min et al., 2015]. Differential testing compares the output of multiple programs that have the same functionality and checks for inconsistent behaviors. This approach is used by processor fuzzers [Lee and Cook, 2015, Google, 2021c, Hur et al., 2021] as well. In particular, the processor fuzzer provides the same input to both the RTL simulator and the reference model. Here, the reference model is an ISA simulator that mimics the behavior of all the ISA-level operations. The ISA simulator is a software model of the hardware and does not require any low-level microarchitectural details (e.g., the pipeline depth, buffer size). For a given program, it computes the values of architectural registers and memory state after the execution of each instruction. In contrast, the RTL simulator is cycle-accurate and realizes the effect of executed instructions in the microarchitectural level such as the available packets in a buffer or branch prediction result of a conditional branch. The hardware fuzzer extracts an execution trace log from both the ISA simulator and the RTL simulator for the same input and cross-checks the traces. Here, the execution trace contains the final memory states and architectural registers. Any mismatch in the traces is considered a potential bug in the processor and is marked for further investigation by the verification engineer.

2.5 Hardware Fuzzing Works

Over the past few years, fuzzing has gained traction in RTL verification due to its bug-finding success in the software domain [Google, 2019]. In Table 2.2, we provide a high-level overview of all fuzzing-based RTL verification approaches. For each approach, we include the input format, the coverage metric used to guide the fuzzer,

Table 2.2: Existing RTL Fuzzers.

	Input Format	Coverage Metric	Evaluated RTL Designs	Bug Discovery Method
RFUZZ [Laeufer et al., 2018]	A Series of Bits	Mux Toggle	Peripherals, RISC-V Processors (Sodor 1-3-5)	Assertion
Li et. al [Li et al., 2021a]	A Series of Bits	Full Mux Toggle	Custom RISC-V Processor, OpenCore 1200	Assertion
DIFUZZRTL [Hur et al., 2021]	Assembly	Register Coverage	RISC-V Processors (BOOM, Mork1x, Rocket Chip,)	Golden Model
Trippel et al. [Trippel et al., 2021]	Byte Sequence	Edge Coverage	RISC-V IP Cores (AES, HMAC, KMAC, Timer)	Golden Model Assertion
HYPERFUZZER [Muduli et al., 2020]	A Series of Bits	High-Level	Custom SoC	Property Check
Logic Fuzzer [Kabylkas et al., 2021]	A Series of Bits, Random Data	N/A	RISC-V Processors (BlackParrot, BOOM, CVA6)	Golden Model
TheHuzz [Tyagi et al., 2022]	Assembly	branch, conditional, FSM, expression, toggle, statement	RISC-V Processors (Rocket Chip, Ariane), mor1kx, or1200	Golden Model
ProcessorFuzz	Assembly	Control Path Register, ISA-Sim Transition	RISC-V processors (BOOM, BlackParrot, Rocket Chip)	Golden Model

and the method to identify bugs.

RFUZZ [Laeufer et al., 2018] defines a simple input format (i.e., a series of bits) to increase the portability of hardware fuzzing to a wide range of RTL designs. Unfortunately, this input format is not effective when fuzzing processors since a processor requires instructions defined by an ISA. RFUZZ also proposes a new coverage metric, the multiplexer toggle coverage. RFUZZ monitors all the multiplexers in the RTL design. It retains an input for further mutations if the input toggles a previously uncovered multiplexer selection signal. A follow-up work by Li et al. [Li et al., 2021a] enhances RFUZZ with symbolic simulation and defines a full multiplexer toggle coverage metric that counts a multiplexer signal as covered for either 1-0-1 or 0-1-0 toggles. Both RFUZZ and Li et al. are highly coupled to Chisel HDL which limits the applicability of the approach [Sadeghi et al., 2021]. Additionally, monitoring multiplexers in complex designs introduces excessive performance overhead [Hur et al., 2021]. ProcessorFuzz is agnostic to HDL and also does not require any instrumentation in the HDL code, which makes it both practical and efficient.

DIFUZZRTL monitors registers that directly or indirectly control multiplexer selection signals. This design choice makes it more efficient than RFUZZ since the total number of bits in the identified registers is significantly less than multiplexers. Moreover, DIFUZZRTL shows that RFUZZ's coverage metric does not precisely capture the FSM states. To mitigate this issue, DIFUZZRTL monitors value changes in the

identified registers for each cycle. Unfortunately, DIFUZZRTL monitors many registers in the datapath as well, thereby misguiding the fuzzer as detailed in Section 5.2.

Trippel et al. [Trippel et al., 2021] translate hardware designs to software models and fuzzes those models. This way, available coverage metrics used by software fuzzers (e.g., basic block, edge) can be used for fuzzing hardware as well. However, this method of converting hardware designs to software models introduces additional challenges such as proving the equivalency between hardware design and software model [Sadeghi et al., 2021].

TheHuzz [Tyagi et al., 2022] aims to identify instruction and mutation sequences that are more likely to increase coverage. Specifically, it formulates an optimization problem that computes the weights for different instruction and mutation sequences. Using the weights, it then guides seed scheduling and seed mutation phases of the fuzzing. TheHuzz relies on already existing coverage metrics such as branch, condition, FSM.

The common goal of the aforementioned fuzzing works is to maximize coverage of an RTL design, thereby discovering bugs across the entire RTL design. Researchers have also proposed fuzzing frameworks for achieving alternate goals. HYPERFUZZER [Muduli et al., 2020] introduces a new grammar that represents the hardware security properties. During fuzzing, HYPERFUZZER checks if any of the fuzzer-generated inputs violates a security property. Defining properties requires human expertise which is error-prone. Logic Fuzzer [Kabylkas et al., 2021] randomizes control signals and states of a DUT without compromising the functional correctness of the DUT. Logic Fuzzer needs to be provided with fuzzing targets (e.g., congestible points in an RTL design), and therefore requires domain expertise. INTROSPECTRE [Ghaniyoun et al., 2021] and Osiris [Weber et al., 2021] use blackbox fuzzing approach to discover microarchitectural side channels (i.e., Meltdown [Lipp et al.,

2018] and Spectre [Kocher et al., 2019]) in processors.

2.6 Differential Testing in Software Domain

Differential testing is commonly used in the software domain to discover inconsistencies (e.g., semantic bugs, side-channels, consensus bugs) across multiple programs with similar functionalities. One use case of differential testing in the software domain is to identify discrepancies between emulators and real hardware. Prior works [Sahin et al., 2018, Martignoni et al., 2009, Paleari et al., 2009] aim to eliminate the source of discrepancies in emulation environments since adversaries use discrepancies to infer the execution environment and bypass malware analysis. ProcessorFuzz differs from these works in two ways. First, ProcessorFuzz's test input generation is coverage-guided whereas these works employ blackbox fuzzing. Second, these works aim to identify discrepancies that may or may not necessarily translate to actual bugs. Besides emulators, differential testing has been used to test different types of software including Web application firewalls [Argyros et al., 2016], SSL/TLS libraries [Petsios et al., 2017, Brubaker et al., 2014, Chen and Su, 2015, Sivakorn et al., 2017], compilers [Yang et al., 2011], cryptocurrency protocols [Yang et al., 2021, Fu et al., 2019, EVM, 2021], deep learning systems [Pei et al., 2017, Noller et al., 2020], Java Virtual Machines [Chen et al., 2016, Chen et al., 2019], PDF viewers [Petsios et al., 2017], mobile applications [Jung et al., 2008], file systems [Min et al., 2015], and Java programs [Nilizadeh et al., 2019, Noller et al., 2020].

Chapter 3

Guiding Software Fuzzing with a Target-specific Seed Corpus

3.1 Introduction

The nature of modern software development is continuous and incremental. Rather than in one large batch, developers continuously build, test, and deploy their changes in small batches. Developers use effective testing techniques to detect bugs as software evolves. Due to its ease in deployment and effectiveness in bug finding, it has become a common practice for developers to use GF as part of their testing framework. After each pull request or commit, it is common to run some fuzzing sessions with the goal of finding bugs before integrating submitted changes into the codebase.

To date, researchers have proposed two types of GF techniques: CGF and DGF. The goal of CGF is to maximize the code coverage of the program, thereby discovering bugs anywhere in the program. When the program undergoes substantial changes, CGF is effective, especially with long fuzzing runs. However, it is rare to observe substantial software changes in a real-world software project. The maintenance of real-world projects often involves frequent commits where each commit only modifies a few lines of code. As a reference point, the average commit size across five popular projects (i.e., PHP, Libxml2, Openssl, SQLite3, and Libpng) is 34.8 lines of code [1]. Moreover, fuzzing requires resources and hence is subject to budget constraints,

[1]We use the master branch of each project and include commits that only change source code (including both additions and deletions) in the last 14 years.

especially time. The commit frequency of popular projects is generally high (on average one commit for every 20.4 hour for the aforementioned five projects) and in turn, results in limited fuzzing time. A well-suited fuzzing technique for testing small-size commits, especially under limited fuzzing time, is DGF which steers the test generation towards specific target sites (e.g., code lines modified by a recent commit) rather than unrelated program components. DGF computes the distance of each seed with respect to target sites by averaging the weight of executed basic blocks. The weight of each basic block is determined based on the shortest paths to target basic blocks in the program inter-procedural control-flow graph. During fuzzing, DGF focuses on closer seeds (i.e., seeds with lower distances) to cover modified code regions, thereby increasing the likelihood of detecting bugs even under limited fuzzing time.

While prior works improve several aspects of DGF (e.g., energy assignment [Chen et al., 2018], seed scheduling [Zong et al., 2020], and target selection [Österlund et al., 2020, Wang et al., 2020a]), little attention has been given to improving the seed selection process that determines seeds in an initial seed corpus. The initial seed corpus consists of a set of valid inputs (e.g., png files for a png processor tool) serving as starting points to the fuzzer. The seed selection process aims to identify high-quality seeds from a large seed pool that maximize the effectiveness of a fuzzer (mainly measured with its bug-finding capability). If an initial seed corpus is not provided, the fuzzer wastes the limited fuzzing time to infer the file format that the PUT accepts. In case a seed corpus is provided, the quality and the quantity of seeds significantly impact the effectiveness of the fuzzer in bug detection [Rebert et al., 2014, Klees et al., 2018]. Although there exists a variety of seed selection strategies/tools [Zalewski, 2017,Rebert et al., 2014,LLVM, 2021,Herrera et al., 2021], they are mainly designed for CGF. In this work, we tackle the question of whether a seed corpus tailored for DGF outperforms a CGF-based seed corpus when used with

Table 3.1: Seed corpora used by DGF tools. RT refers to regression tests, GIT refers to seeds obtained from a git repo, N/A states that the origin of seed corpus is not clearly specified.

Tool Name	Seed Corpus
AFLGo [Böhme et al., 2017a]	RT, OSS-Fuzz Seeds, Generic Seeds (AFL)
Hawkeye [Chen et al., 2018]	Same as AFLGo
LOLLY [Liang et al., 2019]	RT, Generic Seeds (AFL)
Memfuzz [Coppik et al., 2019]	Null Seed, RT, Generic Dictionary (AFL)
TortoiseFuzz [Wang et al., 2020b]	GIT, RT, Generic Seeds (N/A)
UAFuzz [Nguyen et al., 2020]	Null Seed, Generic Seeds (N/A)
UAFL [Wang et al., 2020a]	RT, Random Seeds from the internet
Memlock [Wen et al., 2020]	N/A
IJON [Aschermann et al., 2020]	Seed Containing 'a', Generic Seeds (random) Dictionary of Strings from Source
FuzzGuard [Zong et al., 2020]	Same as AFLGo
ParmeSan [Österlund et al., 2020]	Single file with '\n', Google Fuzzer Test-suite
1DVUL [Peng et al., 2019]	N/A
SAVIOR [Chen et al., 2020]	Generic Seeds (AFL), RT
CAFL [Lee et al., 2021]	N/A

a DGF tool.

To date, all the academic DGF works [Böhme et al., 2017a, Chen et al., 2018, Liang et al., 2019, Coppik et al., 2019, Wang et al., 2020b, Nguyen et al., 2020, Wang et al., 2020a, Aschermann et al., 2020, Zong et al., 2020, Österlund et al., 2020, Chen et al., 2020] equip their directed greybox fuzzers with a seed corpus designed for CGF (CGF-based seed corpus) for the evaluation (see Table 3.1). Specifically, these fuzzers commonly use regression tests [SQLite, 2021, Google, 2021a], generic seeds [Google, 2020b,glennrp, 2018], and generic dictionaries [Google, 2020a] as part of a seed corpus. Additionally, they extensively use a variety of seed selection strategies [Rebert et al., 2014,Zalewski, 2017,LLVM, 2021,Herrera et al., 2021] to ensure that only high-quality seeds construct their seed corpora. At their core, these strategies select a minimal set of inputs (as part of a seed corpus) that provide maximal code coverage and therefore are designed for CGF.

We make a key observation that using CGF-based seed corpora significantly limits the bug detection capability of DGF. Moreover, we observe that a seed selection mechanism (namely `MINSET` [Rebert et al., 2014]) that is well-suited for CGF performs poorly with DGF. These observations are intuitive. A seed corpus tailored for CGF exercises different regions of the whole program, thereby triggering bugs in different parts of the program. However, DGF aims to reach a specific region (e.g., newly written or modified code) of the program and trigger bugs in that region. When DGF uses a seed corpus that covers different regions of the program, DGF wastes part of its effort by spending time on fuzzing "farther away" seeds that are unrelated to the target regions. Instead, DGF should be provided with a seed corpus that contains "closer" seeds to the target sites so that DGF spends the limited testing time on fuzzing seeds related to targets.

In this work, we present TargetFuzz, a mechanism that provides a target-specific seed corpus to a DGF tool. At its core, TargetFuzz exploits the continuous and incremental nature of software development. For the frequent mode of software development (i.e., the mode where small-size changes happen), TargetFuzz outputs a commit-specific seed corpus which we refer to as DART corpus[2]. TargetFuzz uses a seed selection strategy that accounts for the seed distances rather than only code coverage to select seeds in DART corpus. The strategy selects a subset of seeds that are 'close' to the modified target regions from a large seed pool. A DGF tool equipped with DART corpus spends most of the fuzzing time budget on stressing modified code regions rather than unrelated program components. In this way, the DGF tool achieves better code coverage and successfully exposes bugs in target sites even under limited testing time. TargetFuzz uses the rare mode of software development (i.e., the mode where substantial software changes happen) to generate the large seed pool.

[2]We use DART to signify that our corpus is targeted to a (code) region and needs to achieve its goal (covering that particular code region) quickly similar to a dart that can be thrown at a target with a quick movement.

Using coverage-based fuzzers, TargetFuzz generates a variety of seeds that exercise different regions of the program. Whenever a developer submits a small-size commit, TargetFuzz selects those seeds that are related to the modified code region from the large seed pool as opposed to a CGF-based corpus that covers unrelated code regions as well.

We evaluate TargetFuzz using the state-of-the-art directed greybox fuzzer (AFLGo [Böhme et al., 2017a]) using a fuzzing benchmark (Magma [Hazimeh et al., 2020b]). In total, we test 34 vulnerabilities on 7 popular programs. Experimental results show that AFLGo, when equipped with DART corpus, triggers 10 more unique bugs (out of 34) and achieves 4.03× speedup, on average, in the time-to-exposure (TTE) of bugs compared to a generic corpus. We also compare TargetFuzz's distance-based seed selection strategy with a seed selection strategy tailored for CGF (`MINSET` [Rebert et al., 2014]). When equipped with DART, AFLGo discovers 8 more unique bugs (out of 34) compared to `MINSET`-based corpus and triggers the bugs 30% faster (on average).

Overall, the main contributions of this work are as follows:

- We are first to observe that CGF-based seed corpora consisting of generic seeds limit the effectiveness of a DGF tool. Based on this observation, we design TargetFuzz, a mechanism that provides a DGF tool with different target-specific seed corpora for different fuzzing targets.

- We show that the state-of-the-art CGF-based seed selection strategies are not well-suited for DGF. To increase the bug detection capability of a DGF tool, TargetFuzz proposes a seed selection strategy that takes seed distance metric into account rather than merely code coverage.

- We evaluate TargetFuzz on a variety of real bugs in real-world programs (in total 34 bugs across 7 libraries). Experimental results show that a DGF tool can find

more unique bugs when equipped with a seed corpus provided by TargetFuzz compared to a corpus tailored for CGF.

3.2 TargetFuzz: Design

We illustrate the design overview of TargetFuzz in Figure 3·1. TargetFuzz mainly consists of three components, a Baseline Seed Corpus Generator (BSCG), a Seed Refiner and Accumulator (SRA), and a Target-oriented Seed Selector (TSS). The BSCG is in charge of generating a variety of seeds that cover different code regions of a program. To accomplish this goal, the BSCG performs fuzzing using multiple coverage-based greybox fuzzers by allocating a relatively long time limit and collects all of the 'interesting seeds' (i.e., any seed that increases coverage) from each fuzzer's output queue in one place. The fuzzer-generated interesting seeds comprise a large seed pool that is used in a later stage of TargetFuzz when selecting seeds for a commit-specific corpus. We refer to this large seed pool as *baseline seed corpus* and the elements of this corpus as *baseline seeds*. TargetFuzz runs the BSCG for two different scenarios. The first scenario is when TargetFuzz is initially integrated into the testing framework of a software project. The second scenario is when the software project undergoes substantial changes. This is an occasional scenario in a software project as the maintenance of real-world projects mostly involves small size commits that are submitted with high frequency.

Once the baseline seed corpus is generated, TargetFuzz waits until a developer submits a new commit that modifies the program source. After the submission of the commit, TargetFuzz runs the SRA component to identify crashing inputs and to eliminate duplicate seeds. Specifically, the SRA first checks if any of the available seeds in the baseline seed corpus triggers a bug introduced by a new commit. Next, the SRA refines the baseline seed corpus since different coverage-based fuzzers used

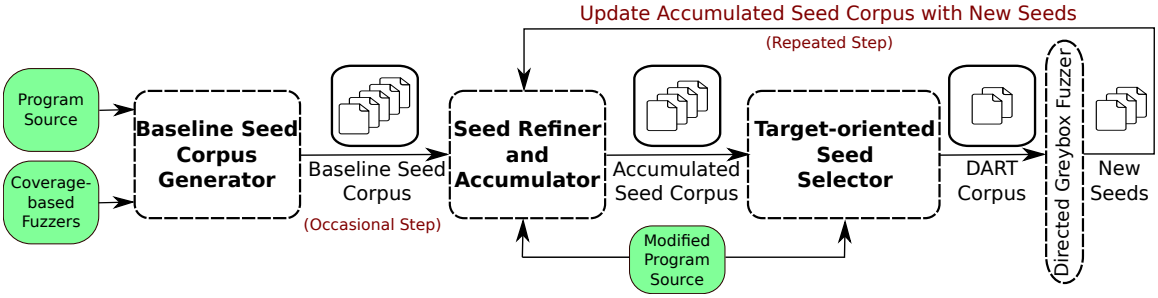


Figure 3·1: TargetFuzz Design: The inputs of TargetFuzz are color-coded with green. The Baseline Seed Corpus Generator produces a wide range of seeds using CGF tools. Seed Refiner and Accumulator eliminates the baseline seeds with similar characteristics and accumulates incoming seeds from the fuzzing sessions of submitted commits. Target-oriented Seed Selector selects commit-specific seeds from the accumulated seed corpus and outputs a DART corpus.

in the BSCG can generate the same and/or similar inputs. In particular, the SRA retains only the smallest set of seeds that result in the same code coverage. The seeds after the refining process are used to form the *accumulated seed corpus*.

The TSS selects a subset of seeds from the accumulated seed corpus and outputs a target-specific seed corpus which we refer to as *DART corpus*. While selecting seeds of the DART corpus, the TSS takes the seed distance into account to select only those seeds that are 'close' to the modified code region. The TSS takes the modified program source to compute seed distances. TargetFuzz equips the directed greybox fuzzer with the DART corpus, sets the modified lines of code as target sites, and starts the fuzzing session. Once the fuzzing session is over, TargetFuzz first checks if a new bug is detected. Next, the fuzzer-generated 'interesting' new seeds (i.e., seeds that increased coverage during fuzzing) are provided to the SRA, which augments the accumulated seed corpus. When the next commit is submitted, the SRA first refines the augmented accumulated seed corpus to eliminate redundant seeds obtained from the previous fuzzing session. Note that the SRA uses the accumulated seed corpus for the next commits (instead of the baseline seed corpus) until the BSCG needs to

generate a new baseline seed corpus because of a major program change.

3.2.1 Baseline Seed Corpus Generator (BSCG)

The BSCG creates a large seed pool consisting of a variety of seeds that cover different code regions of the program. To provide a commit-specific DART corpus to a DGF tool, TargetFuzz applies its seed selection strategy on the accumulated seed corpus mostly consisting of baseline seeds along with a limited number of new seeds generated by DGF. This is because TargetFuzz allocates very limited time for fuzzing small-size commits, and so it is unlikely to achieve a promising code coverage with new seeds (i.e., the output of Directed Greybox Fuzzer in Figure 3·1) coming from DGF sessions. If the baseline seeds cover only a certain region of the program or a limited number of easy-to-reach program regions, the DART corpus will include 'further away' seeds for most of the commits and fail to guide DGF towards the target code regions. To ensure that the DART corpus includes a set of 'close' seeds that are related to program regions modified by the recent commit, it is essential to collect a wide range of baseline seeds that exercise different regions of the program. To this end, TargetFuzz runs multiple coverage-based fuzzers with different characteristics since none of the prior fuzzers manifest clear superiority over the others [Li et al., 2021b, Hazimeh et al., 2020b]. Additionally, to improve the code coverage, TargetFuzz allocates a relatively longer fuzzing time limit for this step compared to DGF sessions. However, it is important to note that this step is occasional as it is performed only after a major software change and the time cost of this step is negligible in the long term given the high frequency of small-size commits.

3.2.2 Seed Refiner and Accumulator (SRA)

The SRA initially checks if the modified program contains a bug that can be detected with any of the seeds in the accumulated seed corpus. If a bug is detected with a seed

that resides in the accumulated seed corpus, the developers can fix the bug with a follow-up commit before running any fuzzing sessions. In case the developers ascribe low priority to a detected bug, it is essential to ensure that the bug is not a 'fuzz blocker' that limits the effectiveness of fuzzing [Firefox, 2021].

As a next step, the SRA eliminates seeds that result in the same branch coverage since using the same type of seeds leads to wasted fuzzing effort [Rebert et al., 2014]. The SRA uses two practices that are widely used by existing corpus minimization tools [Zalewski, 2017, LLVM, 2021]. First, the SRA eliminates seeds that achieve the same branch coverage. The SRA chooses only one seed from each set of seeds that produces the same branch coverage. Second, we choose the seed with the minimum file size among all the seeds that result in the same branch coverage. As discussed by prior works [Xu et al., 2017, Zalewski, 2017, Manès et al., 2019], here the rationale is to reduce the search space for mutation while minimizing I/O requests.

3.2.3 Target-oriented Seed Selector (TSS)

As detailed before, the time limits of DGF fuzzing sessions are significantly limited (usually less than 24 hours). TargetFuzz constructs a seed corpus consisting of "closer" inputs that guide DGF towards the modified code regions, thereby reaching target sites under a limited fuzzing time. To steer the execution towards target sites (e.g., modified lines of code after a commit), the TSS component uses a seed selection strategy that takes the seed distance into account rather than merely code coverage when selecting the seeds for the target-oriented DART corpus. The distance-based DART corpus helps DGF to focus on 'close' seeds that are more likely to lead to the target regions during fuzzing. Additionally, by eliminating 'further away' seeds, the TSS prevents a DGF tool from wasting the limited testing time on fuzzing seeds that are unrelated to the modified region.

The TSS component instruments the program source to generate a DGF-specific

binary. The DGF-specific binary is generated by applying necessary compiler passes provided by the directed greybox fuzzer and it includes bookkeeping logic to compute the distance of a provided seed to the given set of target sites. The TSS then executes the instrumented binary with each of the seeds in the accumulated seed corpus to obtain their corresponding seed distance values. Next, based on the computed seed distances, the TSS selects a certain number of 'close' seeds (with smallest seed distances) from the accumulated seed corpus and assembles these seeds in the DART corpus. Here, one important design aspect of TargetFuzz is the process of determining DART corpus size. At one extreme, the TSS can select the single closest seed to include the most related seed to the target region in DART corpus. At another extreme, the TSS can select all the seeds to increase the variety in DART corpus. Overall, there is a trade-off between the closeness (∼quality) and variety (∼quantity) of seeds. In Section 3.3, we explain the details of our implementation choice that tackles this problem.

A DGF tool uses the DART corpus for fuzzing the modified program. After the fuzzing time limit is over, the new 'interesting' seeds obtained from this particular fuzzing session are added to the accumulated seed corpus. The new seeds can potentially cover a program region that is not covered by any of the seeds in the accumulated seed corpus, thereby increasing the diversity of the accumulated seed corpus. Whenever a new commit arrives, the SRA processes the augmented accumulated seed corpus and updates it by refining redundant seeds.

3.3 Implementation

In this section, we provide the implementation details of TargetFuzz for each component presented in Figure 3·1.

BSCG. To generate the baseline seed corpus, we used four different coverage-based

fuzzers deployed as part of a fuzzing environment, namely Magma [Hazimeh et al., 2020b]. The coverage-based fuzzers are AFL [Google, 2017], Angora [Chen and Chen, 2018], MOpt-AFL [Lyu et al., 2019], and honggfuzz [Google, 2021b]. For each software that we tested, we run separate fuzzing sessions using each of the fuzzers for a certain time limit (detailed in Section 3.4). Once the fuzzing sessions are over, we collect all the coverage-increasing seeds residing in the output queues of each fuzzer and assemble them in the baseline seed corpus.

SRA. To increase the chance of detecting bugs in the program, we compiled the program source using LLVM passes that insert the sanitizer checks such as Address Sanitizer. The SRA also eliminates any seeds which made the application hang or require more than a fixed-memory limit determined by the fuzzers (which is 100MB in our case). To eliminate the duplicate seeds, the SRA utilizes the two previously discussed practices that are implemented as part of `afl-cmin` [Zalewski, 2017], an existing seed minimization tool. The SRA uses `afl-showmap` to extract the edges exercised by each seed. Note that our seed deduplication uses `afl-cmin`'s logarithmic branch hit count mode (i.e., no `-e`) when measuring branch coverage.

TSS. We implement TargetFuzz on top of AFLGo [Böhme et al., 2017a]. We use AFLGo because it is one of the state-of-the-art directed greybox fuzzers and it is open-sourced. The TSS compiles the target program by using AFLGo's LLVM passes. These LLVM passes modify the program source to include the book-keeping logic and to compute the distance of a provided seed to the given set of target sites by analyzing control-flow graphs and call graphs of the program. We leave the details of these passes in terms of the performed static analysis to the original AFLGo paper [Böhme et al., 2017a] and the github repo of AFLGo [aflgo, 2017]. The TSS also sets the target sites as the changed code lines after a commit. After the generation of an instrumented binary, we perform a dry-run with AFLGo to compute the distance

values of baseline seeds and use the same binary for fuzzing.

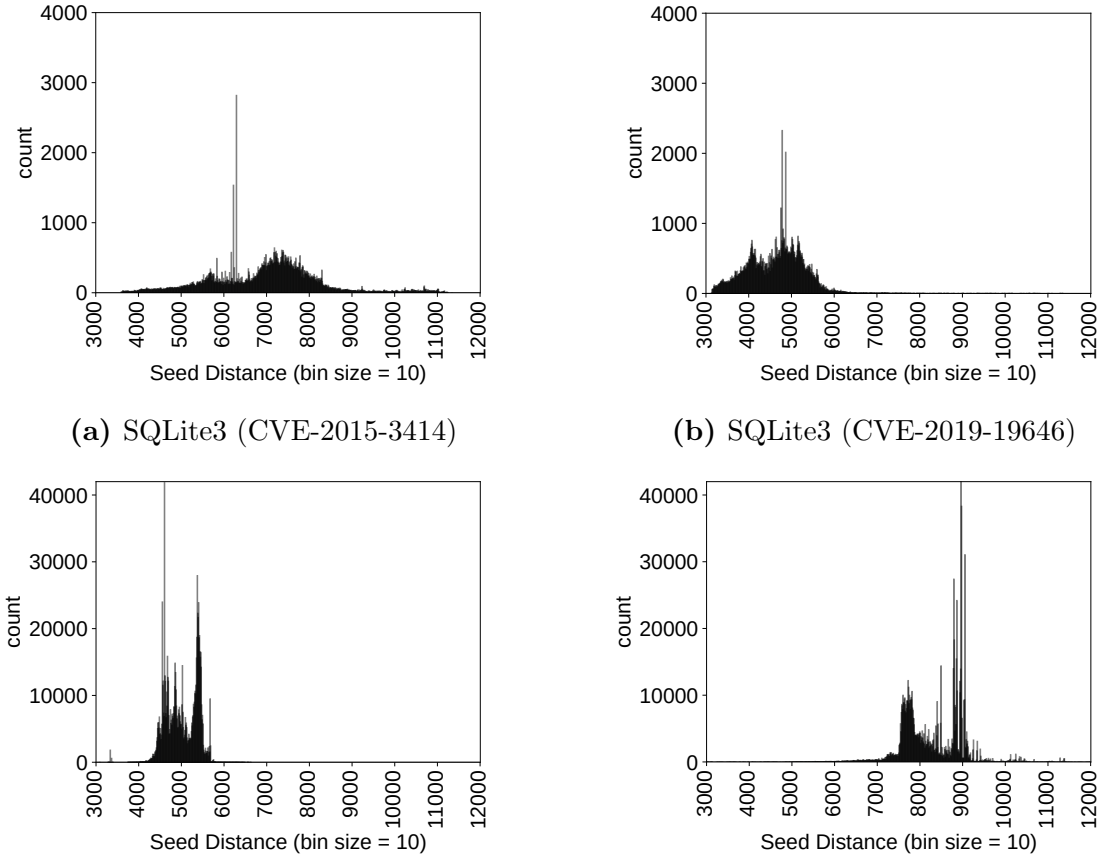
As described in Section 3.2, there is a trade-off between the quality and the quantity of seeds that comprise DART corpus. Here, we explain how we tackle this trade-off by providing the implementation details of the TSS. Specifically, we explain how TargetFuzz determines the total number of seeds that construct the DART corpus (i.e., the size of DART corpus). Before we explain our approach, we deem it useful to present the seed distance distributions for two programs (SQLite3 and Libxml2) as histograms in Figure 3·2. We choose two different target sites per program where each target site refers to code lines that introduce a known CVE. The seed distance distributions demonstrate several key points that we take into account in our approach: **(K1)** A seed distance distribution does not necessarily conform to a common probability distribution (e.g., normal distribution) as in several examples provided in Figure 3·2. Therefore, it is not feasible to forecast a seed distance value by using metrics that define a specific probability distribution such as mean and standard deviation for normal distribution. **(K2)** Seed distance distributions that belong to two different programs can significantly vary. As presented in Figure 3·2a and Figure 3·2c, SQLite3 and Libxml2 libraries show different characteristics in their seed distance distributions. Therefore, the selection strategy should ensure that it works smoothly for different types of programs. **(K3)** Seed distance distributions that belong to the same program can vary significantly when different target sites are chosen. As shown in Figure 3·2a and Figure 3·2b, even the same program (i.e., SQLite3) can present varying seed distance distributions for two different code targets. This point shows that it could be misleading to rely on program characteristics when selecting seeds. **(K4)** Seed distance distributions contain outliers as we observe in several programs. Selecting seeds from outliers can potentially misguide DGF.

Based on the provided seed distance distributions, there is no clear seed distance

cut-off value that applies to all programs and target sites. TargetFuzz proposes to use a percentile-based seed selection strategy that considers the above discussed points. TargetFuzz first calculates the k^{th} percentile of a set of seed distance values. Next, the calculated percentile value is used as a threshold when selecting the seeds from the baseline seed corpus. The TSS determines all the seeds that have a lower distance than the threshold and assembles those seeds in the DART corpus. To show the trade-off between the quantity and quality of seeds, we evaluate TargetFuzz using different percentile values. We provide the details in Section 3.4.4. Our threshold method uses percentiles for the following advantages: **(1)** Percentiles are versatile. They do not depend on a specific probability density function and can be calculated regardless of the probability distribution [Nesselroade Jr and Grimm, 2018]. As pointed out by **(K1)**, **(K2)**, and **(K3)**, this property is essential to determine a seed distance threshold for different distributions that belong to different programs and targets. **(2)** Percentiles are resistant to outliers in the data set [Bornmann et al., 2013], which occur frequently in seed distance data, as we pointed out in **(K4)**.

3.4 Evaluation

In this section, we evaluate the effectiveness and efficiency of TargetFuzz using real-world programs. We first test TargetFuzz using the fuzzing benchmark Magma on a number of real bugs to assess its impact on the bug-finding capability of a DGF tool. Specifically, we want to assess if a corpus tailored for DGF (i.e., DART corpus provided by TargetFuzz) outperforms CGF-based corpora in terms of bug-finding capability. Second, for the same set of real bugs, we measure the performance boost of a DGF tool when equipped with our distance-based DART corpus rather than CGF-based corpora. This aspect of evaluation is important to show that AFLGo can successfully expose bugs even under limited fuzzing time as real-world applications



(a) SQLite3 (CVE-2015-3414)

(b) SQLite3 (CVE-2019-19646)

(c) Libxml2 (CVE-2015-8317)

(d) Libxml2 (CVE-2017-9047)

Figure 3·2: The seed distance distributions for different code patches.

have limited testing time due to the high-frequency of commits. Last, we evaluate TargetFuzz on several commit sequences obtained from real-world git repositories to demonstrate that TargetFuzz can successfully be deployed in real-world continuous integration (CI) systems. We analyze the achieved code coverage in target code regions (i.e., lines of code modified by the commit) for each commit to understand if TargetFuzz successfully guides DGF to the target regions. Overall, our evaluation aims to answer the following questions:

RQ1: How effectively does TargetFuzz aid a DGF tool in detecting bugs (Section 3.4.4)? Does DART corpus present a clear advantage over CGF-based cor-

pora?

RQ2: Does TargetFuzz boost the bug-finding speed of a DGF tool (Section 3.4.5)?

RQ3: Can TargetFuzz effectively guide DGF tools to the target code regions (Section 3.4.6)?

3.4.1 Evaluation Dataset

To assess TargetFuzz in terms of its effectiveness in bug detection (**RQ1**) and in terms of its efficiency in bug-finding speed (**RQ2**), we use Magma [Hazimeh et al., 2020b], a fuzzing benchmark suite that includes a variety of real bugs as part of seven different real-world programs. We choose Magma in our evaluation for two main reasons. First, Magma is a ground-truth fuzzing benchmark that enables us to provide an accurate quantitative evaluation. We can easily compare our new mechanism (TargetFuzz) with prior works based on the known bugs that reside in the provided program by Magma. Second, Magma follows an approach called *forward-porting* that is well-suited for the evaluation of TargetFuzz. As detailed before, real-world projects frequently rely on small size changes. Similarly, Magma provides a separate patch file for each bug where each patch modifies a certain region of the program with a limited number of code line changes. As a reference point, the average patch size of all the Magma bugs that we used in our evaluation is 15 lines of code which are in the same order as the commit size of the real-world projects (i.e., 34.8 lines as detailed in Section 3.1). By applying these small-size Magma patches (one patch at a time) on a complex real-world program, we imitate a scenario where a recent commit modifies a certain region of the program. For fuzzing, we set the target sites as the lines modified by the imitated commit.

In our experiments, we test all the seven libraries integrated as part of Magma; libpng, LibTIFF, Libxml2, Poppler, SQLite3, OpenSSL, and PHP. It is not feasible

Table 3.2: Driver programs of libraries provided by Magma.

Library	Drivers
libpng	**read_fuzzer**, readpng
LibTIFF	read_rgba_fuzzer, **tiffcp**
Libxml2	**xml_reader_for_file_fuzz**, xmllint
Poppler	pdf_fuzzer, **pdfimages**, pdftoppm
OpenSSL	asn1, asn1parse, bignum, bndiv, client, cms, conf, crl, ct, **server**, x509
SQLite3	**sqlite3_fuzz**
PHP	**exif**, json, parser, unserialize

to directly fuzz a library, and so the common practice to overcome this is to prepare driver programs. For each library, Magma provides one or more driver programs (in total 26 drivers for 7 libraries presented in Table 3.2) that call functions in the library. Magma does not provide the name of the driver program(s) that triggers a certain bug, and so one needs to fuzz each library with all the available driver programs. Evaluating TargetFuzz (and related works) on all the bugs available in Magma (in total 118) using all the provided drivers requires $\sim 1M$ CPU hours, which is infeasible for our resources. Therefore, we rely on the proof of concepts (PoC) provided by Magma [Hazimeh et al., 2020a] to understand the driver program of each bug (in total 58 PoCs). For each library, we pick one driver program (highlighted in Table 3.2) that exposes the maximum number of bugs. In total, this adds up to 34 bugs. The selected bugs have different bug classes such as integer overflow, heap buffer overflow, uninitialized memory access, etc.

3.4.2 Infrastructure and Settings

As described in Section 3.2, TargetFuzz has a seed collection phase (i.e., baseline seed corpus generation) that creates a large seed pool with a variety of fuzzer-generated baseline seeds. To generate the baseline seeds, we fuzzed the bug-free version[3] of each

[3]Throughout the paper, we refer to a library version as bug-free if it does not include any of the ground-truth bugs provided by Magma.

library using its application driver highlighted in Table 3.2. We run four instances of each of the coverage-based fuzzers (i.e., AFL, Angora, MOPT-AFL, honggfuzz) for each library where each instance uses one core and has a time limit of 168 hours (7 days). In total, collecting baseline seeds for 7 application drivers took 18816 CPU hours ($168hours \times 4instances \times 4fuzzers \times 7drivers$).

The collected baseline seeds were used to test ground-truth bugs in Magma. Specifically, to test each bug, we first applied its corresponding patch to the clean library version. Next, AFLGo was equipped with a patch-specific DART corpus to fuzz the patched library for 24 hours. To evaluate the impact of the quantity and quality of seeds that comprise DART corpus, we used 0.1^{st},1^{st}, and 10^{th} percentiles to output three different DART corpora which we refer to as DART-0.1^{st}, DART-1^{st}, and DART-10^{th}, respectively. We used a logarithmic scale to show the impact of percentiles more explicitly. Due to the probabilistic nature of fuzzing, we run 20 fuzzing instances for each patched library and report the geometric mean of Time-to-Exposures (TTE). TTE is the total duration from the beginning of the fuzzing session until the bug is triggered. A run that did not reproduce the vulnerability within 24 hours received a TTE of 24 hours. The dedicated CPU resources for each fuzzing instance were one CPU core and 1GB of memory. AFLGo was configured with the parameters used in the original paper (`-z exp -c 4h`). We used the most recent commit of the AFLGo repo [aflgo, 2017] at the time we started conducting experiments (commit `e27a908`). All the experiments were conducted in virtual machines started on server nodes with Intel® Xeon® Gold 6132 CPUs and Ubuntu 18.04 LTS as the operating system.

We compared DART corpus with two different CGF-based seed corpora. The first corpus is provided by Magma and consists of generic seeds well-suited for CGF. Specifically, Magma sources these seeds from the library repositories, OSS-Fuzz [Google,

2021a] and the AFL repository [Google, 2017]. As shown in Table 3.1, Magma corpus is similar to the corpus used in the original AFLGo experiments [Böhme et al., 2017a]. As part of our experiments, we use Magma corpus rather than AFLGo corpus since Magma corpus is constructed more recently and therefore contains more up-to-date seeds. Since developers from both OSS-Fuzz and the tested libraries improve their seed corpora over time, using an outdated seed corpus (e.g., AFLGo corpus) could bring unfair performance benefits in our experiments.

As shown by a recent work [Herrera et al., 2021], there is no clear superiority of any available CGF-based corpus minimization tools (i.e., `OPTIMIN`, `afl-cmin`, and `MINSET`) over the others. We assembled the seeds in the second corpus by using one of these tools, specifically `MINSET` [Rebert et al., 2014], which showed promising results with CGF tools. To evaluate `MINSET`, we applied the greedy algorithm (detailed in Algorithm 2) to the accumulated seed corpus[4] in Figure 3·1 by removing the TSS component. In total, we spent 81600 CPU hours ($24hours \times 20instances \times 5corpora \times 34bugs$) to compare the effectiveness of distance-based DART corpora with CGF-based seed corpora.

3.4.3 Seed Corpus Size

This section provides the details related to seed corpora sizes. In the second column of Table 3.3, we report the baseline number that CGF-based fuzzers generated by fuzzing bug-free version of each library in Magma. Additionally, we calculate the corpora sizes of DART corpus for three different percentile values (column 3-5) and `MINSET` corpus (column 6) after applying their corresponding seed selection algorithms to the baseline seed corpus. As shown in Table 3.3, the libraries that we tested with TargetFuzz resulted in different sizes of baseline seed corpora. This observation is important to

[4]It is evident that seeds with high seed distance values are less likely to lead to the target sites. Therefore, for a more fair comparison, we eliminate any seed with a distance value higher than the geometric mean of the population from baseline seed corpus.

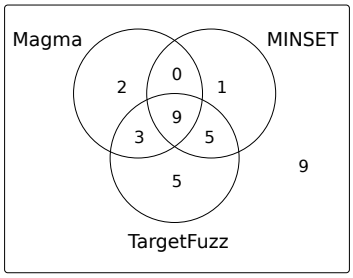
Table 3.3: Corpora size.

Library	Baseline Seed #	DART Seed #			MINSET Seed #
		0.1^{st}	1^{st}	10^{th}	
libpng	28,048	51	238	2425	172
LibTIFF	127,425	48	315	782	76
Libxml2	1,095,819	1583	9663	99503	496
Poppler	164,806	113	221	944	133
SQLite3	119,323	234	1659	11938	639
OpenSSL	39,949	133	336	1999	187
PHP	21,556	17	167	1668	77

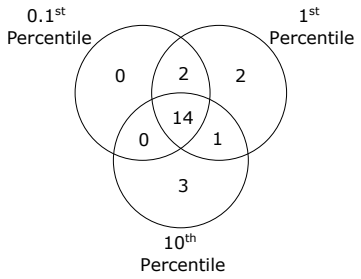
demonstrate that our distance-based seed selection strategy is effective not only for certain sizes of seed pools. In fact, the distance-based seed selection strategy worked effectively for baseline seed corpora with different scales. As an example, TargetFuzz detected almost all the ground-truth bugs (3 out of 3 in OpenSSL and 4 out of 5 in Libxml2) in two libraries which differ by orders of magnitude in their baseline seed number (i.e., OpenSSL(∼40K) and Libxml2 (∼1M)). As we detail in Section 3.4.2, we allocated the same fuzzing time limit for all the libraries to create each baseline seed corpus. Therefore, there is a difference in the number of collected baseline seed results based on the different characteristics of libraries.

Table 3.3 demonstrates the corpus minimization results for `MINSET`, DART-0.1^{st}, DART-1^{st}, and DART-10^{th} corpora. Due to varying seed distance distributions of different programs (see Section 3.3), the same percentile value can result in different DART seed numbers. For instance, libpng and PHP libraries have different DART seed numbers for the 0.1^{st} percentile although they have similar baseline seed numbers. Other than Libxml2 library, the size of `MINSET` corpus is always between the size of DART-0.1^{st} and the size of DART-1^{st}.

Additionally, DART-10^{st} corpus contains at least an order of magnitude more seeds than `MINSET` corpus for all the libraries. As detailed in the next sections and shown by prior work [Herrera et al., 2021], the corpus size by itself is not the ultimate metric to evaluate a seed selection strategy. Different seed selection strategies can



(a) The quantity of found bugs for different corpora.



(b) The impact of percentiles on bug detection.

Figure 3·3: Unique bug detection.

result in different corpus sizes and the increase in corpus size does not necessarily increase the effectiveness of a fuzzer as shown by our work and also by Herrera et. al [Herrera et al., 2021]. However, it still helps us to interpret certain aspects of our evaluation results, especially when interpreting why some of the corpora achieve/fail to discover certain bugs.

3.4.4 Unique Bug Detection

Overall results. As discussed by several works [Manès et al., 2019, Klees et al., 2018], the bug-finding capability of a fuzzer should be the ultimate metric for the evaluation. To this end, we evaluate the bug-finding capability of AFLGo when equipped with different types of corpora; DART corpus, Magma corpus, and `MINSET` corpus. Specifically, we equip AFLGo with each type of corpora separately, fuzz each buggy program version, and check if at least one of the twenty fuzzing sessions triggers the bug.

Table 3.4: Speedups achieved by Magma, MINSET, and DART corpus. We provide TTE (in seconds) for each bug when tested with AFLGo using different corpora.

Library	Bug No	Magma	MINSET	DART			Speedup			
				0.1^{st}	1^{st}	10^{th}	MINSET	0.1^{st}	1^{st}	10^{th}
libpng	AAH001	86400	86400	84804	86079	86400	1.00	1.02	1.00	1.00
	AAH007	86400	277	86400	15	69	312.20	1.00	5,760	1,261
	AAH008	59910	86400	68168	62491	86400	0.69	0.88	0.96	0.69
LibTIFF	AAH010	86400	79564	86400	86400	86400	1.09	1.00	1.00	1.00
	AAH014	37835	10	10	10	10	3,784	3,784	3,784	3,784
	AAH017	52326	74227	49053	42116	42116	0.70	1.07	1.24	1.24
	AAH020	57084	21533	34219	13799	16170	2.65	1.67	4.14	3.53
	AAH022	81682	818	81649	13795	57948	99.89	1.00	5.92	1.41
Libxml2	AAH024	86400	86400	1381	33887	86400	1.00	62.55	2.55	1.00
	AAH025	86400	86400	86400	82104	86400	1.00	1.00	1.05	1.00
	AAH026	86400	86400	86400	18115	74342	1.00	1.00	4.77	1.16
	AAH032	86400	82128	34334	4674	35761	1.05	2.52	18.49	2.42
Poppler	AAH043	86400	84461	51594	51594	71300	1.02	1.67	1.67	1.21
	AAH045	86400	86400	86400	86400	85317	1.00	1.00	1.00	1.01
	AAH050	86400	86400	647	661	86400	1.00	133.49	130.76	1.00
	JCH201	44299	86400	86400	86400	86400	0.51	0.51	0.51	0.51
	JCH209	86400	6028	6219	6321	5963	14.33	13.89	13.67	14.49
	JCH210	86400	17533	21225	20075	82833	4.93	4.07	4.30	1.04
SQLite3	JCH215	53302	47081	26327	26327	26327	1.13	2.02	2.02	2.02
	JCH228	21805	86400	86400	86400	86400	0.25	0.25	0.25	0.25
	JCH232	79222	86072	86400	86400	82545	0.92	0.92	0.92	0.96
OpenSSL	AAH055	15721	86400	236	1777	75112	0.18	66.69	8.85	0.21
	AAH056	610	95	367	367	1315	6.44	1.67	1.67	0.46
	MAE115	80898	8499	86400	86400	26476	9.52	0.94	0.94	3.06
PHP	MAE016	800	15	3908	3778	3541	53.33	0.20	0.21	0.23
Geo. mean	-	45225	14968	16744	11220	23035	3.02	2.70	4.03	1.96

Our main goal here is to show that a DGF-based corpus outperforms a CGF-based corpus when used with a DGF tool. We present our results as a Venn diagram in Figure 3·3a. Unfortunately, AFLGo could not trigger 9 out of 34 bugs with any of the seed corpora[5]. DART corpora (i.e., DART-0.1st, DART-1st, and DART-10th) triggered 22 out of 25 remaining bugs in total where we provide the details of the sensitivity analysis later in this section. Magma and `MINSET`-based corpus could trigger 14 and 15 bugs, respectively. In total, DART corpora discovered 5 unique bugs while Magma and `MINSET`-based corpora discovered 2 and 1 unique bugs, respectively. Overall, our experimental results demonstrate that a seed corpus created with a CGF-based seed selection strategy (i.e., `MINSET`) can perform poorly when used with a DGF tool. Moreover, a seed corpus well-suited for CGF (i.e., Magma corpus) can fail to trigger bugs with a directed greybox fuzzer. By taking the seed distance metric into account in the seed selection strategy, the bug detection capability of DGF improves as demonstrated with TargetFuzz.

Sensitivity analysis. As discussed in Section 3.3, TargetFuzz uses percentiles to determine the size of DART corpus. By choosing three different percentile values with different orders of magnitude, we evaluate how the quality and quantity of seeds in DART corpus impact the bug-detection capability of a DGF tool. We demonstrate the impact of the chosen percentile value on the effectiveness of bug detection as a Venn diagram in Figure 3·3b. Our results demonstrate that the size of DART corpus clearly affects the bug-finding capability of DGF. Specifically, DART-0.1st corpus detected the least number of bugs although it is constructed with the closest seeds to the target sites. In fact, DART-1st corpus triggered all 16 bugs that DART-0.1st corpus triggered and three more bugs that were not triggered by 0.1st percentile corpus. The main reason behind the ineffectiveness of DART-0.1st corpus is the limited variety of

[5]The library and id of each bug are as follows; Libxml2 (`AAH035`), Poppler (`AAH049`, `JCH212`), SQLite3 (`JCH216`, `JCH223`, `JCH226`, `JCH227`), and PHP (`MAE008`, `MAE014`).

seeds compared to DART-1^{st} and DART-10^{th}. A clear example is `AAH007` bug where a simple bit-flip operation of a baseline seed easily triggers the bug. Since DART-0.1^{st} does not include this seed as part of the corpus (i.e., low variety), it failed to trigger this bug after 24 hours of fuzzing. In fact, five out of six bugs, which were not triggered with DART-0.1^{st} corpus, were discovered with less than two hours of fuzzing after a set of mutations on one of the seeds that resides in either a DART-1^{st} corpus or a DART-10^{th} corpus. While DART-1^{st} and DART-10^{th} corpora did not show clear superiority (4 and 3 unique bugs, respectively) over each other in terms of bug detection capability, they differ significantly in their time-to-exposure as detailed in Section 3.4.5.

3.4.5 TTE Speedups

We also demonstrate that TargetFuzz helps DGF to reduce the time-to-exposure of bugs, which is important especially when the project has limited time for fuzzing. To show the efficiency of the DART corpus over the Magma corpus and the `MINSET` corpus, we report the TTE of bugs (in seconds) and the speedups in Table 3.4. When calculating the speedups, we use the results obtained with Magma corpus as a baseline since it performed the worst. In summary, `MINSET`, DART-0.1^{st}, DART-1^{st}, and DART-10^{th} resulted in 3.02×, 2.7×, 4.03×, 1.96× speedup over Magma corpus, respectively.

Table 3.4 presents that `MINSET` corpus (4^{th} column) outperformed Magma corpus (3^{rd} column). Within the first 10 hours (36000 seconds) of fuzzing, `MINSET` corpus triggered several bugs (e.g., `AAH007`, `AAH014`, `AAH022`) which could not be triggered by Magma corpus. Evidently, the efficiency of `MINSET` corpus over Magma corpus mainly comes from the usage of baseline seeds. As an example, `MINSET` corpus easily triggered `AAH014` after AFLGo applied a single bit-flip on a seed selected from the baseline seed corpus.

The performance benefit of DART corpus over `MINSET` corpus depends on the percentile value that TargetFuzz uses. DART-1^{st} corpus leads to best results among the three percentile-based corpora and it outperforms `MINSET` corpus in bug-finding speed averaged for 25 bugs. Compared to `MINSET`, it discovered several bugs significantly faster (e.g., `AAH032`, `AAH017`, and `JCH215`) and was able to discover more bugs within the time limit (e.g., `AAH050`, and `AAH055`). These examples clearly present the benefit of using distance metric in the seed selection strategy rather than only relying on code coverage. However, the important observation here is that DART-0.1^{st} and DART-10^{th} are less efficient than `MINSET` although the seeds in those two corpora are also selected using a seed distance metric.

The advantage of DART-1^{st} corpus over DART-0.1^{st} corpus is the variety and quantity of seeds. For instance, for some of the seed distributions, we observe that the distance values in the 1^{st} percentile are very close to each other. In this scenario, choosing 0.1^{st} over 1^{st} percentile prevents the fuzzer from using a high number of 'close' seeds as part of the seed corpus, and therefore it is less likely for the fuzzer to explore new paths with mutations. The scenarios discussed above for 0.1^{st} and 1^{st} percentiles comparison are valid for 1^{st} and 10^{th} percentiles as well. As we demonstrate in Figure 3·3b, DART-10^{th} corpus identified three unique bugs (i.e., `AAH045`, `JCH232`, `MAE115`) that could not be triggered by DART-0.1^{st} and DART-1^{st} corpus. While DART-0.1^{st} and DART-1^{st} corpus could not trigger these bugs due to the limited variety of seeds in their corpora, DART-10^{th} triggered it by a sequence of mutations on a seed with a distance value between 1^{st} and 10^{th} percentile.

Overall, our experimental results show that there needs to be a balance between the quantity and the quality of seeds. While a couple of bugs could be identified by only DART-10^{th}, the effectiveness of DART-1^{th} corpus is the highest in terms of bug-finding capability. As the number of seeds that construct the seed corpus

increases, a DGF tool allocates less time for each seed. Therefore, the fuzzer performs a lower number of mutations on each seed (especially for DART-10^{th}). As an example from Table 3.3, AFLGo starts with 99500 and 9663 seeds (on average) for Libxml2 library when 10^{th} and 1^{st} percentiles are chosen. The $10\times$ difference in corpus size significantly reduces the total time that the fuzzer spends on each seed including seeds that are likely to lead to the buggy region with mutations. This increases TTE significantly for several Libxml2 bugs including `AAH024`, `AAH026`, `AAH032`. Similar scenario is valid for bugs residing in other libraries such as `AAH050` from Poppler, `AAH055` from OpenSSL.

3.4.6 Continuous Fuzzing

In addition to the Magma benchmark, we tested TargetFuzz on several commit sequences obtained from the real-world project repositories. The goal of this experiment is to show that TargetFuzz can successfully be deployed as part of CI systems. For instance, Google's OSS-fuzz has a service that builds a project using its source code at a particular commit and subsequently runs a fuzzing session. Currently, OSS-fuzz includes only coverage-based fuzzers [Google, 2019] (afl, honggfuzz, libFuzzer) and provides a corpus minimization tool tailored for coverage-based fuzzers [Google, 2019]. TargetFuzz outputs DART corpus tailored for DGF; therefore, it can be potentially used as part of OSS-fuzz if a directed greybox fuzzer like AFLGo is actively used and maintained in the OSS-fuzz infrastructure.

Since there is no available ground-truth for project repositories, we rely on code coverage. Specifically, we use line coverage as the metric since the lines of code need to be executed in order to find a bug. The details of our evaluation data set are provided in Table 3.5. We used the latest commits from the master branch of three different libraries SQLite3, LibTIFF, libpng at the time we started the experiments. The commit numbers are provided in the second column. We preferred these libraries

Table 3.5: Total line coverage percentage achieved by fuzzing SQLite3, LibTIFF, and libpng over a commit sequence.

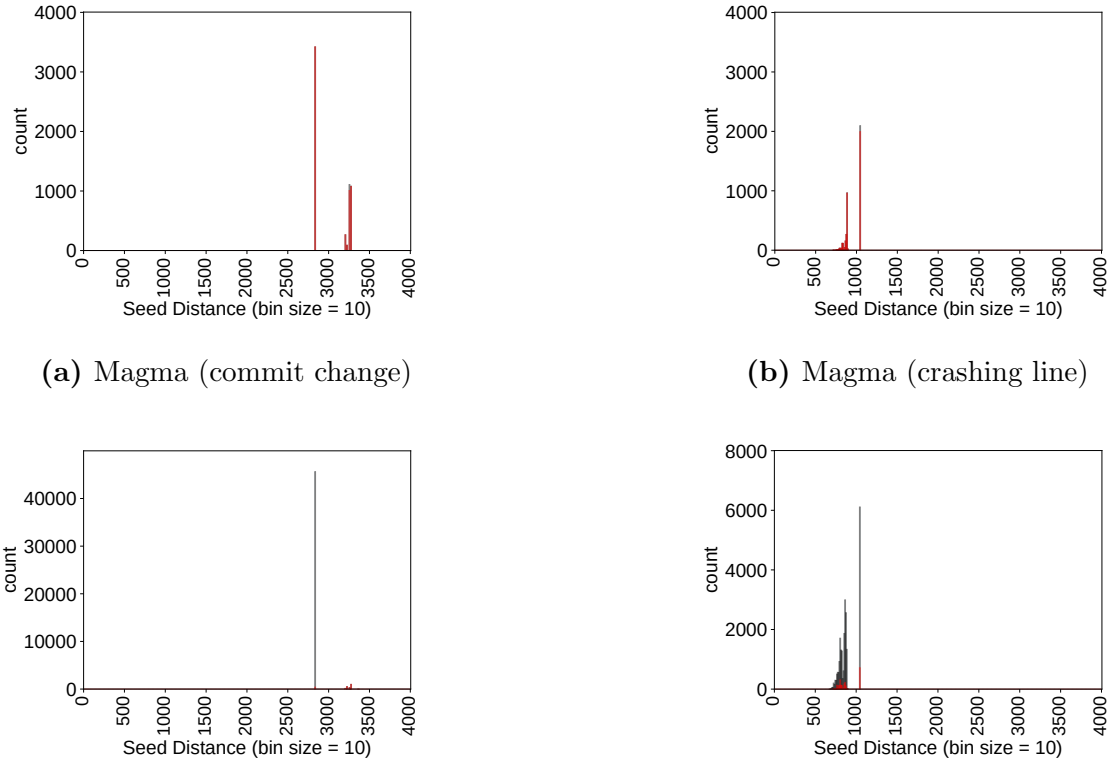
Library	Commit	Line Coverage (%)		Total Line #
		Magma	DART	
SQLite3	14c4d42	82.14	100	28
	c00727a	0	100	21
	bf7f3a0	100	100	10
	542812	0	80	10
	7cc73b3	0	100	5
	be12083	10	100	10
Average	-	32.02	96.67	13.16
LibTIFF	4ecf751	0	0	10
	de7617a	93	100	15
	f0f68dc	66.67	66.67	3
	120aa39	0	0	54
	86a8232	0	0	2
	1c7e305	100	100	2
Average	-	43.33	44.44	14.3
libpng	a37d483	0	0	11
	3796518	100	100	1
	c4bd411	0	100	1
	eb67672	0	100	2
Average	-	25	75	2.1

because of their lower compilation times when generating AFLGo-specific binaries. As a reference point, it takes ∼6 hours to compile OpenSSL and PHP while it is around 20 minutes for LibTIFF. The commit numbers are listed from newest (top) to oldest (bottom) for each library (e.g., `14c4d42` and `be12083` are the newest and the oldest commits for SQLite3, respectively). The time limit is determined for each repository based on the commit frequency to that repository. Specifically, we dedicated 12, 24, and 24 hours to SQLite3, LibTIFF, and libpng libraries, respectively. After fuzzing each commit with a commit-specific DART corpus, we collected all the seeds that increased coverage in the fuzzing sessions and combined them with the available accumulator seed corpus as described in Section 3.2. For this experiment, we used Magma corpus as a comparison point to DART-1^{st} corpus. As described in Section 3.4.2, Magma corpus contains a subset of seeds sourced from the actual repositories. Therefore, our evaluation aims to compare the effectiveness of AFLGo

when equipped with a seed corpus sourced from the actual repository and DART corpus.

The achieved line coverage by Magma and DART corpora are provided under the third and fourth columns of Table 3.5, respectively. The results present the advantage of using DGF-based DART corpus over a generic CGF-based corpus. For SQLite3 library, DART corpus achieved 3× more line coverage over Magma corpus (32% and 96%). For several commits, Magma corpus failed to reach the target code lines. Similar observations apply to libpng library as well, where DART corpus achieved 3× more line coverage over Magma corpus (25% and 75%). The line coverage results for these two libraries clearly signify the contribution of TargetFuzz when testing a certain program region under a limited fuzzing time.

The coverage results of DART corpus for LibTIFF do not present a clear advantage over Magma corpus. However, interestingly, AFLGo detected two unique bugs (stack overflow and heap-based buffer overflow) when fuzzing `1c7e305`, `120aa39`, and `4ecf751` commits only with DART corpus. We confirmed that the bugs reside in the most recent commit of the repository as well. We reported the stack overflow bug to the developers and also observed that the heap-based buffer overflow bug was an old bug reported by another group in the past. Our further analysis showed that none of the bugs reside in the modified lines of the code by the commits. To realize why using DART corpus resulted in bug detection while using Magma corpus failed, we plotted the seed distributions in Figure 3·4. Specifically, we provide four different seed distribution histograms that are generated using seeds collected from AFLGo's fuzzing sessions (including seeds that form initial seed corpus) for Magma corpus and DART corpus. Figure 3·4a and Figure 3·4c present seed distributions when we set the modified lines by `4ecf751` commit as targets. Figure 3·4b and Figure 3·4d show seed distributions when the buggy line that caused the stack overflow is set as the



(a) Magma (commit change)

(b) Magma (crashing line)

(c) DART (commit change)

(d) DART (crashing line)

Figure 3·4: The seed distance distributions for Magma and DART corpus when setting targets as modified lines by `4ecf751f` commit and crashing line of buffer overflow bug.

target. In each figure, we highlighted seeds generated during fuzzing with red.

The histograms provide several takeaways. First, for both Magma and DART corpus, the collected seeds are significantly closer to the crashing line which can be observed from the seed distance values (e.g., seed distances in Figure 3·4c and Figure 3·4d for DART corpus). Therefore, some of the seeds reached the buggy line of the code during fuzzing with DART corpus and eventually triggered the bug. Second, when using Magma corpus as an initial seed corpus, AFLGo generated a variety of seeds to cover certain program regions which were already covered by some of the seeds in DART corpus. Indeed, in Figure 3·4d, we see a subset of seeds generated during the fuzzing session (highlighted with red) accumulating around 800 seed distance.

While AFLGo spends the limited fuzzing-time to generate seeds around ∼800 when equipped with Magma corpus (see Figure 3·4b), DART corpus provides seeds with ∼800 seed distance as part of the initial seed corpus. Therefore, AFLGo is served with seeds closer to the buggy line by DART corpus when the fuzzing session starts, and eventually performing a sequence of mutations on one of the seeds with seed distance value ∼800 resulted in discovering the bug.

3.5 Summary

In this work, we present TargetFuzz, a mechanism that equips DGF tools with target-specific seed corpus. We show that the bug-finding capability of a DGF tool is limited if provided with seed corpus designed for CGF. Our observation is intuitive because CGF-based corpora misguides a DGF to the unrelated program regions. By taking the closeness of each seed into account, TargetFuzz is able to steer fuzzing towards modified code regions and expose many bugs that were not identified with CGF-based corpora.

Chapter 4

Guiding Test Generation for Hardware Designs using DGF

4.1 Introduction

A critical challenge in the RTL verification process is to generate test inputs that can cover all parts of the RTL design, and identify discrepancies between the RTL design of a hardware system and its functional specifications. Although several commercial tools and studies based on formal verification techniques [Mukherjee et al., 2015, Cadence, 2019, Chen and Mishra, 2011] have shown promising results in the verification of RTL designs, test generation using formal methods usually suffers from a state explosion problem as the size of the design increases [Chen et al., 2012, Dessouky et al., 2019]. Therefore, researchers have proposed several dynamic verification techniques that rely on RTL simulators [Verilator, 2003]. In an ideal scenario, a dynamic verification technique should generate inputs that cover every single circuit component in the design.

Unfortunately, it is challenging and time-consuming to achieve high coverage with manually crafted inputs, especially for large designs with millions of gates. Therefore, researchers proposed several CDG mechanisms [Fine and Ziv, 2003, Wagner et al., 2005, Squillero, 2005]. These mechanisms obtain coverage feedback from the DUT to automatically fine-tune the biases of test generators, for example tuning the parameters of a Bayesian network used for test generation [Fine and Ziv, 2003]. However,

these mechanisms are usually either DUT-specific or require in-depth design knowledge for the initial setup. To increase the general applicability of a test generation mechanism, a recent work named RFUZZ [Laeufer et al., 2018] leveraged GF, which is widely used for testing software applications. GF is an automated input generation technique at its core, and so it could also be used to test hardware designs. In Figure 4·1, we summarize how GF is utilized as a hardware test generation mechanism. The GF-based test generation mechanism records the achieved coverage for each test input obtained from an input queue using RTL simulations. It generates new inputs from 'interesting' inputs (i.e., the inputs that increase coverage) by employing a variety of mutations. GF requires very limited domain knowledge and manual effort for constructing the testing environment.

While RFUZZ is promising, it is not completely suitable for hardware design. Similar to software design, hardware design is an incremental process where new components are gradually added instead of designing the entire system in one step. After adding a new hardware component, not all of the older verified components of the DUT need to undergo thorough testing. Consider a scenario where an existing processor design is extended with a new feature such as a floating-point unit or a scenario where we replace the existing branch predictor in the processor with a better branch predictor. For these scenarios, the test-time budget needs to be allocated for the verification of the new/modified components and their interactions with the rest of the DUT. Unfortunately, RFUZZ is agnostic of such scenarios and it focuses on verifying the whole processor design. As a result, it spends unnecessary effort to maximize coverage of the **whole** design rather than a specific **part** of the hardware design.

In this work, we present DirectFuzz, which generates test inputs to maximize the coverage of a specific part of the hardware design. At a high-level, DirectFuzz

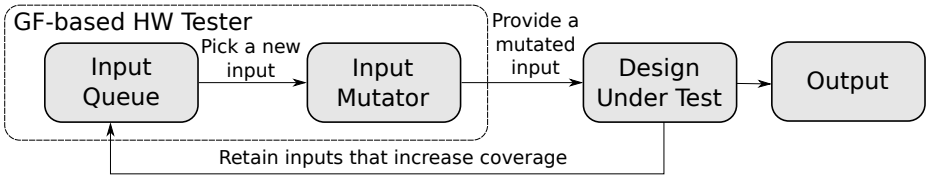


Figure 4·1: A simplified overview of a graybox fuzzer.

spends most of its time budget on reaching specific target sites instead of covering the whole design. Here, target sites refer to module instances that need to be tested. DirectFuzz achieves this goal by leveraging Directed Graybox Fuzzing (DGF) [Böhme et al., 2017a], an approach used by the software community for patch testing, and identifying special types of bugs (e.g., use-after-free). DirectFuzz differs from RFUZZ in two ways. First, RFUZZ selects the test inputs from the input queue in the order they are inserted. This increases the total time for reaching target sites when an effective test input (the one that increases the target site coverage) resides close to the rear of a long queue. DirectFuzz prioritizes those inputs that cover at least part of the target sites when choosing the next input for the DUT in order to quickly reach target sites. Second, RFUZZ and DirectFuzz differ in the number of mutations that they apply to an input. For each test input, RFUZZ applies the same number of mutations while DirectFuzz adjusts the number of mutations based on how close the input is to the target sites. The closeness of the test input to the target site is determined by a distance metric that accounts for the module instance hierarchy. DirectFuzz also generates more inputs from a test input if it achieves high coverage in the module instances that are close to the target module instance.

Overall, the main contributions of this work are as follows:

- To the best of our knowledge, we are the first to apply the notion of DGF to hardware test generation for effectively and efficiently verifying specific target sites in hardware designs.

- For generating new test inputs, we propose to use a distance metric that accounts for the instance hierarchy of a hardware design. The distance metric determines the importance of each test input.

- We demonstrate the efficacy and utility of DirectFuzz on several real-world RTL designs. DirectFuzz covers the same set of target sites up to 17.5× (on average 2.23×) faster than RFUZZ under the same time budget.

4.2 Background on RFUZZ

DirectFuzz is closely related to RFUZZ [Laeufer et al., 2018]. In this subsection, we explain how RFUZZ adopts GF into a test generation mechanism for hardware designs based on the steps provided in Algorithm 1.

RFUZZ consists of two components: a fuzzing logic, and an instrumentation suite. **The fuzzing logic** requires an initial seed corpus that consists of a set of test inputs **(S1)**. An RTL design requires a rigid test input size determined by the RTL's input port widths as opposed to a software program that usually accepts an arbitrary-sized test inputs. RFUZZ generates a bit vector of size N, where N is determined by the input port widths and the total number of test cycles. Next, the fuzzing logic chooses a test input from a queue in FIFO order **(S2)** and assigns an energy level **(S3)** which determines the number of mutations that need to be applied in **(S4)**. Note that RFUZZ uses the same energy level for each test input, thereby performing the same number of mutations. Similar to graybox fuzzers targeting software programs, RFUZZ implements several deterministic (e.g., a single bit flip at a constant offset) and non-deterministic mutations (e.g., random byte overwrite). Finally, the fuzzing logic retains any test input that increases the coverage of the RTL design **(S5-S6)**. **The instrumentation suite** is in charge of collecting coverage feedback when the current input exercises the DUT. In software testing, graybox fuzzers mostly rely

on covered branch instructions (e.g., `jump`). The branches (like if-then-else control structure or switch control structure) written in a Hardware Description Language (HDL) for an RTL design are mapped to multiplexers in the circuit. Also, there are usually multiple activated multiplexers during any cycle in hardware as opposed to a sequential software program that can only trigger one branch instruction at a time. RFUZZ takes into account these two fundamental differences in its definition of coverage metric. Specifically, RFUZZ uses *mux control coverage* metric which considers the select signal of each 2:1 multiplexer as a coverage point[1]. It instruments the DUT with additional bookkeeping logic for each multiplexer to observe the value of the selection signal when exercising the DUT with an input. The coverage feedback includes the multiplexers whose selection bits are toggled. RFUZZ defines the achieved coverage in a design as the ratio of the number of multiplexers with selection bits toggled over a total number of multiplexer selection signals in the RTL.

DirectFuzz modifies the second and third stages of Algorithm 1 to convert a gray-box fuzzer into a DGF. DirectFuzz implements an input selection scheme that gives priority to inputs that are likely to increase coverage of the target sites **(S2)**. Moreover, DirectFuzz implements a power scheduling algorithm that assigns different energy levels to the inputs **(S3)**, which leads to a different number of employed mutations on each input. These modifications enable DirectFuzz to generate test inputs tailored for specific targets in the hardware design. We provide the details of our modifications in Section 4.3.2.

4.3 Directed Test Generation For Hardware

In this section, we present DirectFuzz, our proposed DGF technique for hardware verification. In Figure 4·2, we provide the overview of DirectFuzz, which consists of

[1]RFUZZ converts any other multiplexer type such as a 4:1 multiplexer into a set of 2:1 multiplexers.

two components, **(1)** The Static Analysis Unit and **(2)** the Fuzzing Logic. The Static Analysis Unit takes the Intermediate Representation (IR) of the RTL design [Izraelevitz et al., 2017] as an input and applies several IR passes to extract information that the Fuzzing Logic will use to generate the test inputs. Specifically, the Static Analysis Unit has three main tasks: 1) Identify the coverage points in the target module instance; 2) Generate an instance connectivity graph to calculate the relative distance of each module instance with respect to the target module instance; 3) Create an instrumented DUT that includes the necessary bookkeeping logic to record coverage for each test input.

The Fuzzing Logic is in charge of choosing the next test input, assigning an energy value to the test input based on a power scheduling function, and performing input mutations. The instrumented DUT and the Fuzzing Logic communicate via a shared memory region allocated by the operating system to exchange generated inputs and the achieved coverage information per input.

4.3.1 Static Analysis Unit

Target Module Instance Selection. DirectFuzz aims to generate test inputs towards maximizing the coverage of specific target sites in an RTL design. More specifically, the target sites refer to the multiplexer selection signals that reside in a specific **module instance** chosen by a verification engineer. Note that DirectFuzz accepts a module instance as a target point rather than a module. When there are multiple module instances produced by the same module, the verification engineer needs to account for the position of each module instance with respect to the DUT to determine the target module instance. The verification engineer can determine the target module instance with a manual or an automated process. For the former, she can choose the name of the module instance if she is directly aware of a change in the RTL code of a specific module instance. For the latter, she can determine the

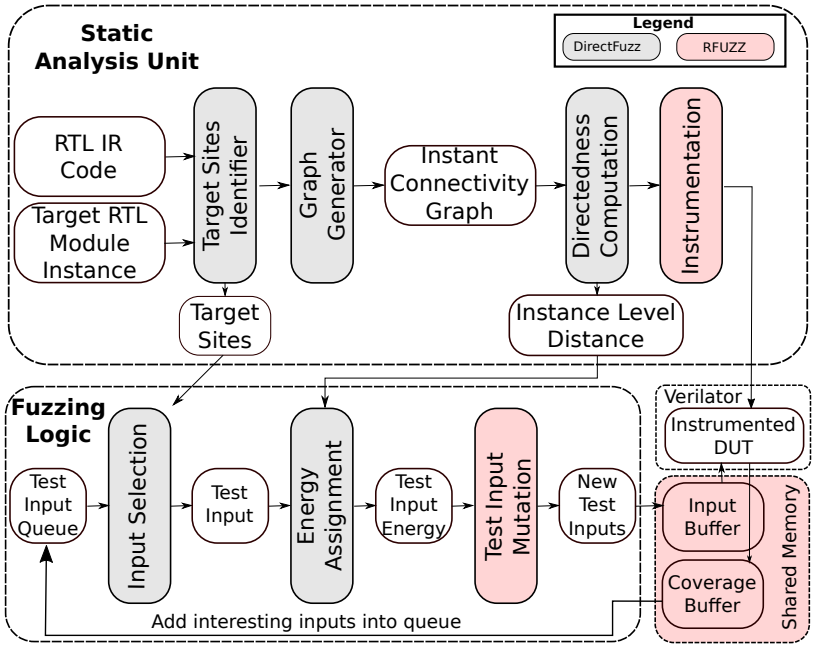


Figure 4·2: Overview of DirectFuzz. The gray boxes represent the components of DirectFuzz. The red boxes correspond to the components that DirectFuzz leverages from RFUZZ.

target module instance with software tools (e.g., git-diff and svn diff) and extract the modified instances between two versions of an RTL code.

Target Sites Identifier. As detailed in Section 4.2, the coverage points for DirectFuzz are multiplexer selection signals. To effectively generate test-cases for the target module instance, it is necessary to identify which multiplexer selection signals are covered by the current test input. First, the Target Sites Identifier (TSI) analyzes the provided RTL IR code and extracts all multiplexer selection signals in the whole RTL design. Next, TSI labels any multiplexer selection signal as 'target' if the multiplexer selection signal is part of the provided target module instance. The identified multiplexer selection signals are provided to the fuzzing logic.

Module Instance Connectivity Graph Generation. An important part of the directed test generation is to calculate the distance of each module instance in the RTL design with respect to the target module instance. In fact, if the target module

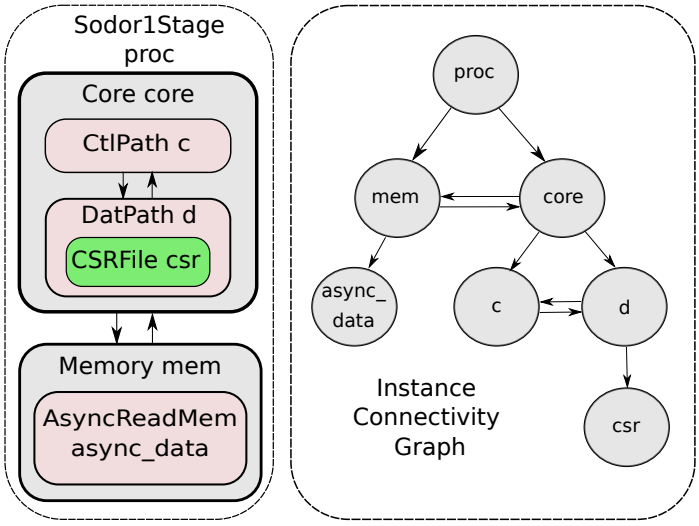


Figure 4·3: RTL design (on the left) and the corresponding module instance connectivity graph (on the right) of Sodor 1-Stage processor.

instance constitutes a small portion of the RTL design, test inputs might mostly cover the non-target multiplexer selection signals during fuzzing. Therefore, a module instance hierarchy is essential to realize which parts of the DUT are covered by the current test input and how close the covered sites are to the target sites.

The graph generator creates a module instance hierarchy of the RTL design in the form of a directed module instance connectivity graph by using the RTL IR code. In this graph, the nodes represent module instances in the RTL design while the edges represent the module instance connections. Two module instances are considered as connected if either instance is a subinstance ('child' instance) of the other instance or they are subinstances with the same 'parent' instance. The generated graph is a directed graph for two reasons: 1) it presents the hierarchical view of the overall hardware design with parent instances and child instance(s); and 2) it presents the communication direction between two module instances. For example, if instance A provides data to the input ports of instance B but not vice versa, the direction of the edge should be only from A to B.

To understand the process of graph generation, we use an example 1-Stage pro-

cessor (Sodor [Ucb-bar, 2015]) with its corresponding module instance connectivity graph (see Figure 4·3). The processor consists of seven module instances – `proc`, `mem`, `core`, `c`, `d`, `async_data`, `csr`, one instance of each of the `Sodor1Stage`, `Memory`, `Core`, `CtlPath`, `DatPath`, `AsyncReadMem`, `CSRFile` modules, respectively. As shown in Figure 4·3 (on the right), any connection between a child instance and its parent instance is represented with a one-way edge (e.g., from `proc` to `mem` and from `proc` to `core`). We use directed edges to represent the connections between the child instances (for example, `c` and `d`).

Directedness Computation. DirectFuzz calculates the instance level distance of the RTL design to determine if a test input covers multiplexer selection signals that are closer to the target sites. Our intuition is that mutating an input that covers multiplexer selection signals from the instances that directly interact with the target instance is likely to increase coverage in the target instance. For example, if the target site is the `csr` instance in Figure 4·3, an input that covers multiplexer selection signals in `d` is more likely to result in an increase in the multiplexer selection signal coverage of our target `csr` given that `d` and `csr` are directly connected to each other.

Based on this intuition, we define a metric called instance level distance d_{il} for multiplexer selection signal m with respect to target instance I_t as

$$d_{il}(m, I_t) = \begin{cases} undefined & \text{if } S(I_t, I_m) = \emptyset \\ S(I_t, I_m) & \text{otherwise} \end{cases} \tag{4.1}$$

where I_m is the module instance in which m resides and $S(I_t, I_m)$ is the number of edges along the shortest path between I_m and I_t in the instance connectivity graph. Note that all the multiplexer selection signals that reside in the same module instance are assigned the same instance level distance. The instance level distance of a multiplexer selection signal is undefined if its corresponding module instance cannot

reach the target instance. The multiplexer selection signals in the target instance are assigned to zero instance level distance. The instance level distances are used by the Fuzzing Logic to calculate the energy of a test input. We provide the details in Section 4.3.2.

4.3.2 Fuzzing Logic

The Fuzzing Logic is in charge of selecting the current input, assigning energy to the selected input, and performing mutations on the selected input to generate new test inputs. As explained before, we use the same test input mutation mechanism implemented by RFUZZ. However, we modify the input selection mechanism and add an energy assignment mechanism in the Fuzzing Logic (see Figure 4·2) to adapt the notion of DGF to the hardware test generation problem. We provide the details of our changes below.

Input Prioritization. RFUZZ strictly chooses the next input from the input queue, which has a FIFO ordering. Unfortunately, this is not ideal when targeting a specific instance in the RTL design. Not all inputs should have the same priority because only a subset of the generated inputs increase the coverage in the target module instance. Moreover, if an input leads to a new coverage point in the target instance, it is likely that the mutated inputs from that specific input will lead to new coverage points in the target instance. Therefore, the inputs should be stored in a priority queue to assign higher priorities to the inputs that have a higher chance of increasing coverage in the target instance. DirectFuzz implements an additional priority queue to separately store the test inputs that covered at least one multiplexer selection signal in the target module instance. Inputs in this priority queue are always picked (in FIFO order) before picking any inputs from the regular queue. If the priority queue is empty, DirectFuzz uses an input from the regular queue in FIFO order.

Power Scheduling. Power scheduling determines the number of mutations that

need to be applied to the current input. During the test input generation, we apply power scheduling on a selected input based on a dynamically-computed *input distance*, which is the distance of an input i to the target instance I_t. The intuition is that if the current input covers multiplexer selection signals closer to the target site, more mutations on this input should help in increasing coverage in the target instance. Below we present the formal definition of the input distance metric and a power scheduling function that relies on the input distance.

The input distance $d(i, I_t)$ is computed as

$$d(i, I_t) = \frac{\sum_{m \in C(i)} d_{il}(m, I_t)}{|C(i)|} \tag{4.2}$$

where $C(i)$ is the set of all multiplexer selection signals that the input i covered in the RTL design. Note that $d_{il}(m, I_t)$ is defined for all $m \in C(i)$. The range of $d(i, I_t)$ is $[0, d_{max}]$, where d_{max} represents the distance between the I_t and the instance with the largest 'shortest path' to I_t. If the input covers only the multiplexer selection signals in the instance farthest from I_t, the input distance will be d_{max}. If the input covers multiplexer selection signals only from the I_t, the input distance will be assigned zero.

The power scheduling function is defined as

$$p(i, I_t) = maxE - \left((maxE - minE) \cdot \frac{d(i, I_t)}{d_{max}} \right) \tag{4.3}$$

where I_t represents the target instance. $minE$ and $maxE$ are the constant lower and upper energy limits. When $d(i, I_t)$ is zero and d_{max}, the assigned energy will be $maxE$ and $minE$, respectively. Overall, the power scheduling function favors the inputs with lower distances, thereby performing more input mutations on those inputs.

DirectFuzz assigns a power coefficient to each input i based on the power scheduling function. We calculate the input energy by multiplying the power coefficient

of an input with the default mutation number provided by RFUZZ. The input energy determines the total mutation number for the input (e in Algorithm 1). In essence, DirectFuzz adjusts the total number of mutations employed by each mutator in RFUZZ based on the computed power coefficient of each input. For example, if the current mutator performs N random bit flips in RFUZZ, the same mutator performs $N \times p$ flips in DirectFuzz. By adjusting the power coefficient of inputs, DirectFuzz can prevent excessive mutations that do not increase the coverage in the target.

Random Input Scheduling. Our power scheduling function always favors the inputs with lower distances. Unfortunately, this greedy approach may get stuck in a local minimum instead of reaching a global minimum, thereby not covering the target multiplexer selection signals. For example, when the target is `csr` for the RTL design in Figure 4·3, DirectFuzz can favor the inputs that increase coverage in `c` by considering multiplexer selection signals in `c` as global minimum. To prevent DirectFuzz from getting stuck at local minimums, we randomly pick an input with low energy value after an interval and schedule this input with its default energy value (i.e., `p` is set to 1). The interval of random input scheduling is determined by the coverage progress in the target instance. DirectFuzz runs the random input scheduling mechanism if there is no coverage increase in the target module instance for the scheduled last ten inputs.

4.4 Evaluation

We implemented DirectFuzz by extending the open-source RFUZZ repository [Ekiwi, 2020]. For a fair head-to-head comparison between RFUZZ and DirectFuzz, we used all the RTL designs evaluated by RFUZZ [Laeufer et al., 2018]. These RTL designs include several peripheral IPs (e.g., SPI [Sifive, 2020], I2C [Sifive, 2020], UART [Sifive, 2020]), a DSP block (FFT) [Ucb-art, 2020], a Pulse Width Modulator (PWM) [Sifive,

2020], and three (1-stage, 3-stage, 5-stage) in-order 32-bit RISC-V processors [Ucb-bar, 2015]. We used FIRRTL [Izraelevitz et al., 2017] as the intermediate representation for the RTL designs. Both RFUZZ and DirectFuzz are compatible with any design expressed in FIRRTL form. We applied several FIRRTL passes to the RTL IR code in order to identify target sites, generate instance connectivity graph, and perform directedness computation. In order to collect coverage feedback, we used the FIRRTL passes provided by RFUZZ [Ekiwi, 2020].

We conducted the experiments using Verilator [Verilator, 2003] on an Intel® Core™ i7-9700 3 GHz machine. We ran each experiment for 24 hours. If all multiplexer selection signals in the target were covered in less than 24 hours, we terminated those experiments early. Due to the probabilistic nature of fuzzing, we repeated each experiment ten times and report the geometric mean for each design in Table 4.1. To demonstrate the variation across ten runs, we provide the box (25%ile) and whisker (75%ile) plot for each design in Figure 4·4.

4.4.1 Evaluation Dataset and Target Module Instance Selection

We determined the target module instances from small designs (UART, SPI, PWM, I2C, and FFT) based on the number of multiplexer selection signals. For these designs, we determine the module instances with the highest number of multiplexer selection signals as targets since any change in these RTL designs will likely modify these module instances. To determine the target module instances for RISC-V processors (i.e., Sodor1Stage, Sodor3Stage, and Sodor5Stage), we extracted the area of each module instance in three processor designs. We picked two module instances from the processor cores – one with the smallest area, `CtlPath`, and one with the largest area, `CSR`, as targets to show the impact of the target instance size on the performance of DirectFuzz. We report the cell percentage of `CtlPath` and `CSR` instances under the 'Target Instance Cell Percentage' column in Table 4.1. For our area estimation, we

synthesized the benchmark circuits using Synopsys Design Compiler [Synopsys, 2019] for GlobalFoundries 22nm process at 1 GHz clock frequency.

4.4.2 Results

In this subsection, we show the efficiency of DirectFuzz over RFUZZ where the goal is to reduce the total testing time for generating test inputs that cover a specific set of multiplexer selection signals in a target module instance. We present the experimental results in Table 4.1. The first column lists the design name, while the second column shows the total number of module instances in that design. We provide the name of the target instance(s) within the design in the third column with their corresponding total number of multiplexer selection signals listed in the fourth column. The achieved coverage ratio in each target instance for RFUZZ and DirectFuzz are presented in the sixth and eighth column, respectively. The total time required for achieving each coverage ratio is provided under the seventh and ninth columns for RFUZZ and DirectFuzz, respectively. On average, DirectFuzz covers the multiplexer selection points 2.23× faster than RFUZZ for the target instances in Table 4.1 thanks to its directed test generation mechanism.

DirectFuzz achieves the highest performance improvement (17.5×) in UART when the target is `Tx`. DirectFuzz achieves the lowest performance improvement (1.08×) in Sodor1Stage RISC-V processor when the target is `CSR`. Here, the difference in the total number of multiplexer selection signals is the key reason behind the speedup difference. It is intuitive that covering a low number of target sites in a small design (e.g., UART) is easier than covering a high number of target sites in a relatively more complex design (e.g., a processor like Sodor5Stage). However, even a small speedup in a complex design could save a lot of testing time. For example, DirectFuzz saves 834 seconds of testing time to cover all the target points in Sodor5Stage (`CtlPath`).

Table 4.1: The experimental results of RFUZZ and DirectFuzz on 12 module instances from 8 RTL designs.

Benchmark	Total # of Instances	Target Instance	Total # of Mux Selection Signals	Target Instance Cell Percentage	RFUZZ		DirectFuzz		Speedup
					Coverage	Time(s)	Coverage	Time(s)	
UART	7	Tx	6	5.1%	100%	7.35	100%	0.42	17.5
		Rx	9	6.9%	88.89%	4.95	88.89%	1.71	2.89
SPI	7	SPIFIFO	5	34.4%	100%	55.84	100%	31.75	1.76
PWM	3	PWM	14	26.9%	100%	12.79	100%	100%	5.87
FFT	3	DirectFFT	107	87%	13%	0.075	13%	0.073	1.03
I2C	2	TLI2C	65	31%	98%	13.73	98%	8.49	1.61
Sodor1Stage	8	CSR	93	16.6%	96.77%	500.56	96.77%	463.63	1.08
		CtlPath	68	0.3%	100%	694.42	100%	526.53	1.32
Sodor3Stage	10	CSR	90	16.4%	98.89%	568.05	98.89%	446.29	1.27
		CtlPath	66	0.3%	100%	1283.4	100%	1034.86	1.24
Sodor5Stage	7	CSR	93	3.1%	96.77%	817.58	96.77%	322.19	2.54
		CtlPath	70	0.1%	100%	1227.35	100%	393.15	3.12
Geo. Mean	5	-	37	5.43%	82.87%	152	82.87%	29	2.23

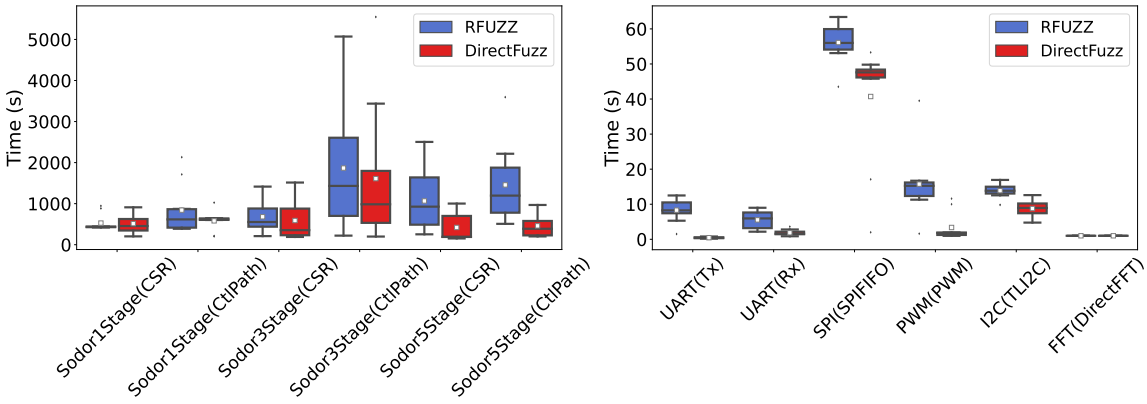


Figure 4·4: Whisker plots of DirectFuzz and RFUZZ for various RTL designs.

To assess the effectiveness of DirectFuzz on different sizes of target instances, we pick two target module instances with varying target instance cell percentages for each one of the three RISC-V processors. Overall, we did not observe a direct correlation between the target module instance cell percentage and the speedup to cover the same set of target points. On the one hand, DirectFuzz covers all targets in `CtlPath` faster than RFUZZ (up to 3.12×) for all three RISC-V processors even though this instance occupies a small area (0.1%) of the overall processor design. On the other hand, DirectFuzz achieves 1.27× speedup to cover the same set of target points on `CSR` target of Sodor3Stage, which has a relatively high cell percentage (e.g., 16.4%).

We report the coverage progress of RFUZZ and DirectFuzz over time for all the designs in Figure 4·5. The coverage progress is averaged over ten runs. The target instance for each design is provided inside the parenthesis of figure titles. For several benchmarks (such as UART, PWM, and Sodor5Stage for CSR target), the benefit of DirectFuzz over RFUZZ is clear based on the coverage progress. For example, DirectFuzz reaches the peak coverage more rapidly than RFUZZ for the UART benchmark (for both `Tx` and `Rx` target instances). RFUZZ's coverage gradually in-

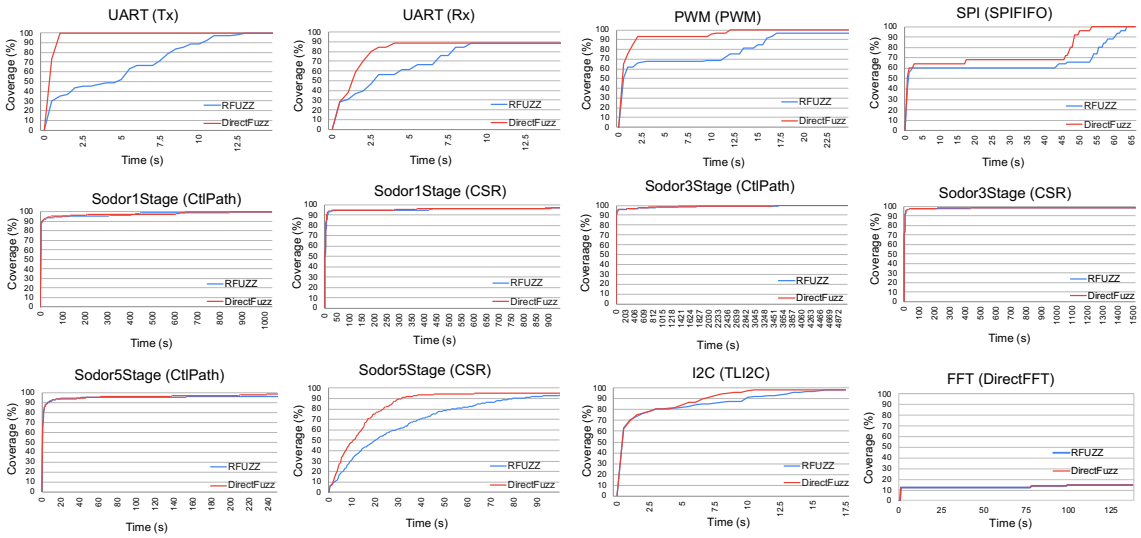


Figure 4·5: Coverage progress of RFUZZ and DirectFuzz over time.

creases and reaches the peak coverage later than DirectFuzz. For some benchmarks (such as Sodor1Stage, Sodor3Stage, and Sodor5Stage for `CtlPath`), we observe that both RFUZZ and DirectFuzz follow a similar pace.

4.5 Summary

This work presents DirectFuzz, a mechanism that generates test inputs for accelerating the testing of specific module instances in an RTL design. As opposed to prior work (RFUZZ) that aims to maximize the coverage of the RTL design, DirectFuzz aims to cover a set of target sites in a specific module instance in the RTL design. A head-to-head comparison with RFUZZ using a variety of benchmarks demonstrates that DirectFuzz achieves the same coverage on specific target sites, on average, 2.23× faster than RFUZZ.

Chapter 5

Guiding Processor Fuzzing via Control and Status Registers

5.1 Introduction

As the complexity of processor designs has continuously grown over the years, verification has become one of the most challenging tasks in processor manufacturing. The state-space of a complex processor is extremely large, while the processor vendors have limited time and resources for verification. An exhaustive verification (i.e., testing each and every scenario) is an unrealistic goal to achieve, and therefore, a high-quality verification methodology is essential to discover bugs before fabrication. A timely, pre-silicon bug discovery can circumvent potentially millions-of-dollars of losses [Intel, 1994]. Otherwise, undiscovered bugs can manifest as severe security vulnerabilities in both proprietary and open-source processors such as transient execution vulnerabilities (e.g., Spectre [Kocher et al., 2019], Foreshadow [Van Bulck et al., 2018]), x86's guest privilege escalation bug [Wojtczuk, 2012], Intel's TSX bug [Jang et al., 2016] that breaks KASLR, Intel's machine check vulnerability [Intel, 2019] that enables denial-of-service attacks, and Pentium's FOOF [Collins, 1998] and FDIV bugs [Edelman, 1997].

Broadly, the verification techniques can be divided into two categories - static and dynamic. Static verification techniques [Chen and Mishra, 2011, Mukherjee et al., 2015, Cadence, 2019] aim to prove that the implementation is accurate with respect

to a specification. Due to the well-known state explosion problem [Dessouky et al., 2019] of these techniques, dynamic verification techniques [Fine and Ziv, 2003, Wagner et al., 2005, Tasiran et al., 2001, Nativ et al., 2001, Gal et al., 2021, Bose et al., 2001] are commonly used as part of the processor verification process. Dynamic verification involves simulating a DUT with a test input and analyzing the behavior of the DUT during or after simulation to identify bugs. Recent works [Hur et al., 2021, Laeufer et al., 2018, Trippel et al., 2021] demonstrate that CGF, a widely-used software testing technique, can be adapted as a dynamic verification technique to identify bugs in a processor design if certain differences between hardware and software are addressed.

Prior works on processor fuzzing mainly focus on addressing **two major challenges.** First, code coverage metrics used for fuzzing software programs (basic block, branch coverage, etc.) are not well-suited for fuzzing hardware [Tasiran et al., 2001, Hur et al., 2021]. Second, a bug in a processor design does not result in an observable anomaly (i.e., crash) during testing as opposed to many software programs which can indicate the presence of bugs by throwing memory violation errors or raising exceptions.

To address the first challenge, researchers have introduced a variety of coverage metrics such as multiplexer toggle coverage, register coverage, etc [Laeufer et al., 2018, Hur et al., 2021, Muduli et al., 2020, Li et al., 2021a] that are tailored for hardware. In the context of a processor, the processor is effectively a complex Finite State Machine (FSM) that consists of a large number of states. Exploring different states in 'processor FSM' is the key to identifying bugs in the processor. Therefore, hardware-specific coverage metrics mainly aim to guide the fuzzer towards different uncovered 'processor FSM' states. These metrics take the hardware intrinsic (e.g., wire connections) into account rather than merely the code structure of the hardware. For instance, DIFUZZRTL [Hur et al., 2021], a state-of-the-art processor fuzzer, intro-

duces *register coverage* metric where the goal is to monitor value changes in registers that control multiplexer selection signals. The intuition is that a particular value in these registers represents a unique state in the 'processor FSM' and guiding the fuzzer based on this feedback explores additional FSM states.

DIFUZZRTL's register coverage metric improves on prior works [Laeufer et al., 2018, Moundanos et al., 1998, Acharya et al., 2015] in terms of scalability, efficiency, and precision. However, we make a key observation that the register coverage can be a highly misleading metric for a processor fuzzer. Specifically, we find that DIFUZZRTL monitors many datapath registers which have minimal control over the current FSM state of the processor. The coverage increase resulting from the datapath registers does not provide meaningful information related to the current FSM state of the processor. This results in inputs that increase datapath register coverage incorrectly being classified as 'interesting' inputs, which in turn leads to wasted fuzzing time.

To address the second challenge, existing processor fuzzers [Hur et al., 2021, Kabylkas et al., 2021, Lee and Cook, 2015, Google, 2021c] adapt differential testing from the software domain to the hardware domain. Differential testing in software compares outputs of multiple programs that have the same functional behavior and checks for inconsistencies. In the hardware domain, the results of a Register Transfer Level (RTL) simulator are compared with those of an Instruction Set Architecture (ISA) simulator. An RTL simulator is used to simulate the execution of an instruction stream on the detailed microarchitecture implementation of the processor. The ISA simulator is used to simulate the functional behavior of the processor design and used as a reference model. A difference in the execution output of RTL simulation and ISA simulation indicates a potential bug in the processor.

In this work, we present ProcessorFuzz, a processor fuzzer that implements two novel features. First, ProcessorFuzz uses a new coverage metric called *CSR-transition*

coverage to effectively guide processor fuzzing towards exploring unique processor states. Specifically, it monitors transitions in Control and Status Registers (CSRs) that form the core of the architecture specifications. Our intuition is that certain CSRs dictated by ISA readily expose the current 'processor FSM' state (e.g., current privilege mode, the event that caused floating mode exception), and thus the transitions in these CSRs signify a new 'processor FSM' state.

ProcessorFuzz's second feature is that it uses ISA simulation to rapidly determine if a test input is interesting. Prior works rely on RTL simulation for the same goal, which is time-consuming. In fact, this problem gets compounded if the coverage guidance is misleading and results in the execution of repetitive test inputs. ISA simulation is significantly faster than RTL simulation[1]. Hence, ProcessorFuzz can efficiently eliminate repetitive test inputs and focus on as many qualitatively distinct test input patterns as possible to expose bugs faster. Another benefit of this design feature is that ProcessorFuzz is agnostic to the hardware description language (HDL) used for designing the processor. Unlike prior works [Laeufer et al., 2018, Hur et al., 2021], ProcessorFuzz does not require any HDL-specific hardware instrumentation because it identifies interesting inputs using ISA simulation. Hence, processors expressed in different HDLs (VHDL, System Verilog, etc.) can easily utilize ProcessorFuzz as a verification tool without having to worry about integration issues.

We evaluate ProcessorFuzz using a variety of widely-used open-source RISC-V based processors Rocket Core [Asanović et al., 2016], BOOM [Celio et al., 2015], and BlackParrot [Petrisko et al., 2020]. Here Rocket Core [Asanović et al., 2016] and BOOM [Celio et al., 2015] have been designed using Chisel HDL, while BlackParrot [Petrisko et al., 2020] has been designed using SystemVerilog. In addition, these processors vary in microarchitectural implementations such as their pipeline depths,

[1]As a reference point, ISA simulation is 79× faster than RTL simulation for open-source RISC-V based processor (i.e., BOOM [Celio et al., 2015]).

execution type (i.e., in-order and out-of-order execution), etc. We compare the bug-finding effectiveness of ProcessorFuzz against the state-of-the-art register coverage guided DIFUZZRTL. On average, for the bugs found by DIFUZZRTL, ProcessorFuzz triggers bugs 1.23× faster than DIFUZZRTL. In addition, ProcessorFuzz revealed 8 new bugs in widely-used open-source processors and one new bug in a reference model.

In summary, we make the following contributions:

- We propose ProcessorFuzz, a new processor fuzzing mechanism. ProcessorFuzz uses a novel *CSR-transition coverage* metric, to effectively guide processor fuzzing towards interesting processor states.
- We propose to use the ISA simulator as part of a coverage feedback mechanism to rapidly identify interesting test inputs, thereby accelerating the bug-finding process.
- We demonstrate the practicality of ProcessorFuzz using 3 different open-sourced RISC-V processors and present eight new bugs identified in those three different processor designs and one new bug in a reference model.

5.2 Background and Motivation

In this section, we provide background about DIFUZZRTL's register coverage and present some examples that clarify why DIFUZZRTL's register coverage can mislead fuzzing. DIFUZZRTL [Hur et al., 2021] adapts CGF approach as a dynamic verification method for hardware to identify bugs in processors. It proposes a feedback strategy that aims to capture FSM state transitions during RTL simulation. The strategy follows a two-stage approach as depicted in Figure 5·1. In stage ①, it performs static analysis to identify a small set of registers in each RTL module and instruments the

RTL with necessary hardware logic to record register coverage at simulation time. At a high level, DIFUZZRTL monitors a register if its value is directly or indirectly used to control a multiplexer selection signal. DIFUZZRTL creates a circuit graph of the RTL design where nodes and edges of this graph represent circuit elements (e.g., multiplexers, wires, ports, registers) and connections, respectively. Then, it recursively performs a backward data-flow analysis for each multiplexer's selection signal and identifies any register in the traversed path. In stage ②, DIFUZZRTL monitors value changes in the identified registers during the RTL simulation. For each clock cycle, DIFUZZRTL hashes all the values in the identified registers into a coverage map to represent the current FSM state. If a new hash value is observed, DIFUZZRTL increases register coverage to signify that the current test is interesting for further mutations.

DIFUZZRTL's register coverage improves prior work [Laeufer et al., 2018,Li et al., 2021a] in terms of scalability, efficiency, and precision. However, using register coverage metric for hardware fuzzing can be highly misleading. At a high level, we observe that a subset of registers leads to misleading coverage increase, and therefore, misguides the hardware fuzzer. We provide more details using two examples (illustrated in Figure 5·1) from the open-source RISC-V-based Rocket Core [Asanović et al., 2016].

Example 1. In the multiplication unit of Rocket Core, there is a 130-bit `remainder` register that indirectly controls 98 mux selection signals and therefore; DIFUZZRTL identifies this register to monitor during fuzzing. [2] The change in the value of `remainder` results in an increase in coverage. Indeed, our further analysis showed that the multiplication of two different numbers (see code snippet in List 5.1) increases the register coverage (i.e., explores a new state) even after 2M iterations.

[2]DIFUZZRTL applies some optimizations to reduce search space. As one of their optimizations, it is able to track only a subset of bits of a register and therefore; ultimately tracks 98-bit in `remainder` register.

Example 2. In the Branch Target Buffer (BTB) unit, there is a buffer with 28 entries that hold branch target addresses for different program counters. Additionally, the BTB contains a set of 2-bit registers (`cfiType*` in Figure 5·1) for each entry to hold control-flow instruction (CFI) type (e.g., `return`, `jump`, and `branch`). The BTB unit first compares the content of each `cfiType` to check for entries that contain `return` as CFI type. Next, it performs a set of AND logical operations to realize if at least one of the BTB entries has `return` as CFI type. The 1-bit output controls a mux selection signal and therefore; 28 different 2-bit `cfiType` registers are identified as registers that will be monitored during fuzzing. This leads to a situation where the CFI type changes in buffer entries result in a coverage increase.

Broadly, as pointed out with the above two examples, DIFUZZRTL monitors and uses coverage information from registers even if they are mostly involved in datapath-related operations and have minimal control over the current FSM state of the hardware. Unfortunately, data-path registers (e.g., remainder and cfiType*) increase search space significantly, yet the coverage increase resulting from data-path registers indeed does not provide meaningful information related to the current hardware state to the fuzzer. Therefore, it is not interesting to keep an input for further mutations if it increases coverage based on certain data-path registers like `remainder` and `cfiType*` registers. In our work, we present a new coverage metric that aims to tackle this problem.

```
1 void main() {
2   unsigned int num1, num2, res;
3   for (int i = 0; i < 2000000; i++){
4     num1 = i; num2 = i+1;
5     res = num1 * num2;
6   }
7 }
```

Listing 5.1: Code snippet for testing multiplication unit.

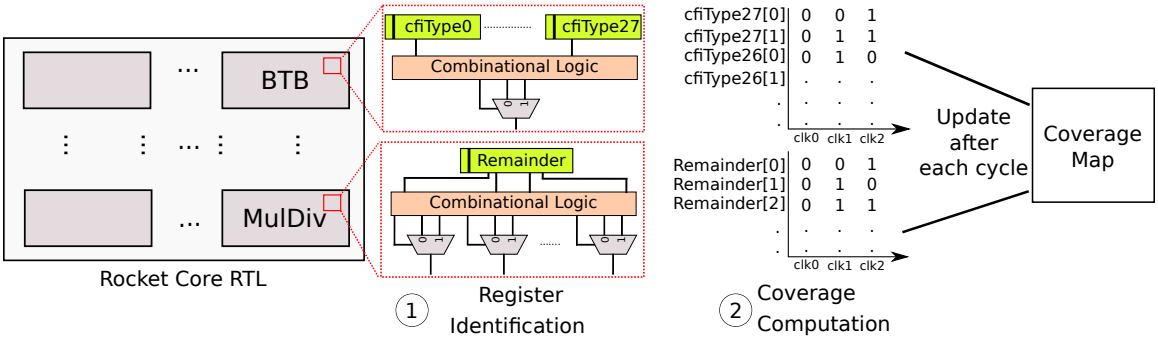


Figure 5·1: Overview of DIFUZZRTL's coverage feedback strategy.

5.3 ProcessorFuzz: Design

In this section, we present the design of ProcessorFuzz, a fuzzing mechanism tailored for processors. We first provide a high-level overview of the different stages of ProcessorFuzz. Then, we outline our reasoning for using ISA simulation instead of RTL simulation to evaluate coverage. Finally, we explain the details of our CSR transition coverage metric and how ProcessorFuzz uses this metric to guide the fuzzing procedure.

We illustrate the design overview of ProcessorFuzz in Figure 5·2. In stage **(1)**, ProcessorFuzz is provided with an empty seed corpus. It populates the seed corpus by generating a set of random test inputs in the form of assembly programs that conforms to the target ISA. Next, ProcessorFuzz chooses a test input from the seed corpus in stage **(2)** and subsequently applies a set of mutations (such as removing instructions, appending instructions, or replacing instructions) on the chosen input in stage **(3)**. For these three stages, ProcessorFuzz uses the same methods applied by a prior work [Hur et al., 2021]. In stage **(4)**, ProcessorFuzz runs an ISA simulator with one of the mutated inputs and generates an extended ISA trace log. A typical trace log generated by the ISA simulator contains (for each executed instruction) a program counter, the disassembled instruction, current privilege mode, and a write-back value as detailed in Section 5.2. The extended ISA trace log additionally includes

the value of CSRs for each executed instruction. The Transition Unit (TU) receives the ISA trace log in stage **(5)**. The TU extracts the transitions that occur in the CSRs. Each observed transition is cross-checked against the Transition Map (TM). The TM is initially empty and populated with unique CSR transitions during the fuzzing session. If the observed transition is not present in the TM, it is classified as a unique transition and added to the TM. In case the current test input triggers at least one new transition, the input is deemed interesting and added to the seed corpus for further mutations. If, however, there are no new transitions triggered, the input is discarded. In stage **(6)**, ProcessorFuzz runs the RTL simulation of the target processor with the mutated input only if the input is determined as interesting. The RTL simulation also generates an extended RTL trace log similar to the extended ISA trace log. The extended RTL trace log contains the same information as the extended trace log. The ISA trace log and the RTL trace log are compared in stage **(7)**. Any mismatch between the logs signifies a potential bug that needs to be confirmed by a verification engineer usually by manual inspection.

5.3.1 Feedback from the ISA Simulation

One design feature of ProcessorFuzz is that it relies on the ISA simulation to determine if a test input is interesting as opposed to prior works that rely on the RTL simulation. Specifically, ProcessorFuzz runs the ISA simulator with each input obtained from the mutation engine and collects necessary feedback (i.e., CSR transitions which we detail in the following subsections) from the simulator. ProcessorFuzz later processes the collected feedback to determine if the input should be ignored or used by the RTL simulator.

We use the ISA simulator to capture the CSR transitions for two main reasons. First, ISA simulators are generally much faster in executing a given program in comparison to executing that program on a processor using the RTL simulation. For

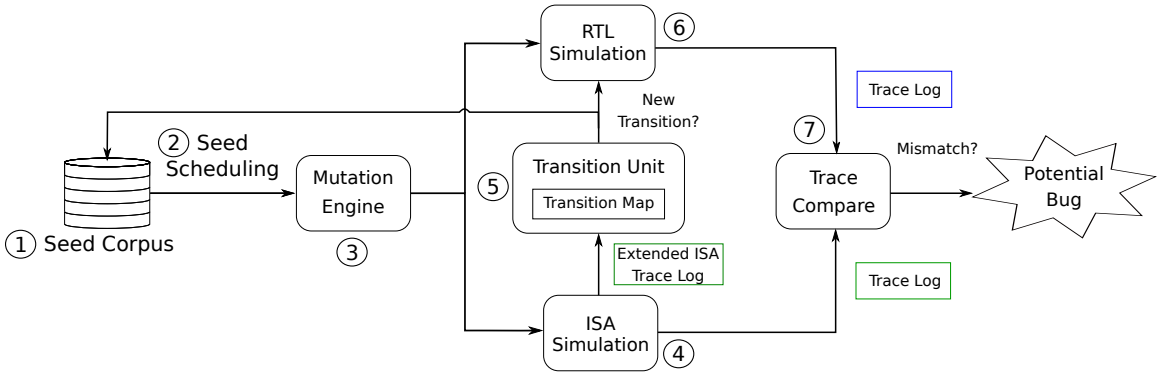


Figure 5·2: ProcessorFuzz Design: ProcessorFuzz runs the ISA simulator with an input generated by the mutation engine and outputs an extended ISA trace log that contains CSR values. The transition unit extracts CSR transitions, determines if a transition is new by checking the transition map, and stores new ones in the transition map. ProcessorFuzz runs the RTL simulation only with interesting inputs and creates an RTL trace log to be compared with the ISA log for bug detection.

instance, we observed that the RISC-V Spike ISA simulator is, on average 79× faster than the RTL simulation of the RISC-V BOOM processor. This speedup provides a considerable advantage as ProcessorFuzz can then quickly identify if a test input is interesting without performing the slow RTL simulation. Eliminating inputs with similar characteristics help ProcessorFuzz to achieve faster bug discovery times as shown in Section 5.4.

Second is the reduced effort needed to instrument the simulator. A simulator needs to be instrumented to generate an extended trace log with the selected CSRs. An ISA simulator can be easily instrumented by extending the already available trace logic with the selected CSRs. The same instrumented ISA simulator can be used to fuzz any processor design as long as it has been designed for the same ISA target. In contrast, instrumenting RTL designs for tracking the coverage metrics requires extensive effort. Moreover, instrumentation in one HDL does not readily translate to other HDLs.

#	PC	Instruction	[Privileged	Unprivileged]
1	0x045c	sret	[8000000a00006000,00,0f,b100,	0,00]
2	0x283c	sraiw s5, s0, 6	[8000000a00006020,00,0f,b100,	0,00]
3	0x2840	fdiv.s fs11, ft0, fa7	[8000000a00006020,00,0f,b100,	0,00]
4	0x2844	fence iorw,iorw	[8000000a00006020,00,0f,b100,	0,03]
5	0x2848	fsqrt.s ft0, ft5	[8000000a00006020,00,0f,b100,	0,03]
6	0x284c	fcvt.wu.s s6, fs5	[8000000a00006020,00,0f,b100,	0,03]
7	0x2850	addi a0, a0, 1344	[8000000a00006020,00,0f,b100,	0,13]
8	0x2854	fsgnj.s ft4, ft3, fa3	[8000000a00006020,00,0f,b100,	0,13]

Figure 5·3: Extended trace log generated by the ISA simulator. The values (in hexadecimal) of a subset of CSRs in Table 5.1 are included within the square brackets in the given order; `mstatus`, `mcause`, `scause`, `medeleg`, `frm`, and `fflags`. Transitions are color coded; red and blue for `mstatus` and `fflags` CSR transitions, respectively.

5.3.2 CSR-transition Coverage

Description of the Metric. As described in Section 5.2, DIFUZZRTL's register coverage technique monitors many datapath registers (e.g., `remainder` and `cfiType` registers) to determine the current FSM state, which leads to large state space. Hence, guiding the fuzzing procedure with DIFUZZRTL's register coverage metric can be highly misleading when fuzzing processors. To test the processor with as many qualitatively distinct input patterns as possible, we propose a novel CSR transition-based coverage metric.

CSRs are system registers in an ISA specification. These registers are used to control (e.g., delegated exceptions) or hold information (e.g., state of the floating-point unit) about the current architectural state of the processor. Our intuition for using CSRs is as follows. A processor is a complex FSM where CSRs have direct control over the current processor state. A value change in a CSR often signifies an architectural state change such as a value change in a CSR that stores exception code or privilege level. Therefore, ProcessorFuzz aims to realize the current state of the processor by monitoring transitions in CSRs to guide the fuzzer towards interesting processor states.

CSRs are part of both an ISA simulator and the RTL design of a processor. Hence, CSR transitions can be extracted either from the ISA simulation or the RTL

simulation. As detailed before, ProcessorFuzz uses the ISA simulator to capture CSR transitions. Specifically, to extract a CSR transition, ProcessorFuzz monitors the CSR values resulting from the execution of the previous and current instructions and checks if they differ. If so, ProcessorFuzz uses the transition to determine if the input is interesting as detailed in the following subsections. Here, we provide a concrete example to illustrate how ProcessorFuzz identifies a CSR transition in the ISA trace log. Consider the extended ISA trace log shown in Figure 5·3. The CSR value changes after execution of the *sret* instruction shown in Line 1, which can be seen by comparing the entries in Line 1 and Line 2 of the 'Privileged' column. Specifically, we observe a CSR-transition in `mstatus` CSR from $0x8000000a00006000$ to $0x8000000a00006020$ as highlighted in red in Figure 5·3. Overall, from Figure 5·3, we represent the CSR transition caused by *sret* instruction as (S_0, S_1) = (8000000a00006000000fb100000, 8000000a00006020000fb100000), where S_0 and S_1 are defined as the concatenated CSR values before and after the transition, respectively..

Why Transitions Instead of Values? DIFUZZRTL determines the current processor state based on the register coverage as detailed in Section 5.2. For each newly covered FSM state, DIFUZZRTL's register coverage only stores the current state of the processor and does not consider the previous state. Unfortunately, this design choice can lead to important test inputs being discarded by the fuzzer and the fuzzer can potentially miss out on the discovery of a bug. We illustrate this in detail below. Figure 5·4 represents a subset of the abstract states associated with a real-world bug (Bug 2 in Table 5.3) that we identified in an open-source RISC-V processor.

In the figure, the processor starts out in the N0 state. The bug triggers in the N2 state only if the previous state is N1. It does not trigger when the previous state is N0. During a coverage-guided fuzzing session, if both N1 (through P0 transition) and N2 (through P2 transition) are covered individually, there will not be a coverage increase

for the denoted P1 state transition. And so, the unique P1 transition is not particularly driven towards. Thus, the fuzzing session fails to trigger the bug. Contrarily, by monitoring transitions, we can detect P1 as a new transition even though N1 and N2 states are already covered. Overall, we monitor new transitions in CSRs rather than just identifying unique CSRs values to improve the sensitivity of the feedback metric. Indeed, our rationale is similar to widely-used software fuzzers [Google, 2017, LLVM, 2021]'s rationale that monitors edges in a program instead of basic blocks. We provide the details on how ProcessorFuzz extracts CSR transitions in the next subsection.

CSR Selection Criteria. An ISA specification usually specifies a large number of CSRs[3]. Monitoring all available CSRs for transitions can mislead the fuzzer because not all CSRs are required for representing the architectural state. Therefore, we introduce the following two criteria to select the CSRs that ProcessorFuzz monitors transitions. First, we select any CSRs that contain status information about the processor to learn about the current architectural state. As an example, we select a CSR that stores the cause for an exception taken by the processor (e.g., `mstatus`). Second, we select any CSR that is used to set a certain configuration in the processor. Here, we aim to realize if the processor behaves as expected under different configurations. For instance, we select the CSR that is used to configure which traps can be delegated to lower privilege levels (i.e., `medeleg`) because it directly determines the architectural state of the processor in case an exception occurs. In Table 5.1, we list all the CSRs in the RISC-V ISA that we used for identifying transitions in the current implementation of ProcessorFuzz based on the aforementioned two criteria.

[3]As a reference point, RISC-V ISA defines up to 4096 CSRs

Table 5.1: CSR selection for RISC-V ISA implementation of ProcessorFuzz

CSR Group	CSR	Description
Privileged	mstatus.xIE	Controls the global interrupt enable bit for privilege x, x = {M, S, U}
	mstatus.xPIE	Holds the value of interrupt-enable bit active prior to the trap for privilege mode x
	mstatus.xPP	Holds the previous privilege mode active prior to a trap taken to privilege mode x
	mstatus.XS	Contains the state of any additional user-mode extensions
	mstatus.FS	Contains the state of the floating-point unit
	mstatus.MPRV	Controls the privilege mode in which the memory operations are performed
	mstatus.SUM	Controls the permission for accessing user memory from supervisor mode
	mstatus.MXR	Controls the privilege with which loads access virtual memory
	mstatus.TVM	Controls the ability to edit virtual-memory configuration from supervisor mode
	mstatus.TW	Controls the privilege modes that wait for interrupt (WFI) is allowed to execute
	mstatus.TSR	Provides the ability to trigger a trap when SRET instruction is executed in supervisor mode
	mstatus.xXL	Controls the width of an integer register for privilege mode x, x = {S, U}
	mstatus.SD	Indicate the combined state of mstatus.FS and mstatus.XS for context switches
	mcause	Contains the trap cause when a trap is taken in to machine mode
	scause	Contains the trap cause when a trap is taken in to supervisor mode
	medeleg	Decides what type of exceptions are delegated to supervisor mode from machine mode
	mcounteren	Controls the availability of the hardware performance-monitoring counters for supervisor mode
	scounteren	Controls the availability of the hardware performance-monitoring counters for user mode
Unprivileged	frm	Controls the dynamic rounding mode for floating-point operations
	fflags	Holds the accrued exceptions from the floating-point operations

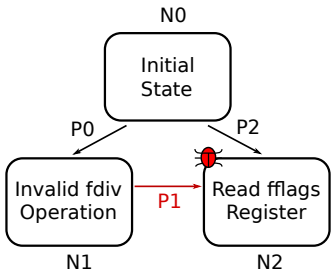


Figure 5·4: Abstract state diagram for triggering Bug 2 in Table 5.3

Apart from these two criteria, CSR selection can be further limited depending on the features supported by the target processor or the desired scope of verification. For example, if the target processor does not support interrupts within the testing framework, any CSRs related to the configuration or status of interrupts can be excluded. CSR selection can be limited if the scope of required verification is narrowed down to a unit in the target processor. As an example, if we only want to verify the functionality of the floating-point unit in the processor, only floating-point CSRs can be monitored to identify transitions.

5.3.3 Transition Unit

As shown in Figure 5·2, the TU takes an extended ISA trace log as input and communicates with the TM to output whether the trace log contains any new transitions. We describe the complete workflow of the TU in Figure 5·5. As a first step, the TU extracts all CSR transitions in the trace log based on the description in Section 5.3.2. Then, ProcessorFuzz applies a filter to remove unnecessary transitions. Next, the TU groups the transitions to reduce the state space. We describe how the TU filters and groups the transitions in the rest of this subsection.

Filtering Transitions. We note that the number of possible CSR transitions can be large depending on the cumulative width of the selected CSRs. However, not all CSR transitions represent interesting architectural state changes that are relevant for testing processors. For instance, a test program running on the target processor

can write to a CSR that contains processor status, e.g. `mstatus` CSR in RISC-V ISA. This could get identified as a new CSR transition. If the write operation is legal, the processor continues the execution of the program and eventually overwrites the CSR with the updated status. Overall, the type of transitions that occur from writes to status CSRs do not affect the architectural state of the processor. Thus, ProcessorFuzz filters out transitions that occur from explicit writes to status CSRs.

Grouping Transitions. ProcessorFuzz provides the flexibility to customize the CSR-transition coverage metric to be suitable for verifying different Architectural Units (AUs) individually. Specifically, ProcessorFuzz allows a designer to group CSR transitions of AUs, thereby considering them as independent events. Grouping transitions improve the exploration of CSR transitions within each group. As a result, the fuzzer is able to generate tests targeted towards individual AUs and verify them thoroughly. This is a useful feature for a verification engineer as AUs in a processor can be individually verified as an initial step of verification. For example, privileged and unprivileged architectures in a RISC-V processor can be verified individually by grouping transitions as shown in Figure 5·3. Identifying and fixing the bugs in each AU before fuzzing the processor as a whole can reduce the overall verification effort.

Transition Map ProcessorFuzz maintains a transition map to store CSR-transitions. Each transition is stored in the map as a tuple: (I_m, S_0, S_1) where I_m is the mnemonic of the instruction whose execution resulted in the CSR transition. S_0 and S_1 are CSR values before and after the transition as defined in subsection 5.3.2. Revisiting the same example given in subsection 5.3.2, privileged CSR-transition caused by *sret* instruction can be represented

as (*sret*, 8000000a00006000000fb1000000, 8000000a00006020000fb1000000). Likewise, ProcessorFuzz converts the unprivileged CSR-transition in lines 3 and 4 in Figure 5·3 to (fdiv.s, 0000, 0003).

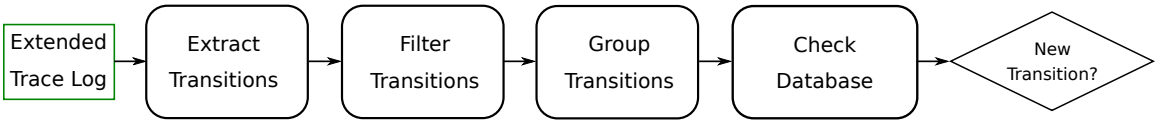


Figure 5·5: Workflow of the ProcessorFuzz transition unit.

We include instruction mnemonic in the aforementioned tuple because the same transition can be triggered by different instructions. For example, both floating-point division and floating-point square-root instructions can trigger the same transition in fflags CSR in RISC-V ISA due to invalid operations. Nevertheless, only the invalid operation of floating-point division instruction might contain a bug. Therefore, we tag each transition with the mnemonic of the instruction that triggered it to uniquely identify transitions triggered by different instructions. Only the mnemonic of the instructions is included to ignore repetitive transitions that get triggered by different operands of the same instruction.

Once tuples are created, the map is queried to check whether the detected transition is new or a duplicate. Tuples that are identified to contain new transitions are added to the map while marking the current test input as interesting. The transition map is empty at the beginning of a fuzzing session and maintained throughout the session.

5.3.4 RTL Simulation and Trace Comparison

If the TU determines that the current input results in a unique CSR transition, ProcessorFuzz launches the RTL simulation and generates the RTL trace log. ProcessorFuzz then compares the RTL trace log with the ISA trace log. Any difference between these logs signifies a potential bug in the processor design and needs to be investigated further by a verification engineer. In case the input does not result in a unique transition, ProcessorFuzz discards the input and proceeds to the next fuzzing iteration.

5.4 Evaluation

In this section, we evaluate the effectiveness of ProcessorFuzz using real-world processor designs. First, we provide the details of our evaluation setup. Then, we assess the bug-finding capability of ProcessorFuzz using ground-truth bugs and compare ProcessorFuzz's performance against DIFUZZRTL. Specifically, we analyze if ProcessorFuzz can expose the same set of bugs reported by DIFUZZRTL in a more efficient way. Finally, we describe the list of new real-world bugs that ProcessorFuzz identified along with the severity of the bugs.

5.4.1 Evaluation Setup

Implementation Details. ProcessorFuzz has two main implementation steps; generation of an extended trace log using the ISA simulator and building the TU (see Figure 5·2). For the former, we extended SPIKE [Spike, 2019] open-source ISA simulator to store the values of monitored CSRs (see Table 5.1) for each executed instruction during the ISA simulation. The TU is implemented as a Python library. For the RTL simulation of all processors designs, we used Verilator [Verilator, 2003], an open-source RTL simulator. We used the same mutation engine (see Figure 5·2) as provided by DIFUZZRTL's open-source repository. Using the same engine is important since our goal is to compare two coverage feedback mechanisms (i.e., register coverage and CSR-transition coverage) rather than input generation mechanisms. We separated transitions belonging to `frm` and `fflags` to separate floating-point operations from the rest of the CSRs.

Processor Designs. In our evaluation, we use three real-world open-source processors designed using the open-standard RISC-V ISA. *RISC-V Rocket Core* is an open-source, general-purpose, in-order, RISC-V processor core that can be generated using the Rocket Chip SoC Generator framework [Asanović et al., 2016]. Rocket

core is designed in Chisel HDL [Bachrach et al., 2012], and is shown to integrate well with custom hardware accelerators. Rocket core has been taped out multiple times [Asanović et al., 2016] and is capable of booting Linux. Essentially, it is well-tested. We used Spike [Spike, 2019] as a reference model to verify the correctness during fuzzing. The commit version of the Rocket core that we used is `148d5d2`. *RISC-V BOOM Core* [Celio et al., 2015] core can also be generated from the same Rocket Chip SoC Generator framework [Asanović et al., 2016] and is also designed in Chisel HDL. BOOM is an out-of-order, superscalar RISC-V processor core and capable of booting Linux. BOOM has also been taped out [Celio, 2017]. We used the Spike ISA simulator to verify the correctness during fuzzing. The commit version of the BOOM core that we used is `148d5d2`. *RISC-V BlackParrot Core* [Petrisko et al., 2020] is an open-source 64-bit RISC-V core, designed in the industry-standard SystemVerilog HDL. BlackParrot is an ideal candidate for hosting accelerator fabrics and for hardware research owing to its tiny, modular, and friendly design approach. BlackParrot is silicon-validated and is in active development. We used Dromajo [Dromajo, 2019] as a reference model to expose the bugs in BlackParrot for the `bc3b48b` commit version.

Settings. We compared ProcessorFuzz with two different settings of DIFUZZRTL. The first setting is `no-cov-difuzzrtl` where DIFUZZRTL fuzzing framework is used without any coverage guidance (i.e., as a blackbox fuzzer). For all the cores that we evaluated, we successfully used this setting as a comparison point. The second setting is `reg-cov-difuzzrtl` where DIFUZZRTL fuzzing framework relies on register coverage as a guidance mechanism. While this setting is applicable to Rocket and BOOM Cores, it is not the case for BlackParrot Core. This is because DIFUZZRTL's register coverage passes do not support SystemVerilog. They are tailored for FIRRTL [Izraelevitz et al., 2017], an intermediate representation (IR) used by

Chisel HDL, which is used to design Rocket and BOOM cores. We tried to convert SystemVerilog to FIRRTL using an open-source tool (i.e., Yosys [Wolf, 2015]), and apply DIFUZZRTL's register coverage passes. However, we observed several issues during this conversion due to the limited support for SystemVerilog to FIRTTL conversion and thus failed to instrument BlackParrot. In our experiments, we used the DIFUZZRTL as the sole comparison point since it shows clear benefits over previous processor fuzzing frameworks. Also, for each setting, we reported Time-to-Exposures (TTE) which is defined as the total elapsed time from the starting of the fuzzing session until the bug is exposed.

Infrastructure. All the experiments based on ISA and the RTL simulations were conducted on server nodes with Intel®Xeon®E5-2670 CPUs and CentOS Linux 7 as the operating system. We fuzzed each processor design 10 times for each setting and allocated 48 hours (2 days) of time limit for each fuzzing instance. For each fuzzing instance, we dedicated two cores and 8GB of memory. In total, it took 2400 CPU hours to conduct all the experiments detailed in the following sections.

5.4.2 Ground-truth Bugs

As discussed by many prior works [Manès et al., 2019, Klees et al., 2018], the bug-finding capability of a fuzzer is the ultimate litmus test for a fuzzer. While there exist several fuzzing benchmarks for software programs [Li et al., 2021b, Hazimeh et al., 2020b], this is not the case for processors. Therefore, we relied on a set of bugs (in total six bugs) previously reported by DIFUZZRTL for BOOM processor to evaluate the bug-finding capability of ProcessorFuzz and perform a head-to-head comparison with DIFUZZRTL. Overall, our evaluation aims to demonstrate that ProcessorFuzz can guide the fuzzer efficiently to discover ground-truth bugs thanks to the CSR-transition feedback obtained using the ISA simulation.

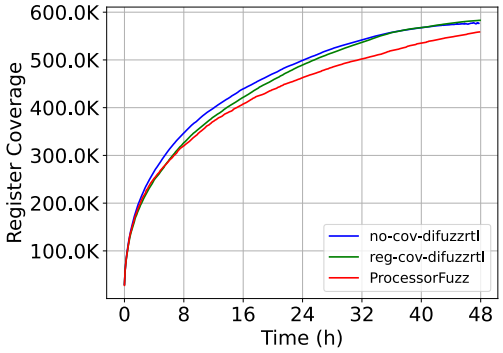
In Table 5.2, we report the TTE of bugs in seconds for three different settings; Pro-

Table 5.2: The speedup achieved by ProcessorFuzz over no-cov-difuzzrtl, and reg-cov-difuzzrtl for the ground-truth bugs in the BOOM processor. We also report speedup of reg-cov-difuzzrtl over no-cov-difuzzrtl. In the table, we state the maximum allowed runtime of 48 hours (172800 seconds) for bugs that could not be found.

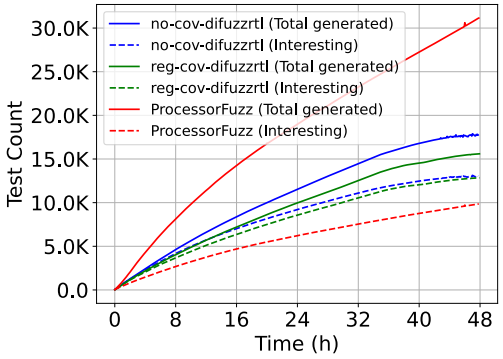
	no-cov-difuzzrtl	**reg-cov-difuzzrtl**		**ProcessorFuzz**		
Issue No	Time (s)	Time (s)	Speedup	Time (s)	Speedup (no-cov)	Speedup (cov)
#458	104.3	70.3	1.48	54.0	1.93	1.3
#454	32883.3	45322	0.73	25020	1.31	1.81
#492	2047.2	4238	0.48	1821.2	1.12	2.32
#493	585.4	495	1.18	278.7	2.1	1.77
#503	1463.7	1011	1.44	2795.9	0.52	0.36
#504	172800	178000	NA	1728000	NA	NA
Geo.	3182.9	3225.9	0.98	2630.7	1.21	1.23

cessorFuzz, `no-cov-difuzzrtl`, and `reg-cov-difuzzrtl`, for the BOOM processor core. We also show the achieved speedups by ProcessorFuzz over `no-cov-difuzzrtl`, and `reg-cov-difuzzrtl`. All three mechanisms discovered five out of six bugs reported in the DIFUZZRTL within the fuzzing time limit in our experiments. Unfortunately, ProcessorFuzz could not detect #504 with any of the settings. In summary, ProcessorFuzz achieved, on average, 1.21× (up to 2.1×) and 1.23× (up to 2.32×) speedups over `no-cov-difuzzrtl` and `reg-cov-difuzzrtl`, respectively.
`no-cov-difuzzrtl` performed slightly better than `reg-cov-difuzzrtl`.

To understand the performance of ProcessorFuzz and DIFUZZRTL for different bugs, we further study the relationship among register coverage, CSR-transition coverage, and bug-finding times. Specifically, in Figure 5·6a, we show the measured register coverage progress for different settings of DIFUZZRTL and ProcessorFuzz. Although ProcessorFuzz covers less number of states (i.e., achieves lower register coverage) during fuzzing, it was still able to discover bugs faster. For instance, ProcessorFuzz triggered the most challenging bug (i.e., #454) after exploring 303K states



(a) Register coverage progress during fuzzing.



(b) Coverage increasing and total test input counts during fuzzing.

Figure 5·6: Coverage details for different settings.

while `no-cov-difuzzrtl` and `reg-cov-difuzzrtl` could trigger that bug even after exploring 364K and 354K states, respectively. This particular bug shows that higher register state coverage does not necessarily translate to a faster bug discovery. Indeed, an increase in coverage due to value changes in datapath registers can mislead the fuzzer since inputs with similar characteristics (see the multiplication example in Section 5.2) are repeatedly used by the fuzzer to generate a new set of inputs.

In Figure 5·6b, we also show the total number of test inputs that lead to a coverage increase, i.e. 'interesting test inputs', and the total number of inputs generated

by the mutation engine for the two settings of DIFUZZRTL and ProcessorFuzz. For `no-cov-difuzzrtl` and `reg-cov-difuzzrtl`, we use the register coverage metric, same as that used in the DIFUZZRTL work, to realize if a test input increases coverage. For ProcessorFuzz, we use the CSR-transition coverage metric to detect inputs that resulted in a coverage increase. The results provide an important takeaway. Although ProcessorFuzz generates significantly more inputs than other approaches, it is very selective when categorizing a test input as an 'interesting' input. Consequently, ProcessorFuzz identified only 33% of the generated test inputs as interesting (i.e., caused a unique CSR transition). Moreover, ProcessorFuzz could expose the bugs faster although it used the least number of test inputs for RTL simulation. Note that ProcessorFuzz launched the RTL simulation only with interesting inputs (i.e., curved dotted red line) and discarded any other generated input. Using the fast ISA simulation enabled ProcessorFuzz to quickly eliminate inputs that do not result in a new FSM state and spend more time on inputs that explore new FSM states.

5.4.3 Newly Discovered Bugs

In Table 5.3, we document the various **new** bugs discovered by ProcessorFuzz in the selected processors mentioned earlier and in the ISA simulator used as a reference model. In the rest of this subsection, we describe and highlight the significance of each bug. Additionally, we provide TTE comparison between ProcessorFuzz and `no-cov-difuzzrtl`.

Bug 1. When multiple floating-point precisions are supported by a floating-point unit in a processor, valid lower precision values are expected to be NaN-boxed (i.e., remaining upper bits set to 1's). Otherwise, lower precision values are expected to be interpreted as NaNs. BlackParrot does not interpret non-boxed floats as NaNs, which leads to functionally incorrect computations on non-boxed floats, thus violating the ISA specification. Incorrect computations in security-critical functions (e.g.,

Table 5.3: Brief description of bugs discovered by ProcessorFuzz, and their current status, in various processor cores.

Bug	Core / Simulator	Brief description of the bug	Status
1	BlackParrot	Non-boxed single-precision floating point values are not interpreted as NaNs	Confirmed; not fixed
2	BlackParrot	Read-after-Write dependencies on `fcsr.fflags` are not satisfied.	Fixed
3	BlackParrot	When `mstatus.FS` is not set and the `fcsr` is written, `FS` is unexpectedly updated.	Fixed
4	BlackParrot	The 2 low-bits of `sepc` CSR are not write-insensitive.	Fixed
5	BlackParrot	No exception raised when writing certain read-only CSRs.	Fixed
6	BlackParrot	Reading `zero` register, following specific instruction sequences, return unexpected non-zero values	Fixed
7	BlackParrot	Unexpected store access-fault on properly aligned, unpaired *sc.d* instruction.	Reported
8	Dromajo	PMP checks are performed, and raise exceptions upon encountering violations, even with no PMP entries set.	Confirmed; not fixed
9	Rocket & BOOM	Instruction page fault not raised when accessing non-leaf PTEs with certain unspecified page attributes.	Fixed
10	BOOM	`mstatus.FS` is gratuitously set to dirty.	Confirmed; not fixed

in cryptographic applications) can compromise the security of a processor (CWE-1201 [MITRE, 2019]).

Bug 2. Certain floating-point instructions update `fflags` CSR which holds events like floating-point overflow, division by zero, etc. A read-after-write (RAW) hazard occurs in a pipelined processor when a floating-point instruction that writes to `fflags` is followed by an instruction that reads `fflags`.ProcessorFuzz detected that this particular hazard is not handled by the BlackParrot processor, causing the software to read an outdated `fflags` value.The RISC-V ISA requires explicit checks of the `fflags` CSR in software to identify floating-point overflows, invalid operations, etc. Therefore, failure to detect an overflow can lead to security issues in software such as buffer overflows (e.g., CVE-2020-10029 [NVD, 2020]). This bug falls under core and compute hardware weakness (CWE-1201 [MITRE, 2019]). ProcessorFuzz

was able to detect this bug since it monitors the transitions in `fflags`.

Bug 3. When the `FS` field of `mstatus` CSR is set to 0, it indicates that floating-point extension is turned off. In such cases, accessing `fcsr` is expected to raise an illegal instruction exception without altering the `FS` field of `mstatus` CSR. However, BlackParrot wrongly sets the `FS` field to *dirty*, instead of keeping it unchanged.

Bug 4. The least significant two bits of `sepc` CSR must be always hardwired to 0 on implementations that only support 32-bit instruction alignment. Any write from software to the least significant two bits of the `sepc` CSR must be discarded. However, BlackParrot updated the low bits when a test input attempted to modify them. Further analysis showed that this issue exists for `mepc` CSR as well.

Bug 5. Any attempt to modify a read-only register is supposed to trap with an illegal instruction exception. ProcessorFuzz discovered that BlackParrot does not raise an illegal instruction exception if a test input updates a read-only register, specifically `mhartid`. ProcessorFuzz monitors `mcause` and `scause` CSRs that are in charge of exception handling and was, therefore, able to expose this bug.

Bug 6. Any write attempt to the zero register (i.e., `x0`) must be ignored according to the RISC-V ISA. However, in the BlackParrot processor, we detected that the `x0` register is read as a non-zero value one of the preceding division instructions that writes to `x0` is still in the pipeline. Further analysis revealed that this discrepancy is due to bypassing the result of division operation to the following instruction even when the destination register of a division operation is `x0`. ProcessorFuzz was able to identify this bug because a test input that has this scenario resulted in a CSR transition in `fflags` due to division by zero.

Bug 7. A store-conditional instruction, if properly aligned to the appropriate word boundary, should not raise a store-access fault. However, BlackParrot raises a store access fault when executing an unpaired, but properly aligned *sc.d* instruction. We

reported the issue to the BlackParrot designers and are currently waiting for their response.

Bug 8. According to RISC-V privileged specification, the effective privilege mode for implicit page table accesses should be supervisor mode. However, we observed that Dromajo accesses page tables in user mode privilege level when executing user-mode programs. Further analysis revealed that Dromajo also carries out Physical Memory Protection (PMP) checks in user mode when no PMP entries are set, violating the RISC-V ISA privileged specification in two counts.

Bug 9. In a multi-level page table implementation, the accessed (A), dirty (D), and user-mode (U) bits of a non-leaf page table entry (PTE) are reserved for future use and should be cleared. If these bits are set in a non-leaf PTE, the processor must raise an instruction page fault when accessing the PTE according to RISC-V ISA. We discovered that Rocket and BOOM cores do not raise instruction page fault when software attempts to access a PTE with any of A, D, or U bits set. This bug is similar to CWE-1209 [MITRE, 2019] where failure to disable reserved bits allows attackers to compromise the hardware state.

Bug 10. The `FS` field in the `mstatus` CSR in RISC-V ISA is used to check whether save and restore of floating-point registers are required when there is a context switch. ProcessorFuzz detected that BOOM set the `FS` field to dirty for any write to fcsr register, even when the value of fcsr is zero and unchanged by the write operation. This scenario is not a violation of RISC-V ISA due to the flexibility allowed by the ISA for maintaining `FS` field. Nevertheless, setting `FS` field when the floating-point unit state is unchanged degrades the performance as the processor unnecessarily saves and restores floating-point registers.

In Table 5.4, we provide the TTEs for six newly identified and confirmed bugs (Bug 1-6) and one newly identified but currently waiting confirmation bug (Bug 7)

Table 5.4: The speedup achieved by ProcessorFuzz on newly discovered bugs in the BlackParrot processor. A maximum of 48 hours (172000 seconds) is assigned if a bug is not triggered.

Bug	no-cov-difuzzrtl	ProcessorFuzz	Speedup
Bug 1	464.9	230.2	2.0
Bug 2	95695.0	57441.3	1.7
Bug 3	1520.1	1474.5	1.0
Bug 4	585.3	308..0	1.9
Bug 5	476.1	242.1	2.0
Bug 6	172800.0	147942.3	1.2
Bug 7	5192.6	3264.6	1.6
Geo.	6217.3	4199.83	1.6

in BlackParrot core. We did not include Bug 8-10 since they were easily detected in all the settings that we used in our evaluation. For this evaluation, we were only able to compare ProcessorFuzz with `no-cov-difuzzrtl`. As detailed in Section 5.4.1, we could not instrument BlackParrot with register coverage since DIFUZZRTL lacks support for SystemVerilog. ProcessorFuzz does not require any instrumentation on the RTL design, therefore, could successfully guide the fuzzer with CSR-transition coverage to expose bugs. Overall, ProcessorFuzz achieved 1.6× speedup, on average, over `no-cov-difuzzrtl`. Note that only ProcessorFuzz was able to detect Bug 6 from Table 5.3.

5.5 Summary

This work introduces ProcessorFuzz, a processor fuzzer guided by a novel CSR-transition coverage feedback obtained from ISA simulation. ProcessorFuzz demonstrates that monitoring CSR transitions can effectively guide fuzzing towards buggy processor states. Moreover, using ISA simulation instead of RTL simulation can quickly eliminate inputs that result in the same coverage, thereby helping the fuzzer to test as many qualitatively different inputs as possible. Our experimental results

discovered eight new bugs in established, real-world, RISC-V processors, and one new bug in a reference model.

Chapter 6

Conclusion and Future Work

6.1 Summary of the thesis

The bugs at different layers of computing systems have resulted in severe consequences over the years. Extensive bug-finding strategies can eliminate many bugs in different layers of a computing system and significantly reduce the attack surface for adversaries. In this thesis, we present three different fuzzing mechanisms to expose bugs in hardware and software layers. In this chapter, we summarize the contributions of each completed research project and discuss our potential future works.

6.2 Guiding Software Fuzzing with a Target-specific Seed Corpus

Existing DGF tools use seed corpora mainly designed for CGF. Providing coverage-based corpora with DGF unnecessarily wastes the fuzzing effort on the unrelated program regions, thereby hindering the bug-finding capability. To mitigate this limitation, we present TargetFuzz, a mechanism that outputs a seed corpus tailored for DGF. By using the seed distances, TargetFuzz provides DART corpus only consisting of the "close" seeds to the modified code region. When a DGF tool is equipped with DART corpus, the execution of the DGF tool is steered towards stressing modified code regions, thereby increasing the chance of detecting bugs. Evaluations on 34 real bugs show that AFLGo (a state-of-the-art directed greybox fuzzer), when equipped with DART corpus, finds 10 additional bugs and achieves 4.03×speedup, on average,

in the time-to-exposure compared to a generic CGF-based corpus.

As the software evolves with new features, the coverage of baseline seeds can potentially change. When the baseline seeds do not cover the project well (i.e., the code coverage decreases), it is necessary to update the baseline seed corpus. The update frequency of the baseline seed corpus is highly project-dependent since different projects have different characteristics. For instance, in recent years, the libpng developers submitted a very limited number of commits where each commit modifies only a few lines of code. However, PHP developers are more active (i.e., higher commit frequency where some of the commits modify a large number of lines). Mechanisms that determine the update frequency of baseline seed corpus are useful, yet out of scope of TargetFuzz work.

Seed trimming approaches [Zalewski, 2020, Pailoor et al., 2018] attempt to transform a seed into another version that is smaller in size yet achieves the same code coverage as the original seed. Seed trimming is useful since smaller seeds consume less I/O (thus having higher throughput). As part of future work, we could leverage seed trimming approaches to reduce seed sizes in the baseline seed corpus and DART corpus and analyze their impact on TargetFuzz in terms of the bug-finding success.

TargetFuzz currently focuses on C/C++ applications. Another potential future direction is to analyze how the different nature of high-level languages change the effectiveness of TargetFuzz. We can explore different seed selection strategies tailored for high-level programming languages (e.g., Python) to discover logic bugs instead of memory violations.

6.3 Guiding Test Generation for Hardware Designs using DGF

Prior hardware fuzzing works aim to maximize the overall coverage of the RTL design. Unfortunately, this goal is not suitable for every single test scenario. Hardware designs are gradually implemented rather than in one single step. Therefore, a thorough verification is not necessary for every single change in the RTL design. To mitigate this shortcoming, we present DirectFuzz, a mechanism that generates test inputs for accelerating the testing of specific module instances in an RTL design. DirectFuzz adapts the notion of directed greybox fuzzing from the software domain to achieve the targeted test generation goal. As opposed to prior work (RFUZZ) that aims to maximize coverage globally, DirectFuzz aims to cover a set of target sites in a specific module instance in the RTL design. This way, DirectFuzz dedicates more fuzzing time to the RTL components that need to undergo thorough testing. Our experimental comparison with RFUZZ using a wide range of benchmarks shows that DirectFuzz achieves the same coverage on specific target sites, on average, 2.23× faster than RFUZZ.

In our current design, DirectFuzz directs fuzzing to a specific module instance and determines target sites (i.e., multiplexer selection signals that need to be covered) by analyzing the module instance. It is possible that only a part of the module instance (e.g., a couple of lines of code in the module instance) could be of interest for fuzzing due to a recent change in the HDL code. For this scenario, we could leverage meta-information provided by FIRRTL to associate the changed lines in the HDL code with multiplexer selection signals. This way, while assigning energy and prioritizing seeds, DirectFuzz could only take into account the associated multiplexer selection signals and be more targeted towards changed HDL lines.

6.4 Guiding Processor Fuzzing via Control and Status Registers

Existing processor fuzzers can impose significant instrumentation overhead and cause misleading coverage guidance. Besides, they do not support widely-used HDLs. To alleviate the shortcomings of prior works, we present ProcessorFuzz, a processor fuzzer guided by a novel CSR-transition coverage feedback. ProcessorFuzz shows that guiding fuzzing process based on unique CSR transitions can effectively lead to buggy processor states. Additionally, collecting coverage using ISA simulation rather than RTL simulation can quickly eliminate inputs with the same characteristic, thereby helping the fuzzer to test as many qualitatively different inputs as possible. ProcessorFuzz does not require any instrumentation in the processor design, and therefore, can be easily used with different HDLs as long as the target ISA is supported in its framework. Our experimental results exposed eight new bugs in real-world open-source RISC-V processors, and one new bug in a reference model. Moreover, for ground-truth bugs, ProcessorFuzz achieved 1.23× speedup compared to state-of-the-art processor fuzzer DIFUZZRTL.

We demonstrated the capability of ProcessorFuzz using the RISC-V ISA. However, CSRs are not only specific to the RISC-V architecture and defined as part of many other ISAs including x86. Therefore, ProcessorFuzz is not limited to the RISC-V-based processors and can be used in processors based on other ISAs.

ProcessorFuzz uses ISA simulation as part of a feedback mechanism since it is faster and agnostic to the HDL. ProcessorFuzz does not use an input for RTL simulation if the input lacks a unique transition in its ISA simulation trace. One limitation of this design choice is that ProcessorFuzz potentially misses certain bugs that follow the given scenario. If a test input would result in an unintended transition in RTL simulation but the same test input does not cause any unique transition in ISA

simulation, such a test input will be discarded. Hence, the bug will not be identified.

ProcessorFuzz aims to identify functional discrepancies between ISA and the processor design. As future work, one direction is to extend the capability of ProcessorFuzz so that it can expose side channels in processor designs. Exposing side channels require extracting a different type of internal behavior than ProcessorFuzz infers currently. Specifically, ProcessorFuzz needs to observe the data flow between several hardware components (e.g., TLB, caches), and therefore, should rely on a different coverage metric than CSR-transition.

Another research direction is to fuzz software applications for exposing bugs in the processors while an operating system is running in the system. We can identify interesting execution traces by fuzzing software applications and monitoring the coverage change in the processor in parallel. Employing mutations on the interesting execution traces can potentially lead to exploring uncovered regions in the processor.

References

Abdelnur, H., Lucangeli, O. J., and Festor, O. (2010). Spectral fuzzing: Evaluation & feedback. Research Report RR-7193, INRIA. https://hal.inria.fr/inria-00452015/document.

Acharya, V. V., Bagri, S., and Hsiao, M. S. (2015). Branch guided functional test generation at the rtl. In *2015 IEEE European Test Symposium (ETS)*, pages 1–6. http://dx.doi.org/10.1109/ETS.2015.7138737.

aflgo (2017). Aflgo: Directed greybox fuzzing. https://github.com/aflgo/aflgo. Accessed on 18-Feb-2022.

Apache (2001). Apache log4j 2. https://logging.apache.org/log4j/2.x/. Accessed on 17-Feb-2022.

Argyros, G., Stais, I., Jana, S., Keromytis, A. D., and Kiayias, A. (2016). Sfadiff: Automated evasion attacks and fingerprinting using black-box differential automata learning. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1690–1701. https://dl.acm.org/doi/pdf/10.1145/2976749.2978383.

Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D. A., Richards, B., Schmidt, C., Twigg, S., Vo, H., and Waterman, A. (2016). The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

Aschermann, C., Schumilo, S., Abbasi, A., and Holz, T. (2020). Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612. http://doi.org/10.1109/SP40000.2020.00117.

Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asanović, K. (2012). Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 2012 Design Automation Conference*, pages 1212–1221. http://doi.org/10.1145/2228360.2228584.

Balakrishnan, G. and Reps, T. (2004). Analyzing memory accesses in x86 executables. In *International Conference on Compiler Construction*, pages 5–23. http://doi.org/10.1007/978-3-540-24723-4_2.

Baldoni, R., Coppa, E., D'elia, D. C., Demetrescu, C., and Finocchi, I. (2018). A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3):1–39. https://doi.org/10.1145/3182657.

Böhme, M., Pham, V.-T., Nguyen, M.-D., and Roychoudhury, A. (2017a). Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344. https://doi.org/10.1145/3133956.3134020.

Böhme, M., Pham, V.-T., and Roychoudhury, A. (2017b). Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506. https://doi.org/10.1109/TSE.2017.2785841.

Bornmann, L., Leydesdorff, L., and Mutz, R. (2013). The use of percentiles and percentile rank classes in the analysis of bibliometric data: Opportunities and limits. *Journal of informetrics*, 7(1):158–165. https://doi.org/10.1016/j.joi.2012.10.001.

Bose, M., Shin, J., Rudnick, E. M., Dukes, T., and Abadir, M. (2001). A genetic approach to automatic bias generation for biased random instruction generation. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, pages 442–448. http://doi.org/10.1109/CEC.2001.934425.

Brubaker, C., Jana, S., Ray, B., Khurshid, S., and Shmatikov, V. (2014). Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In *2014 IEEE Symposium on Security and Privacy (SP)*, pages 114–129. http://doi.org/10.1109/SP.2014.15.

Bug, T. H. (2014). The heartbleed bug. https://heartbleed.com/. Accessed on 9-Feb-2022.

Cadar, C., Dunbar, D., Engler, D. R., et al. (2008). Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, pages 209–224. http://doi.org/10.5555/1855741.1855756.

Cadence (2019). JasperGold Formal Verification Platform. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html. Accessed on 9-Feb-2022.

Celio, C., Patterson, D. A., and Asanović, K. (2015). The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. Technical Report UCB/EECS-2015-167, EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html.

Celio, C. P. (2017). *A Highly Productive Implementation of an Out-of-Order Processor Generator*. PhD thesis, UC Berkeley. https://digitalassets.lib.berkeley.edu/etd/ucb/text/Celio_berkeley_0028E_17641.pdf.

Cha, S. K., Avgerinos, T., Rebert, A., and Brumley, D. (2012). Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy (SP)*, pages 380–394. http://doi.org/10.1109/SP.2012.31.

Chen, H., Xue, Y., Li, Y., Chen, B., Xie, X., Wu, X., and Liu, Y. (2018). Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108. http://doi.org/10.1145/3243734.3243849.

Chen, M. and Mishra, P. (2011). Property learning techniques for efficient generation of directed tests. *IEEE Transactions on Computers*, 60(6):852–864. http://doi.org/10.1109/TC.2011.49.

Chen, M., Qin, X., Koo, H.-M., and Mishra, P. (2012). *System-level validation: high-level modeling and directed test generation techniques*. Springer Science & Business Media. http://doi.org//10.1007/978-1-4614-1359-2.

Chen, P. and Chen, H. (2018). Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. http://doi.org/10.1109/SP.2018.00046.

Chen, Y., Li, P., Xu, J., Guo, S., Zhou, R., Zhang, Y., Wei, T., and Lu, L. (2020). Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1580–1596. http://doi.org/10.1109/SP40000.2020.00002.

Chen, Y., Su, T., and Su, Z. (2019). Deep differential testing of jvm implementations. In *2019 IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 1257–1268. https://doi.org/10.1109/ICSE.2019.00127.

Chen, Y., Su, T., Sun, C., Su, Z., and Zhao, J. (2016). Coverage-directed differential testing of jvm implementations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–99. https://dl.acm.org/doi/10.1145/2908080.2908095.

Chen, Y. and Su, Z. (2015). Guided differential testing of certificate validation in ssl/tls implementations. In *Proceedings of the 2015 Joint Meeting on Foundations of Software Engineering*, pages 793–804. https://dl.acm.org/doi/10.1145/2786805.2786835.

Chipounov, V., Kuznetsov, V., and Candea, G. (2012). The s2e platform: Design, implementation, and applications. *ACM Transactions on Computer Systems*, 30(1):1–49. https://doi.org/10.1145/2110356.2110358.

Clang (2022). Clang Static Analyzer. https://clang-analyzer.llvm.org/. Accessed on 9-Feb-2022.

CNET (2018). How the equifax hack happened, and what still needs to be done. https://www.cnet.com/news/equifaxs-hack-one-year-later-a-look-back-at-how-it-happened-and-whats-changed/. Accessed on 17-Feb-2022.

Collins, R. R. (1998). The Pentium FOOF bug. Dr. Dobb's Journal. https://www.drdobbs.com/embedded-systems/the-pentium-f00f-bug/184410555. Accessed on 18-Feb-2022.

Coppik, N., Schwahn, O., and Suri, N. (2019). Memfuzz: Using memory accesses to guide fuzzing. In *2019 IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 48–58. http://doi.org/10.1109/ICST.2019.00015.

Dessouky, G., Gens, D., Haney, P., Persyn, G., Kanuparthi, A., Khattri, H., Fung, J. M., Sadeghi, A.-R., and Rajendran, J. (2019). Hardfails: Insights into software-exploitable hardware bugs. In *USENIX Security Symposium (USENIX Security 19)*, pages 213–230. https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky.

Dromajo (2019). Dromajo - Esperanto Technology's RISC-V Reference Model. https://github.com/chipsalliance/dromajo. Accessed on 18-Feb-2022.

Durumeric, Z., Li, F., Kasten, J., Amann, J., Beekman, J., Payer, M., Weaver, N., Adrian, D., Paxson, V., Bailey, M., et al. (2014). The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. http://doi.org/10.1145/2663716.2663755.

Edelman, A. (1997). The mathematics of the pentium division bug. *SIAM Review*, 39(1):54–67. http://doi.org/0.1137/S0036144595293959.

Ekiwi (2020). ekiwi/rfuzz. https://github.com/ekiwi/rfuzz. Accessed on 18-Feb-2022.

EOP (2018). The cost of malicious cyber activity to the u.s. economy. https://www.hsdl.org/?view&did=808776. Accessed on 17-Feb-2022.

EVM (2021). Evm lab utilities: Utilities for interacting with the ethereum virtual machine. https://github.com/ethereum/evmlab. Accessed on 18-Feb-2022.

Fine, S. and Ziv, A. (2003). Coverage directed test generation for functional verification using bayesian networks. In *Proceedings of the 2003 Design Automation Conference*, pages 286–291. http://doi.org/10.1145/775832.775907.

Firefox (2021). Fuzzing. https://firefox-source-docs.mozilla.org/tools/fuzzing. Accessed on 18-Feb-2022.

Fu, Y., Ren, M., Ma, F., Shi, H., Yang, X., Jiang, Y., Li, H., and Shi, X. (2019). Evmfuzzer: detect evm vulnerabilities via fuzz testing. In *Proceedings of the 2019 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1110–1114. https://doi.org/10.1145/3338906.3341175.

Futurewei Technologies, I. (2020). force-riscv. https://github.com/openhwgroup/force-riscv. Accessed on 18-Feb-2022.

Gal, R., Haber, E., Ibraheem, W., Irwin, B., Nevo, Z., and Ziv, A. (2021). Automatic scalable system for the coverage-directed generation (cdg) problem. In *2021 Design, Automation & Test in Europe Conference Exhibition (DATE)*, pages 206–211. http://doi.org/10.23919/DATE51398.2021.9474160.

Gan, S., Zhang, C., Chen, P., Zhao, B., Qin, X., Wu, D., and Chen, Z. (2020). Greyone: Data flow sensitive fuzzing. In *USENIX Security Symposium (USENIX Security 20)*, pages 2577–2594. https://www.usenix.org/conference/usenixsecurity20/presentation/gan.

Gan, S., Zhang, C., Qin, X., Tu, X., Li, K., Pei, Z., and Chen, Z. (2018). Collafl: Path sensitive fuzzing. In *Security and Privacy*, pages 679–696.

Ghaniyoun, M., Barber, K., Zhang, Y., and Teodorescu, R. (2021). Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *2021 ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 874–887. http://doi.org/10.1109/ISCA52012.2021.00073.

glennrp (2018). https://github.com/glennrp/libpng/tree/libpng16/contrib/testpngs. Accessed on 18-Feb-2022.

Godefroid, P., Levin, M. Y., and Molnar, D. (2012). Sage: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44. https://doi.org/10.1145/2093548.2093564.

Google (2017). American Fuzzy Lop. https://github.com/google/AFL. Accessed on 18-Feb-2022.

Google (2019). Clusterfuzz. https://google.github.io/clusterfuzz/. Accessed on 17-Feb-2022.

Google (2019). OSS-Fuzz. https://google.github.io/oss-fuzz/#trophies. Accessed on 17-Feb-2022.

Google (2019). Oss-fuzz: Continuous fuzzing for open source software. https://github.com/google/oss-fuzz. Accessed on 17-Feb-2022.

Google (2020a). Afl dictionaries. https://github.com/google/AFL/tree/master/dictionaries. Accessed on 18-Feb-2022.

Google (2020b). Afl test cases. https://github.com/google/AFL/tree/master/testcases. Accessed on 18-Feb-2022.

Google (2021a). Continuous integration. https://google.github.io/oss-fuzz/getting-started/continuous-integration/. Accessed on 18-Feb-2022.

Google (2021b). Honggfuzz. https://github.com/google/honggfuzz. Accessed on 18-Feb-2022.

Google (2021c). Riscv-dv. https://github.com/google/riscv-dv. Accessed on 18-Feb-2022.

Google (2021). A spectre proof-of-concept for a spectre-proof web. https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html. Accessed on 18-Feb-2022.

Google (2022). Understanding the impact of apache log4j vulnerability. https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html. Accessed on 17-Feb-2022.

Group, S. (2018). Shakti aapg. https://gitlab.com/shaktiproject/tools/aapg/-/wikis/Wiki. Accessed on 18-Feb-2022.

Hazimeh, A., Herrera, A., and Payer, M. (2020a). Magma. hexhive.epfl.ch/magma/docs/bugs.html. Accessed on 18-Feb-2022.

Hazimeh, A., Herrera, A., and Payer, M. (2020b). Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):1–29. https://doi.org/10.1145/3428334.

Herdt, V., Große, D., Jentzsch, E., and Drechsler, R. (2020). Efficient cross-level testing for processor verification: A risc-v case-study. In *2020 Forum for Specification and Design Languages (FDL)*, pages 1–7. IEEE. http://doi.org/10.1109/FDL50818.2020.9232941.

Herrera, A., Gunadi, H., Magrath, S., Norrish, M., Payer, M., and Hosking, A. L. (2021). Seed selection for successful fuzzing. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 230–243. https://doi.org/10.1145/3460319.3464795.

Hur, J., Song, S., Kwon, D., Baek, E., Kim, J., and Lee, B. (2021). Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. http://doi.org/10.1109/SP40001.2021.00103.

Intel (1994). Intel Annual Report. https://www.intel.com/content/www/us/en/history/history-1994-annual-report.html. Accessed on 18-Feb-2022.

Intel (2019). Machine Check Error Avoidance on Page Size Change . https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/software-security-guidance/technical-documentation/machine-check-error-avoidance-page-size-change.html. Accessed on 9-Feb-2022.

Ioannides, C. and Eder, K. I. (2012). Coverage-directed test generation automated by machine learning–a review. *ACM Transactions on Design Automation of Electronic Systems*, 17(1):1–21. https://doi.org/10.1145/2071356.2071363.

Izraelevitz, A., Koenig, J., Li, P., Lin, R., Wang, A., Magyar, A., Kim, D., Schmidt, C., Markley, C., Lawson, J., and Bachrach, J. (2017). Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 209–216. http://doi.org/10.1109/ICCAD.2017.8203780.

Jang, Y., Lee, S., and Kim, T. (2016). Breaking kernel address space layout randomization with intel tsx. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, page 380–392. https://doi.org/10.1145/2976749.2978321.

Jung, J., Sheth, A., Greenstein, B., Wetherall, D., Maganis, G., and Kohno, T. (2008). Privacy oracle: a system for finding application leaks with black box differential testing. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 279–288. https://doi.org/10.1145/1455770.1455806.

Kabylkas, N., Thorn, T., Srinath, S., Xekalakis, P., and Renau, J. (2021). Effective processor verification with logic fuzzer enhanced co-simulation. In *IEEE/ACM International Symposium on Microarchitecture*, pages 667–678. https://doi.org/10.1145/3466752.3480092.

Klees, G., Ruef, A., Cooper, B., Wei, S., and Hicks, M. (2018). Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and*

Communications Security, pages 2123–2138. https://doi.org/10.1145/3243734.3243804.

Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. (2019). Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. http://doi.org/10.1109/SP.2019.00002.

Laeufer, K. et al. (2018). Rfuzz: coverage-directed fuzz testing of rtl on fpgas. In *Proceedings of the International Conference on Computer-Aided Design*, pages 1–8. https://doi.org/10.1145/3240765.3240842.

Lee, G., Shim, W., and Lee, B. (2021). Constraint-guided directed greybox fuzzing. In *USENIX Security Symposium (USENIX Security 21)*, pages 3559–3576. https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu.

Lee, Y. and Cook, H. (2015). riscv-torture. https://github.com/ucb-bar/riscv-torture. Accessed on 18-Feb-2022.

Li, T., Zou, H., Luo, D., and Qu, W. (2021a). Symbolic simulation enhanced coverage-directed fuzz testing of rtl design. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. http://doi.org/10.1109/ISCAS51556.2021.9401267.

Li, Y., Ji, S., Chen, Y., Liang, S., Lee, W.-H., Chen, Y., Lyu, C., Wu, C., Beyah, R., and Cheng, P. (2021b). Unifuzz: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *USENIX Security Symposium (USENIX Security 21)*. https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei.

Liang, H., Zhang, Y., Yu, Y., Xie, Z., and Jiang, L. (2019). Sequence coverage directed greybox fuzzing. In *2019 IEEE/ACM International Conference on Program Comprehension (ICPC)*, pages 249–259. IEEE Computer Society. http://doi.org/10.1109/ICPC.2019.00044.

Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. (2018). Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium (USENIX Security 18)*, pages 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

Livshits, V. B. and Lam, M. S. (2005). Finding security vulnerabilities in java applications with static analysis. In *USENIX Security Symposium (USENIX Security 05)*, pages 18–18. https://www.usenix.org/conference/14th-usenix-security-symposium/finding-security-vulnerabilities-java-applications-static.

LLVM (2021). libfuzzer. https://llvm.org/docs/LibFuzzer.html\#corpus. Accessed on 18-Feb-2022.

LLVM (2022a). AddressSanitizer. https://clang.llvm.org/docs/AddressSanitizer.html. Accessed on 9-Feb-2022.

LLVM (2022b). LeakSanitizer. https://clang.llvm.org/docs/LeakSanitizer.html. Accessed on 9-Feb-2022.

LLVM (2022c). MemorySanitizer. https://clang.llvm.org/docs/MemorySanitizer.html. Accessed on 9-Feb-2022.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200. https://doi.org/10.1145/1065010.1065034.

Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.-H., Song, Y., and Beyah, R. (2019). Mopt: Optimized mutation scheduling for fuzzers. In *USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu.

Machiry, A., Spensky, C., Corina, J., Stephens, N., Kruegel, C., and Vigna, G. (2017). Dr.checker: A soundy analysis for linux kernel drivers. In *USENIX Security Symposium (USENIX Security 17)*, pages 1007–1024. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/machiry.

Manès, V. J., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., and Woo, M. (2019). The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331. http://doi.org/0.1109/TSE.2019.2946563.

Martignoni, L., Paleari, R., Roglia, G. F., and Bruschi, D. (2009). Testing cpu emulators. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 261–272. https://doi.org/10.1145/1572272.1572303.

Microsoft (2017). Microsoft security risk detection. https://www.microsoft.com/en-us/security-risk-detection/. Accessed on 17-Feb-2022.

Min, C., Kashyap, S., Lee, B., Song, C., and Kim, T. (2015). Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the Symposium on Operating Systems Principles*, pages 361–377. https://doi.org/10.1145/2815400.2815422.

MITRE (2019). Hardware design CWEs. https://cwe.mitre.org/data/definitions/1194.html. Accessed on 18-Feb-2022.

Moundanos, D., Abraham, J. A., and Hoskote, Y. V. (1998). Abstraction techniques for validation coverage analysis and test generation. *IEEE Transactions on Computers*, 47(1):2–14. http://doi.org/10.1109/12.656068.

Muduli, S. K., Takhar, G., and Subramanyan, P. (2020). Hyperfuzzing for soc security validation. In *Proceedings of the International Conference on Computer-Aided Design*, pages 1–9. https://doi.org/10.1145/3400302.3415709.

Mukherjee, R., Kroening, D., and Melham, T. (2015). Hardware verification using software analyzers. In *2015 IEEE Computer Society Annual Symposium on VLSI*, pages 7–12. http://doi.org/0.1109/ISVLSI.2015.107.

Nativ, G., Mittennaier, S., Ur, S., and Ziv, A. (2001). Cost evaluation of coverage directed test generation for the ibm mainframe. In *Proceedings International Test Conference 2001 (Cat. No.01CH37260)*, pages 793–802. http://doi.org/10.1109/TEST.2001.966701.

Nesselroade Jr, K. P. and Grimm, L. G. (2018). *Statistical applications for the behavioral and social sciences*. John Wiley & Sons. http://doi.org/10.1002/9781119531708.

Newsome, J. and Song, D. X. (2005). Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, pages 3–4. https://www.ndss-symposium.org/ndss2005/dynamic-taint-analysis-automatic-detection-analysis-and-signaturegeneration-exploits-commodity/.

Nguyen, M.-D., Bardin, S., Bonichon, R., Groz, R., and Lemerre, M. (2020). Binary-level directed fuzzing for use-after-free vulnerabilities. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 47–62. https://www.usenix.org/conference/raid2020/presentation/nguyen.

Nilizadeh, S., Noller, Y., and Pasareanu, C. S. (2019). Diffuzz: differential fuzzing for side-channel analysis. In *Proceedings of the 2019 IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 176–187. http://doi.org/10.1109/ICSE.2019.00034.

Noller, Y., Păsăreanu, C. S., Böhme, M., Sun, Y., Nguyen, H. L., and Grunske, L. (2020). Hydiff: Hybrid differential software analysis. In *Proceedings of the 2020 ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 1273–1285. https://doi.org/10.1145/3377811.3380363.

NVD (2020). Cve-2020-10029 detail. https://nvd.nist.gov/vuln/detail/CVE-2020-10029. Accessed on 18-Feb-2022.

Österlund, S., Razavi, K., Bos, H., and Giuffrida, C. (2020). Parmesan: Sanitizer-guided greybox fuzzing. In *USENIX Security Symposium (USENIX Security 20)*, pages 2289–2306. https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund.

Pailoor, S., Aday, A., and Jana, S. (2018). Moonshine: Optimizing os fuzzer seed selection with trace distillation. In *USENIX Security Symposium (USENIX Security 18)*, pages 729–743. https://www.usenix.org/conference/usenixsecurity18/presentation/pailoor.

Paleari, R., Martignoni, L., Roglia, G. F., and Bruschi, D. (2009). A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proceedings of the USENIX Conference on Offensive Technologies*, page 86. http://doi.org/10.5555/1855876.1855878.

Pei, K., Cao, Y., Yang, J., and Jana, S. (2017). Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the Symposium on Operating Systems Principles*, pages 1–18. https://doi.org/10.1145/3132747.3132785.

Peng, J., Li, F., Liu, B., Xu, L., Liu, B., Chen, K., and Huo, W. (2019). 1dvul: Discovering 1-day vulnerabilities through binary patches. In *2019 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 605–616. http://doi.org/10.1109/DSN.2019.00066.

Petrisko, D., Gilani, F., Wyse, M., Jung, D. C., Davidson, S., Gao, P., Zhao, C., Azad, Z., Canakci, S., Veluri, B., Guarino, T., Joshi, A., Oskin, M., and Taylor, M. B. (2020). Blackparrot: An agile open-source risc-v multicore for accelerator socs. *IEEE Micro*, 40(4):93–102. 10.1109/MM.2020.2996145.

Petsios, T., Tang, A., Stolfo, S., Keromytis, A. D., and Jana, S. (2017). Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 615–632. http://doi.org/10.1109/SP.2017.27.

Rebert, A., Cha, S. K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., and Brumley, D. (2014). Optimizing seed selection for fuzzing. In *USENIX Security Symposium (USENIX Security 14)*, pages 861–875. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert.

Sadeghi, A.-R., Rajendran, J., and Kande, R. (2021). Organizing the world's largest hardware security competition: Challenges, opportunities, and lessons learned. In *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, pages 95–100. https://doi.org/10.1145/3453688.3464508.

Sahin, O., Coskun, A. K., and Egele, M. (2018). Proteus: Detecting android emulators from instruction-level profiles. In *2018 International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 3–24. https://doi.org/10.1007/978-3-030-00470-5_1.

Schwartz, E. J., Avgerinos, T., and Brumley, D. (2010). All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE Symposium on Security and Privacy (SP)*, pages 317–331. http://doi.org/10.1109/SP.2010.26.

Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., and Gruss, D. (2019). Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 753–768. https://doi.org/10.1145/3319535.3354252.

Shankar, U., Talwar, K., Foster, J. S., and Wagner, D. (2001). Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium (USENIX Security 01)*. https://www.usenix.org/legacy/publications/library/proceedings/sec01/shankar.html.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., et al. (2016). Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157. http://doi.org/10.1109/SP.2016.17.

Sifive (2020). sifive-blocks. https://github.com/sifive/sifive-blocks/tree/master/src/main/scala/devices. Accessed on 18-Feb-2022.

Sivakorn, S., Argyros, G., Pei, K., Keromytis, A. D., and Jana, S. (2017). Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 521–538. http://doi.org/10.1109/SP.2017.46.

Spike (2019). Spike RISC-V ISA Simulator. https://github.com/riscv-software-src/riscv-isa-sim. Accessed on 18-Feb-2022.

SQLite (2021). Sqlite source repository. https://github.com/sqlite/sqlite/tree/master/test. Accessed on 18-Feb-2022.

Squillero, G. (2005). Microgp—an evolutionary assembly program generator. *Genetic Programming and Evolvable Machines*, 6(3):247–263. http://doi.org/10.1007/s10710-005-2985-x.

Staff, H. I. N. (2018). The biggest healthcare data breaches of 2018. https://www.healthcareitnews.com/projects/biggest-healthcare-data-breaches-2018-so-far. Accessed on 19-Feb-2020.

Synopsys (2019). Synthesis Design Compiler. https://www.synopsys.com/support/training/rtl-synthesis.html. Accessed on 18-Feb-2022.

Tasiran, S., Fallah, F., Chinnery, D. G., Weber, S. J., and Keutzer, K. (2001). A functional validation technique: biased-random simulation guided by observability-based coverage. In *Proceedings 2001 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pages 82–88. http://doi.org/10.1109/ICCD.2001.955007.

Trippel, T., Shin, K. G., Chernyakhovsky, A., Kelly, G., Rizzo, D., and Hicks, M. (2021). Fuzzing hardware like software. *CoRR*, abs/2102.02308. https://arxiv.org/abs/2102.02308.

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

Ucb-art (2020). ucb-art/fft. https://github.com/ucb-art/fft. Accessed on 18-Feb-2022.

Ucb-bar (2015). The Sodor processor: educational microarchitectures for RISC-V ISA. https://github.com/ucb-bar/riscv-sodor. Accessed on 18-Feb-2022.

Valgrind (2022). Valgrind. https://valgrind.org/. Accessed on 9-Feb-2022.

Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. (2018). Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *USENIX Security Symposium (USENIX Security 18)*, pages 991–1008. https://www.usenix.org/conference/usenixsecurity18/presentation/bulc.

Verilator (2003). Verilator. https://github.com/verilator. Accessed on 18-Feb-2022.

Wagner, I., Bertacco, V., and Austin, T. (2005). Stresstest: an automatic approach to test generation via activity monitors. In *Proceedings of the Design Automation Conference*, pages 783–788. https://doi.org/10.1145/1065579.1065788.

Wang, H., Xie, X., Li, Y., Wen, C., Li, Y., Liu, Y., Qin, S., Chen, H., and Sui, Y. (2020a). Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 999–1010. https://doi.org/10.1145/3377811.3380386.

Wang, Y., Jia, X., Liu, Y., Zeng, K., Bao, T., Wu, D., and Su, P. (2020b). Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *Network and Distributed System Security Symposium (NDSS)*. https://www.ndss-symposium.org/wp-content/uploads/2020/02/24422-paper.pdf.

Wcventure (2022). Recent papers related to fuzzing. https://github.com/wcventure/FuzzingPaper. Accessed on 18-Feb-2022.

Weber, D., Ibrahim, A., Nemati, H., Schwarz, M., and Rossow, C. (2021). Osiris: Automated discovery of microarchitectural side channels. In *USENIX Security Symposium (USENIX Security 21)*, pages 1415–1432. https://www.usenix.org/conference/usenixsecurity21/presentation/weber.

Wen, C., Wang, H., Li, Y., Qin, S., Liu, Y., Xu, Z., Chen, H., Xie, X., Pu, G., and Liu, T. (2020). Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 765–777. https://doi.org/10.1145/3377811.3380396.

Wojtczuk, R. (2012). PV Privilege Escalation. https://lists.xen.org/archives/html/xen-announce/2012-06/msg00001.html. Accessed on 9-Feb-2022.

Wolf, C. (2015). Yosys open synthesis suite. https://yosyshq.net/yosys/. Accessed on 18-Feb-2022.

Xu, W., Kashyap, S., Min, C., and Kim, T. (2017). Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2313–2328. https://doi.org/10.1145/3133956.3134046.

Yamaguchi, F., Golde, N., Arp, D., and Rieck, K. (2014). Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. http://doi.org/10.1109/SP.2014.44.

Yang, X., Chen, Y., Eide, E., and Regehr, J. (2011). Finding and understanding bugs in c compilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294. https://doi.org/10.1145/1993498.1993532.

Yang, Y., Kim, T., and Chun, B.-G. (2021). Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 349–365. https://www.usenix.org/conference/osdi21/presentation/yang.

Yue, T., Wang, P., Tang, Y., Wang, E., Yu, B., Lu, K., and Zhou, X. (2020). Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *USENIX Security Symposium (USENIX Security 20)*, pages 2307–2324. https://www.usenix.org/conference/usenixsecurity20/presentation/yue.

Zalewski, M. (2017). afl-cmin. https://github.com/mirrorer/afl/blob/master/afl-cmin. Accessed on 18-Feb-2022.

Zalewski, M. (2020). afl-tmin. https://github.com/google/AFL/blob/master/afl-tmin.c. Accessed on 18-Feb-2022.

Zhu, X. and Böhme, M. (2021). Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2169–2182. https://doi.org/10.1145/3460120.3484596.

Zong, P., Lv, T., Wang, D., Deng, Z., Liang, R., and Chen, K. (2020). Fuzzguard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *USENIX Security Symposium (USENIX Security 20)*, pages 2255–2269. https://www.usenix.org/conference/usenixsecurity20/presentation/zong.

CURRICULUM VITAE