MorFuzz: Fuzzing Processor via Runtime Instruction Morphing enhanced Synchronizable Co-simulation

Jinyan Xu
Zhejiang University
phantom@zju.edu.cn

Yiyuan Liu
Zhejiang University
yiyuanliu@zju.edu.cn

Sirui He
City University of Hong Kong
sol.he@my.cityu.edu.hk

Haoran Lin
Zhejiang University
haoran_lin@zju.edu.cn

Yajin Zhou*
Zhejiang University
yajin_zhou@zju.edu.cn

Cong Wang
City University of Hong Kong
congwang@cityu.edu.hk

Abstract

Modern processors are too complex to be bug free. Recently, a few hardware fuzzing techniques have shown promising results in verifying processor designs. However, due to the complexity of processors, they suffer from complex input grammar, deceptive mutation guidance, and model implementation differences. Therefore, how to effectively and efficiently verify processors is still an open problem.

This paper proposes MorFuzz, a novel processor fuzzer that can efficiently discover software triggerable hardware bugs. The core idea behind MorFuzz is to use runtime information to generate instruction streams with valid formats and meaningful semantics. MorFuzz designs a new input structure to provide multi-level runtime mutation primitives and proposes the instruction morphing technique to mutate instruction dynamically. Besides, we also extend the co-simulation framework to various microarchitectures and develop the state synchronization technique to eliminate implementation differences. We evaluate MorFuzz on three popular open-source RISC-V processors: CVA6, Rocket, BOOM, and discover 17 new bugs (with 13 CVEs assigned). Our evaluation shows MorFuzz achieves 4.4× and 1.6× more state coverage than the state-of-the-art fuzzer, DifuzzRTL, and the famous constrained instruction generator, riscv-dv.

1 Introduction

With extensions to improve performance and extend functionality, processor designs are becoming more and more sophisticated. Modern processors are extremely large and complex, typically with billions of transistors and multiple cores. Meanwhile, processors also become increasingly error-prone, and even the latest commodity processors suffer from hardware bugs. For example, Intel discovered 42 errata in their 12th-gen CPUs [13]. Bugs in the processor not only produce incorrect computations (e.g., the infamous Pentium FDIV bug [11] returns inaccurate floating-point division results) but also cause devastating errors, such as unpredictable system behavior, locking up the machine, and even software security corruptions. The hyper-threading bug [12] can cause data corruption or loss in general-purpose registers, the Barcelona TLB bug [10] and the Pentium F00F bug [11] freeze up the processor, and vulnerabilities like SYSRET bug [15] and memory sinkhole [17] allow unprivileged code escalate into higher privilege. Since hardware bugs are difficult to patch after the chip is manufactured, it is vital to discover bugs in the pre-silicon phase.

Two main methods are proposed to discover hardware bugs automatically: static formal verification [21, 27, 40, 46, 69, 71] and dynamic simulation-based verification. While formal verification methods can thoroughly verify small designs, they are limited by the state explosion problem and fail to scale to large, complex designs such as processors. To automatically maximize the exploration of the state space of the processor under test, researchers proposed two mechanisms, constrained random verification [28, 35, 41, 66] and coverage guided test generation [22, 56, 59, 60], to direct the simulation-based methods to generate better test cases. However, these dynamic methods both require design-specific knowledge to define the generation strategies, which require heavy manual effort.

Recently, fuzzing has become the most popular and effective method in software systems due to the ability to discover unknown vulnerabilities with minimal knowledge [4, 25, 29, 47, 51, 55]. Inspired by the effectiveness of fuzzing, researchers started to apply software fuzzing to processors [6, 7, 30, 32, 33, 37, 39]. Unfortunately, according to our evaluation (§5.3), existing fuzzers are still far from being adopted in practice. Previous efforts fail to effectively and efficiently fuzz processors because of the following three challenges.

First, the input grammar of the processor is complex. Processors usually support many different instructions, each of which has its own unique format. Moreover, these instructions require different types of operands (e.g., integers, floating-point numbers, addresses) to perform meaningful operations, further complicating the input grammar. Existing fuzzers [30, 32, 33] statically generate and mutate instructions, re-

*The corresponding author.

sulting in limited mutation primitives and missing effective semantics. The second challenge is that the control transfer instructions (such as jump and branch instructions) impair the effectiveness of mutations. Existing fuzzers ignore the interference of the input's control flow on the coverage. As a result, valuable mutations may be skipped because of the control transfer instructions and thus incorrectly discarded. And the third issue with existing fuzzers is the implementation differences between models. Almost all previous fuzzers [6, 30, 32, 33] introduce a reference model to check the correctness of the processor. By comparing the state of the processor with that of the reference model, they treat the mismatched states as bugs. However, software reference models are inherently different from hardware, and not all differences are bugs. These false positives caused by implementation differences misguide the fuzzers and prevent them from covering the deep states of the processor.

We address the aforementioned challenges with MorFuzz, a novel processor fuzzer that can detect software triggerable hardware bugs efficiently. MorFuzz addresses the first two challenges by dynamically generating diverse and meaningful instruction streams based on the runtime information. First, MorFuzz introduces a new input structure, the stimulus template, to explore the processor's input space from multiple dimensions. The stimulus template provides primitives to mutate inputs at the processor state, instruction field, and program semantic levels. Second, MorFuzz uses runtime information to morph instructions dynamically. We propose the instruction morphing technique, which collects contextual information from the processor at runtime to mutate instructions with valid formats and meaningful semantics. In addition, since all mutations are executed, the coverage correctly reflects the effect of the mutations, achieving efficient mutation guidance. Finally, MorFuzz eliminates implementation differences through state synchronization. We extend the co-simulation framework to various microarchitectures and add state synchronization support. This allows MorFuzz to identify the source of the differences and synchronize the hardware state to the reference model to eliminate legal differences.

We have implemented a prototype of MorFuzz on RISC-V architecture and evaluated it on three real-world open-source processors: CVA6 [68], Rocket [1], and BOOM [70]. These processors under evaluation cover various microarchitectures, from simple in-order cores to complex out-of-order superscalar cores. Our evaluation shows that MorFuzz achieves at most 4.4× and 1.6× more state coverage than the state-of-the-art processor fuzzer, DifuzzRTL, and the famous constrained instruction generator, riscv-dv, respectively. In terms of performance, MorFuzz achieves the coverage that DifuzzRTL takes 24 hours to achieve in about 30 minutes and takes about 2.4 hours to achieve the coverage that riscv-dv takes 24 hours to complete. MorFuzz identified 17 new bugs in total, 13 of which are assigned with CVE numbers, and all of these bugs are confirmed by the respective communities.

In summary, this paper makes the following contributions:

- We propose a novel processor fuzzing approach that uses runtime information to dynamically generate meaningful input and efficiently guide mutation.

- We present the design and implementation of MorFuzz, a processor fuzzing framework that can efficiently detect software triggerable hardware bugs. MorFuzz achieves 4.4× and 1.6× higher coverage than the state-of-the-art processor fuzzer, DifuzzRTL, and the famous constrained instruction generator, riscv-dv, respectively.

- MorFuzz is a generic RISC-V processor fuzzer that is compatible with various microarchitectures. We evaluate MorFuzz on three popular real-world RISC-V processors (CVA6, Rocket, BOOM) and totally discover 17 new bugs (with 13 CVEs assigned).

- To facilitate the community and future research, we release the source code of MorFuzz at `https://github.com/sycuricon/MorFuzz`.

2 Background

2.1 RISC-V Instruction Set Architecture

The RISC-V instruction set architecture (ISA) is an open-source reduced instruction set architecture that has gradually become popular in industry and academia. It is composed of a base integer instruction set and a set of optional instruction-set extensions. The standard extensions contain integer multiplication and division, atomic memory operations, single/double-precision floating-point, and compressed instructions. In addition, the control and status register (CSR) instruction extension provides control over the privileged architecture, and the instruction-fetch fence extension is designed to synchronize the instruction memory.

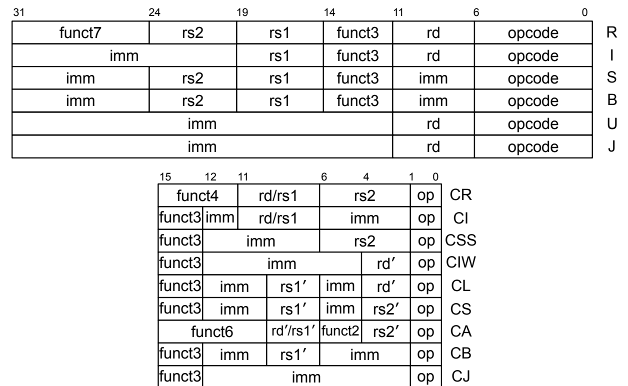


Figure 1: RISC-V instruction formats.

RISC-V instructions currently have two valid lengths. Except for the 16-bit compressed instructions, all other instructions are 32-bit width. Figure 1 shows all 15 instruction formats, each consisting of multiple fields. The format of the 32-bit width instruction is determined by the `opcode` field, while the `op` and the `funct` fields determine that of the 16-bit width instruction. Currently, there are two categories of instruction fields. The first category is opcode related fields. The `funct` and `opcode` fields are used to determine the instruction's operation, also known as the opcode. The lower two bits of the `opcode` field and the `op` field are used to determine the length of the instruction. The `funct` fields and the other bits of the `opcode` field are used to determine the opcode type. Typically, instructions with similar functions have the same `opcode` field and are distinguished by the `funct` fields. The second category is the operand related fields. The `imm` and `rs` fields are designed to provide the operands. The `rs` fields are used to select the source registers, and the `imm` fields are used as the immediate number. And the `rd` fields are used to control the destination register, where the result of the instruction is written back.

2.2 Processor Verification

Unlike software, hardware cannot be easily patched once manufactured. To avoid pre-silicon bugs from escaping to post-silicon, verification is performed throughout the development process. Statistically, about 56% of the project time is spent on verification [23].

2.2.1 Typical Processor Verification

The processor is a finite state machine, and its state includes the microarchitectural state and the architectural state. The microarchitectural state represents the implementation-related internal state that is transparent to the outside of the processor. In contrast, the architectural state holds the state of a program (e.g., the memory and the general-purpose registers) and is consistent across the same ISA. We denote the implementation of the processor design under test (DUT) as a function f_{DUT}, S_{DUT} denotes the state of the DUT in current cycle. At each cycle, the processor generates the next state S'_{DUT} based on the current state S_{DUT} and the external input I (i.e., instructions): $f_{DUT}(I, S_{DUT}) \rightarrow S'_{DUT}$. The task of processor verification is to check whether the implementation function f_{DUT} is a valid subset of the implementation function f_{spec} defined in the specification.

Researchers deploy two main methods to verify hardware designs: static formal verification [21, 27, 40, 46, 69, 71] and dynamic simulation-based verification. As formal verification is limited to scale to complex designs [16], simulation-based verification is more prevalent in practice. The simulation-based verification uses tailored input to simulate the DUT and verify whether the output of the DUT meets expectations.

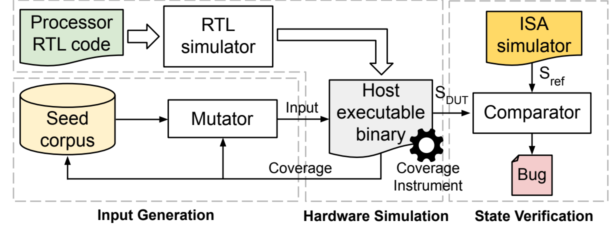


Figure 2: Hardware fuzzer workflow.

The typical simulation-based verification method involves the following three phases. First, the test case generator randomly generates instruction streams based on constraints [28, 35, 41, 66] or coverage [22, 56, 59, 60]. Next, the RTL simulator [14, 52, 65] translates the RTL code of the processor under test into a software model. The simulator then compiles the software model with its test harness (containing the input interpreter) into a host executable binary file. The simulation is performed by executing the binary file, and during the simulation, the input is translated into the bus transactions that are recognized by the DUT. Finally, the correctness of the DUT's behavior during the simulation is checked by verifying the external visible architectural state of the processor due to the difficulty of checking an abstract implementation. A golden reference model is introduced to execute the same input, and the correctness of the DUT can be determined by comparing the architectural state of the DUT $S_{DUT_{arc}}$ and the reference model $S_{REF_{arc}}$.

2.2.2 Processor Fuzzing

The typical processor verification method described above is limited by the quality of the generated test cases, struggling to cover corner cases. Fuzzing has recently become a popular testing technique for automatically detecting software security vulnerabilities. Driven by the success of fuzzing, researchers have recently proposed to apply it to processor verification [6, 7, 30, 32, 33, 37, 39]. Figure 2 illustrates the general workflow of the existing processor fuzzing frameworks, which also consist of three phases. In the input generation phase, the fuzzer generates instruction streams using the seeds and mutates the instruction streams based on the coverage of the previous round. DifuzzRTL [30] uses the static analysis technique to generate instructions with required operands, and TheHuzz [33] optimizes mutations according to its optimal weights. In the hardware simulation phase, the RTL code of the DUT is also translated into the host executable binary file. During the simulation, the fuzzer uses the instruments in the hardware to collect the coverage of the current input. Existing fuzzers have designed various coverage matrices, such as mux coverage [37], control register coverage [30], and hardware behavior coverage [33]. In the state verification phase, the fuzzer extracts the DUT's architectural state and

```
start:
    call init_regs
    call init_page_table
l1:
    addi x2, x4, -935
l2:
    la x2, l86
    jalr x20, 0(x2)
    # ...
l86:
    csrrw x6, satp, x5
l87:
    blt x25, x6, exit
    # ...
exit:
    call signature
```

Figure 3: Example test case generated by DifuzzRTL.

then compares it with a reference model (e.g., an ISA simulator) and reveals the mismatches as bugs. However, previous fuzzers simply port software fuzzing to the traditional verification flow while ignoring the challenges of processor fuzzing. According to our statistics in Figure 8, the performance of the state-of-the-art processor fuzzer, DifuzzRTL [30], is even worse than using randomly generated test cases.

2.3 Challenges of Processor Fuzzing

We use the test case (Figure 3) generated by the state-of-the-art processor fuzzer DifuzzRTL [30] as an example to articulate the challenges of processor fuzzing and analysis why previous fuzzers fail to effectively and efficiently fuzz processors. In the first three lines, the test case initializes the execution environment. Lines 4 to 14 are instructions used to fuzz the functionality of the processor. Each label is a test point, and DifuzzRTL typically generates about 180 test points on average in one test case. In the end, the test case dumps the architectural state of the processor to memory as the signature and exits the simulation (line 16).

Complex Input Grammar. The processor's behavior is determined not only by the external input instructions I but also by its current state S_{DUT}. Since the state of the processor is accumulated from previous instructions, instruction sequences also affect the state of the processor. And the instruction itself also contains two variables, the opcode and the operand, both of which might take on legal or illegal values. Based on these multi-dimensional parameters, processor inputs also have complex semantics, and an instruction only performs meaningful operations with valid operands in a particular execution environment. Existing fuzzers fail to generate diverse and meaningful instruction streams limited by static generation and unidimensional mutation. For example, DifuzzRTL uses approximate static analysis to select operands, while TheHuzz randomly mutates fields ignoring semantics.

Deceptive Mutation Guidance. Unlike software fuzzing, the input of the processor fuzzing contains control transfer instructions and exceptions. For this reason, the generated instructions are not guaranteed to be executed, so the coverage actually reflects the effect of the executed instructions rather than the effect of the generated instructions. Unfortunately, existing fuzzers all choose the latter, making the coverage misleading to the mutation. For example, the `jalr` instruction on line 8 jumps from `l2` to `l86`, causing all instructions from `l3` to `l85` to be skipped. Suppose the skipped instructions contain some valuable mutations, and the executed instructions do not contribute to the coverage. The fuzzer will consider all these mutations unhelpful and will eventually discard them. As a result, the coverage incorrectly guides the fuzzing toward an ineffective direction.

Model Implementation Differences. Existing fuzzers use an ISA simulator as the reference model to detect hardware bugs. However, the ISA simulator is only a functional model of the processor, and there are some inherent differences compared with the actual hardware. For example, the ISA simulator is cycle inaccurate and lacks peripheral simulation. Therefore the two models will get mismatched values when accessing these registers. Another source of the differences is the indeterminateness in the specification. The RISC-V specification does not restrict the implementation, so potentially multiple behaviors are allowed. For instance, the property of the CSR (e.g., `satp`) usually is "Write Any Read Legal", which means that even if the same value is written to the CSR, the value readout may differ depending on the implementation (line 11). Unfortunately, these differences are legal in the specification. And even worse, since the state verification phase is offline, these differences can cause the two models' control flows to diverge. For instance, the DUT uses the mismatched value to execute the branch instruction at line 13. If the DUT and the simulator do not perform consistent branch behavior, it will lead to completely mismatched subsequent traces, resulting in meaningless execution.

Inefficient Execution. The duplicated instruction streams have no contribution to the coverage. For example, before the fuzzing payload in the test case is executed, the fuzzer spends considerable time loading the test case into the DUT's memory and waiting for the DUT to execute several initialization functions (e.g., `init_regs`, `init_page_table`) to set up the environment. However, DifuzzRTL can only access the DUT through limited ports provided by the test harness, and once the simulation starts, the fuzzer has no control over the control flow of the test case. Due to the poor controllability of the DUT, the time-consuming initialization process is repeatedly executed without any improvement in coverage, which results in ineffective fuzzing.

3 Design

MorFuzz is a novel coverage-guided processor fuzzer that can efficiently detect software triggerable hardware bugs. In

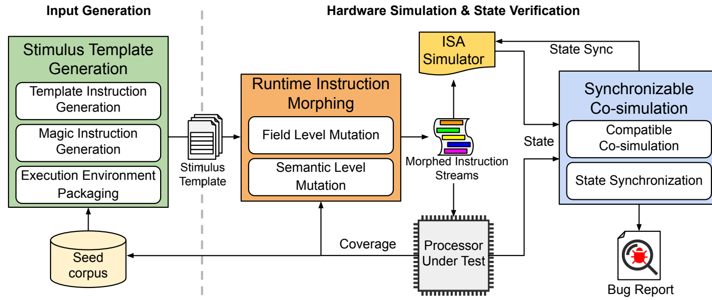


Figure 4: Overview of MorFuzz.

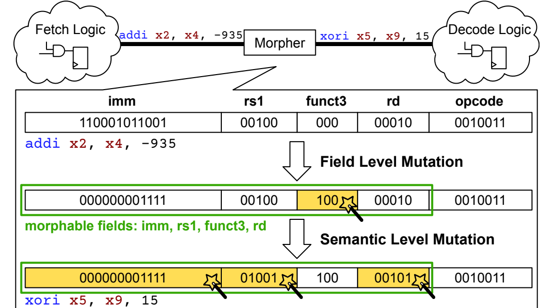


Figure 5: Example of instruction morphing.

this section, we first give an informal verification scope of MorFuzz and then elaborate on the design details.

3.1 Verification Scope

Unlike previous fuzzers [7, 32, 37, 58] that focused on bugs triggered by specific hardware signals, MorFuzz is designed to detect architecture functional bugs triggered by software. Specifically, MorFuzz focuses on bugs triggered by specific combinations of instructions that cause the processor's behavior to deviate from the ISA specification. Therefore, the processor's behaviors under any privilege need to be verified, and behaviors that are undefined or unconstrained in the specification are out of our verification scope. MorFuzz will trust the high privilege levels it relies on and use simplified firmware to provide the required functionality when testing the processor's behavior at low privilege levels. In addition, transient execution bugs caused by microarchitecture mistakes are out of our scope [42, 64].

3.2 Architecture Overview

The core idea behind MorFuzz is to dynamically mutate instructions based on the runtime feedback. In summary, MorFuzz leverages the following techniques to resolve the aforementioned challenges (§2.3): the stimulus template, the instruction morphing, and the synchronizable co-simulation.

Stimulus Template (§3.3). Unlike existing fuzzers that directly generate instruction streams as the stimulus, MorFuzz uses a stimulus template to generate diverse and meaningful instruction streams. The stimulus template provides multi-level runtime mutation primitives, including processor state level, instruction field level, and program semantic level, thereby comprehensively exploring the input space of the processor. In addition, the stimulus template also introduces the ability for the fuzzer to communicate with the DUT to manage the control flow of test cases. Therefore, the fuzzer can accurately control the DUT to skip duplicate instructions and focus on the instruction sequences it is interested in.

Instruction Morphing (§3.4). Instruction morphing only mutates those instructions that are going to be executed instead of mutating instructions indiscriminately like existing fuzzers. Instruction morphing is a dynamic instruction mutation technique that accurately reflects the effect of the mutations, and thereby the coverage can effectively guide the fuzzer. And instruction morphing uses runtime information to mutate opcodes and operands, ensuring that the morphed instructions maintain valid field format and meaningful semantics. Besides, instruction morphing is performed on binary instructions, which makes it easier for MorFuzz to generate corner cases that assembly language cannot represent, thus greatly increasing the efficiency in exploring the input space.

Synchronizable Co-simulation (§3.5). MorFuzz synchronizes the legal differences to address the implementation differences between models. During the simulation, MorFuzz uses a simulator to co-simulate with the DUT, and compares the architectural state of the DUT and the simulator after each instruction is executed to check the correctness. Co-simulation can also allow MorFuzz to locate which instruction caused the mismatched state accurately. Based on this, MorFuzz can further analyze whether the difference is legal and synchronize the correct state from the DUT to the simulator, thus eliminating the mismatch. Benefiting from the synchronizable co-simulation framework, MorFuzz can automatically mitigate the implementation differences and allow the simulator to co-simulate synchronously with the DUT, thus directing the fuzzer to cover more depth states.

Overview. The overall workflow of MorFuzz is depicted in Figure 4. First, MorFuzz uses seeds to generate the stimulus templates. Then, MorFuzz dynamically morphs the template based on the runtime information and executes morphed instruction streams simultaneously on the DUT and the simulator. Finally, after each instruction is executed, MorFuzz compares the architectural state of the two models. After MorFuzz analyzes the mismatches, the legal difference states are synchronized to the simulator, and the others are reported as potential bugs.

3.3 Stimulus Template Generation

We design a new structure for the test case, the stimulus template, to provide runtime mutation primitives for processor

state and instructions. The stimulus template consists of two parts: the runtime morphable fuzzing payload and the read-only fuzzing execution environment. The fuzzing payload contains the runtime mutation primitives, and the fuzzing execution environment is the system firmware responsible for providing a software execution environment that allows the DUT to execute the fuzzing payload continuously.

Template Instruction Generation. Template instructions are blank payload instructions for instruction morphing, which provide mutation primitives for the instruction field and the program semantic at runtime. During the generation, the template instruction acts like a placeholder, only containing the fields that determine the length of the instruction to calculate the memory layout of the stimulus template. The other fields are temporarily filled with dummy values, which MorFuzz will replace with meaningful values based on the contextual information later during the simulation. MorFuzz generates template instructions at block granularity and designs different testing blocks to cover the various hardware functional modules of the processor. A set of sequence patterns are manually constructed in each testing block to constrain the instruction types of each template instruction in the block to achieve the desired test points. Under the constraints of the sequence pattern, each testing block is randomly filled with a bunch of template instruction sequences with special semantics. In addition, the sequence patterns also expose the DUT's internal state by inserting watchpoint instructions at specific locations to enhance observability. For example, MorFuzz inserts instructions to read the floating-point exception flag CSR after the floating-point instruction sequence to check whether the exception flag is set correctly.

Magic Instruction Generation. MorFuzz instruments magic instructions in the prologue of each testing block as the processor state runtime mutation primitives. The magic instructions are the load instructions that access a random number generator mounted in the test harness. During the simulation, the DUT can atomically randomize the general-purpose registers by accessing the generator. The DUT can specify the generated data type by accessing different address offsets of the generator, including integers, floating-point numbers, addresses, page table entries, etc. The random number generator can generate not only random numbers but also particular corner values (e.g., illegal addresses, maximum and minimum in integers, INF and NaN in floating-point numbers). This significantly improves the possibility of covering corner cases and increases fuzzing stress.

Instruction Shuffle. To further increase the sequence level randomness, we also perform a randomized perturbation of the order of all instructions in the fuzzing payload at the end of the generation, called instruction shuffle. Although some watchpoints will be sacrificed, shuffling instructions mix up adjacent testing blocks, increasing the diversity of instruction sequences and further producing more processor states.

Execution Environment Packaging. MorFuzz integrates a powerful fuzzing execution environment into the stimulus template. First, the fuzzing execution environment is responsible for setting up the execution environment, such as initializing general-purpose registers and memory, configuring address translation mode, and switching to the target privilege level. Second, morphed instructions inevitably trigger exceptions, so the fuzzing execution environment is required to be able to handle the exceptions to avoid crashing the execution. And third, the fuzzer manages the control flow of the stimulus template through the fuzzing execution environment. After executing the scheduled fuzzing payload, the fuzzing execution environment communicates with the fuzzer, and the fuzzer decides whether to continue the simulation based on the reported coverage.

3.4 Runtime Instruction Morphing

When the hardware simulation begins, MorFuzz uses instruction morphing to morph the template instructions to generate diverse and meaningful instruction streams. To mutate the instruction being executed, MorFuzz inserted a morpher into the DUT. The morpher is a logic block inserted in the circuit, which does not affect the processor's functionality. Typically, it is placed on the wire that connects the processor fetch unit and the decode unit. Figure 5 illustrates the workflow of the morpher. First, the morpher hijacks the instruction fetched from memory. Next, the morpher decodes the instruction and performs field level mutation on morphable opcode related fields. And then, the morpher uses the contextual information to generate operand related fields with good semantics. Lastly, the morphed instruction is sent back to the decoder unit. From the view of the processor, the instruction fetched from memory magically turns into another different instruction.

Field Level Mutation. Field level mutation is a structured binary mutation approach that ensures the mutated instructions remain a valid format. To avoid subsequent mutations from destroying the structure of the instruction, the morpher chooses to generate an instruction similar to the hijacked template instruction instead of producing a completely different instruction. Therefore, the morpher does not mutate the fields that determine the instruction format and length, e.g., the `opcode` field. When a new template instruction arrives, the morpher first decodes the instructions and determines which fields are morphable. Next, the morpher randomly selects valid opcodes defined in the specification to replace the opcode related fields in the morphable fields. For instance, in Figure 5, by morphing the `funct3` field, the morpher mutates an `addi` instruction into a `xori` instruction. And finally, the morpher passes the half-finished instruction and the list of morphable fields to the subsequent mutation process.

Semantic Level Mutation. To make the morphed instructions close to real-world usage scenarios, the morpher not only generates the operand related fields randomly but also combines contextual information to mutate the semantics.

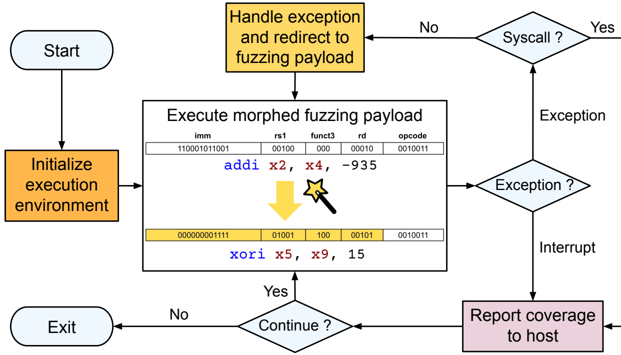


Figure 6: Stimulus template runtime workflow.

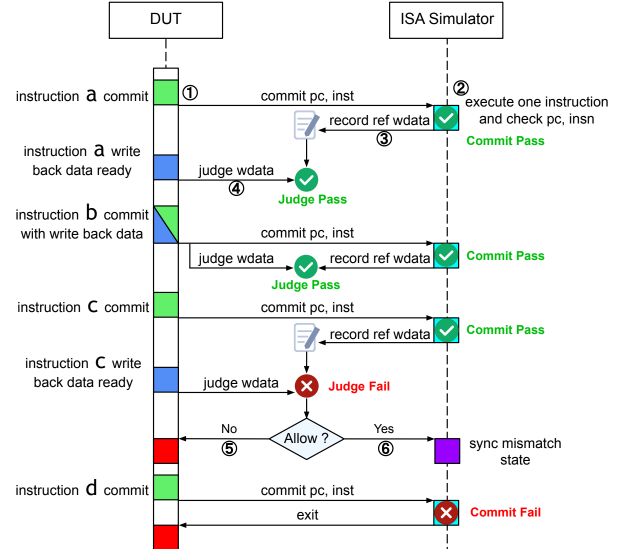


Figure 7: MorFuzz state verification flow.

First, the morpher generates valid address offsets based on the current program counter and address space. During the simulation, the morpher senses DUT's address translation mode and maps the memory layout in the stimulus template into the current address space. When generating an immediate number related to the address (e.g., the `imm` field of the branch instructions), the morpher calculates address offset based on the current program counter and mapped target address, thus ensuring that the morphed immediate number is meaningful.

Second, the morpher maintains a type pool of general-purpose registers to provide operands with the desired type. For memory load and store instructions, they require that the base address register field `rs1` must point to a register containing an address. In order to generate meaningful `rs` fields, the morpher keeps track of the data types (including address and general data) in the general-purpose registers to provide the correct operand type. To simplify type tracing, MorFuzz does not trace the data flow but only marks the type of the destination register as its obtained data type when executing a magic instruction. If another normal instruction writes that destination register again, the register will lose its type.

Third, the morpher uses a sliding window to record the destination register field `rd` of instructions still being executed in the pipeline. The morpher can use the registers in the sliding window as the `rs` and `rd` fields for subsequent template instructions to generate instructions containing pipeline hazards, such as read-after-write and write-after-write. Therefore, MorFuzz is also able to generate inputs matching the microarchitectural details of the DUT spontaneously.

Notice that the morpher would still try illegal cases with a small probability because the input space out of the specification is also a significant source of bugs, such as the illegal opcodes bugs B1 and the illegal operands bugs B4.

Diverse and Meaningful Instruction Streams. Finally, with the help of the stimulus template, MorFuzz morphs template instructions to produce diverse and meaningful instruction streams on the fly. As Figure 6 shown, the DUT starts execution from the initialization function in the fuzzing execution environment and jumps to the fuzzing payload. While executing the morphed instructions, if the DUT triggers an exception, the exception handler in the fuzzing execution environment will try to handle the exception. Whether or not the handler successfully handles the exception, the handler redirects the DUT back to the fuzzing payload. A unique system call is triggered when the DUT reaches the boundary of the fuzzing payload, notifying the fuzzer to collect the current coverage and fix the program counter. By evaluating the coverage, if the fuzzer is interested in the input, it controls the DUT to return to the fuzzing payload again. Otherwise, the fuzzer will terminate the simulation and generate a new stimulus template. In addition, to avoid the DUT from falling into dead loops, MorFuzz also monitors the coverage. If the coverage does not increase for a period of time, the fuzzer will raise an interrupt to stop the simulation. In summary, the fuzzer can control the DUT to continuously execute diverse and meaningful instruction streams in a loop without additional initialization, thus significantly improving the fuzzing performance.

3.5 Synchronizable Co-simulation

MorFuzz applies an online co-simulation approach for state verification, using an ISA simulator running in parallel with the DUT as the reference model. The ISA simulator and the DUT execute the same inputs, so the correctness of the DUT's state can be checked by comparing their states after each instruction is executed.

Compatible Co-simulation. The existing work has assumed that the write-back data is always ready when the DUT commits instructions. However, this assumption is not always true due to the microarchitectural differences between processors. For example, Rocket [1] supports delayed write-back, which means the write-back data of long-latency instructions (e.g.,

multiply and divide, floating-point instructions) may not be ready at the commit stage.

To accommodate different microarchitectures, MorFuzz abstracts the state comparison process into two stages, the commitment stage and the judgment stage. More specifically, we use the instruction a-d in Figure 7 as an example. Assuming that the write-back data is not ready when the DUT commits the instruction a. In the commitment stage, the DUT first commits its program counter and the executed instruction (①). Once the simulator receives the commit request, it executes the next instruction and then checks if the executed instruction is consistent with the one committed by the DUT (②). If the check passes, the simulator records its reference write-back data to the scoreboard (③). The judgment stage starts after the write-back data of the instruction a is ready. MorFuzz compares the write-back value with the reference value in the scoreboard to determine whether the instruction is executed correctly (④). The instruction b shows the case where the commit stage and the judgment stage fire simultaneously, MorFuzz is compatible with this case. When mismatched behavior is detected, MorFuzz reports the potential bug and exits the simulation (e.g., instruction d, c-⑤).

State Synchronization. As discussed in §2.3, not all mismatched differences are bugs. According to our statistics in Figure 10, these implementation differences are triggered with high probability. Although MorFuzz can stop the simulation in time at the mismatched instruction through online state verification, exiting the simulation means that the DUT loses the currently accumulated state, making it difficult for the fuzzer to penetrate the deep states of the processor. Therefore, MorFuzz proposes a synchronizable co-simulation approach to automatically eliminate implementation differences, allowing the DUT to synchronize its state to the reference model to sustain the simulation. For instance, suppose the instruction c accesses a peripheral register, and the judgment stage fails because the simulator lacks a corresponding peripheral simulation. MorFuzz can determine whether the mismatched difference is legal by analyzing the accessed physical address on the simulator. If legal, MorFuzz synchronizes the hardware state to the simulator (⑥) and otherwise reports it as a potential bug (⑤). In addition, MorFuzz can also synchronize external events, such as interrupts, to the simulator. By automatically synchronizing mismatched states, MorFuzz allows the simulation to execute deeper rather than stopping prematurely due to false positives.

4 Implementation

In this section, we discuss several relevant implementation details of MorFuzz. We first describe the stimulus template generator, followed by the fuzzing framework for processor simulation and verification. The prototype we implemented is based on the RISC-V 64-bit architecture.

4.1 Stimulus Template Generation

The stimulus template generator consists of 2.6K lines of python code and 1.3K lines of assembly and C code. We define a 128-bit seed to generate the stimulus template. The seeds determine the fuzzing execution environment and instruction extensions to be tested, control the weight of different testing blocks, seed the random number generator, and set the intensity of the instruction shuffle.

Testing Block. MorFuzz generates different types of testing blocks based on the weights in the seed. The higher the weight of the testing block in the seed, the more likely it is to be generated. We have designed seven types of testing blocks to cover various hardware functional modules of the DUT, including integer arithmetic test, floating-point arithmetic test, CSR test, memory operation test, atomic memory operation test, system operation test, and custom extension test. And control transfer instruction is placed at the end of each testing block to chain them together.

Fuzzing Execution Environment. We extended the testing environment provided by the official RISC-V testing repository [54] as the fuzzing execution environment. The fuzzing execution environment initializes the processor and configures the environment, such as the available instruction extensions, the address translation mode and page table, and the runtime privilege level. During the simulation, the fuzzing execution environment is placed in a non-morphable physical area and is responsible for handling exceptions and interrupts with the highest privilege level. In addition to managing the execution environment, the fuzzing execution environment also provides interfaces to fuzz the system environment, e.g., we provide a series of page table randomization functions to mutate page table entries and evict mapped pages.

4.2 Processor Fuzzing

We use the starship SoC generator [57] to generate the test harness for the processor under test, including the on-chip interconnect system and the memory model that saves the compiled stimulus template. The hardware test harness of the DUT is implemented using about 2K lines of Chisel and 500 lines of Verilog. And we extract the core logic of the official RISC-V ISA simulator, the spike [53], as the reference model to check the correctness of the DUT's behavior. We use about 2.5K lines of C++ code to complete the morpher and the co-simulation framework.

Instruction Morphing. The morpher is implemented as software logic embedded in hardware. It uses the Verilog DPI interface to interact with the hardware, i.e., monitor the processor's internal state, hijack fetched instructions, and return morphed instructions. The morpher performs field-aware mutation on fetched instructions and only replaces the wires between the fetch unit and the decode unit, which ensures that the morphed instructions keep the instruction fetch offset

consistency with the pipeline front-end and does not require modification of the pipeline back-end. Therefore the morpher does not introduce unwanted effects.

In addition, to ensure that the reference model can perform the same morphing as the DUT, the morpher maintains a morphing map, using the instruction before morphing and its address as the key and the morphed instruction as the value. Thus instruction morphing does not introduce false positives, and both models are always able to execute deterministic and identical morphed instructions.

Synchronization Prerequisite. We have strictly defined the rules to approve state synchronization. A difference must meet the following three prerequisites to be considered a legal difference. First, only instructions involving operations beyond the verification scope are allowed to perform subsequent steps. This limits the types of instructions that are allowed to trigger state synchronization to CSR instructions and memory operation instructions. Second, the control flow information of the DUT must pass the commitment stage check. If the DUT incorrectly approves access to privileged registers or reserved address space, an exception will be thrown on the simulator side due to insufficient permissions. MorFuzz will prevent synchronization after observing a program counter violation during the commitment stage. Third, mismatched write-back values are limited to the CSR WARL fields defined in the specification or the data reading from peripheral addresses outside the specification. With further fine-grained checks, MorFuzz can ensure that all synchronized differences are out of our verification scope.

Hardware Simulation. We use an industry-standard commercial tool, Synopsys VCS [14], to simulate hardware RTL designs, but MorFuzz does not rely on features that are exclusive to commercial tools. All hardware modules are translated to Verilog code and then compiled into a host executable binary through the Synopsys VCS RTL simulator.

Hardware Coverage Matrix. MorFuzz is compatible with the coverage matrices proposed by existing designs, and we use the same control register coverage to facilitate comparison with DifuzzRTL [30]. The control register is the register whose value is used for any multiplexers' select signal. We implemented the same FIRRTL [31] pass to instrument all the control registers. The instrumented circuits count the different states triggered in the module and sum up the count as the final coverage. The control register coverage is clock-sensitive and reflects the hardware state better than other coverage matrices. Note that the coverage is only used to evaluate the effect of inputs and mutations, and achieving high coverage in the DUT does not mean that the design is bug-free.

5 Evaluation

In this section, we evaluate the effectiveness of MorFuzz in various aspects. In summary, we aim to answer the following four questions:

- **RQ 1.** How effective is MorFuzz in discovering previously unknown bugs in real-world processors? (§5.2).
- **RQ 2.** How does MorFuzz perform compared with previous methods in exploring the states of processors? (§5.3)
- **RQ 3.** Are the instructions generated by instruction morphing valid and diverse? (§5.4)
- **RQ 4.** How do the instruction morphing and the state synchronization contribute to the effectiveness of our fuzzer? (§5.5)

5.1 Experimental Setup

We conducted the experiments on a 48-core Intel Xeon Silver 4214 processor with 256GB RAM. We ran each experiment for 24 hours and repeated the experiment five times. We fuzzed three popular processors in the RISC-V community to demonstrate that MorFuzz is compatible with different RISC-V microarchitectures. All processors are capable of booting and running Linux, and the configurations of each processor are summarized in Table 1.

CVA6 [68] is an open source 64-bit in-order RISC-V processor core written in SystemVerilog. Although its six-stage pipeline is single-issue, it has independent internal execution functional units. Thus it is able to commit multiple instructions simultaneously. It also has been taped out in 22nm technology and runs at up to 1.7GHz.

Rocket [1] is a five-stage, single-issue, in-order scalar processor written in Chisel [2]. Rocket's pipeline is ingeniously designed to support the delayed write-back, allowing the processor to commit long latency instructions without carrying write-back data. Rocket is the world's first RISC-V processor open sourced by UC Berkeley and is still actively supporting new extensions (e.g., hypervisor [50] and cryptography [49]). Moreover, it has been taped out dozens of times and extensively verified by academia and industrial groups.

BOOM [70] is the third generation of the Berkeley Out-of-Order Machine (BOOM). It is an out-of-order superscalar processor also written in Chisel. Unlike the above in-order cores, BOOM has a more sophisticated microarchitecture, and we used the triple-issue LargeBoom configuration for the experiment. The latest BOOM has been verified on FPGA and achieves better performance than its predecessor.

Table 1: Summary of the cores used for evaluation.

Feature	CVA6	Rocket	BOOM
ISA	RV64GC	RV64GCHX	RV64GCX
Pipeline Stage	6	5	10
Issue Order	In-order	In-order	Out-of-order
Lines of code	24K	99K	339K

5.2 Bugs Found in Real-World Processors

During the evaluation, MorFuzz found 17 new bugs and two already known bugs in total. Our results demonstrate that MorFuzz is capable of finding unknown bugs that are ignored by previous extensive verification conducted by both academia and industrial groups. Moreover, we take responsible disclosure. We report all bugs found to the community using the suggested channels and assist the developers in fixing 9 of them. We also apply CVE identifiers for all newly discovered bugs, and 13 bugs are assigned with CVE numbers. Since MorFuzz does not explicitly target security property violations, direct exploitation of most discovered bugs is to launch denial-of-service attacks. For example, bug B10 prevents the processor from executing crafted instructions correctly, the wrong type generated by bug B13 makes the kernel fail to handle exceptions properly and triggering bug B18 shuts down the system. In general, it is difficult to evaluate the exact security impacts of functional bugs without real-world exploitation scenarios. Recent attacks have shown that even faulty computation results can compromise security isolation [5, 34, 43–45]. We list all the bugs found by MorFuzz in Table 2 together with their corresponding common weakness enumerations (CWEs) to show their potential security implications.

We also compare the average bug reproduction time in Table 3. We select similar bugs reported by previous work to highlight the efficiency improvement over the previous processor fuzzer and instruction generator. In the case of bug B7, MorFuzz triggers the problem significantly faster than riscv-torture and DifuzzRTL. Additionally, since we included binary-level mutations, we may effectively trigger bugs that previous methods failed to cover, such as B8.

Next, we describe in detail the bugs found by MorFuzz. Depending on the complexity, these bugs can be classified into three categories: instruction decoder related, CSR state related, and complex logic bugs. We identify these bugs with the latest RISC-V ISA specification [62, 63].

5.2.1 Instruction Decoder Related Bugs

The first category of bugs is decoder bugs caused by the rare corner format of a single instruction. Previous fuzzers mutate inputs at the instruction level, only generating assembly instructions with valid formats. MorFuzz performs binary-level field-aware mutation, enabling more efficient exploration of unexpected instruction formats.

Bug B1. According to the specification, the `rcon` field of `aes64ks1i` should not greater than `0xA`. When executing an `aes64ks1i` with `rcon` field greater than `0xA`, Rocket does not throw an illegal instruction exception.

Bug B6. By setting the `rm` field in the floating-point instruction, programmers can specify the rounding mode. BOOM can execute floating-point instructions with illegal `rm` fields (such as 5 or 6) without raising exceptions.

Bug B8. In the specification, `sfence.vma` has a zero `rd` field. CVA6 considers illegal `sfence.vma` is valid when its `rd` field is mutated to a non-zero value.

Bug B9. The CVA6 decoder behaves incorrectly when executing `dret` with a non-zero `rd` field, which should be zero according to the specification. CVA6 handles this invalid `dret` as if it were a legal `dret`.

Bug B10. For forward compatibility, implementations must ignore `rd` fields in `fence.i`/`fence`, and standard software must clear them. When executing a non-standard `fence.i`/`fence` with a non-zero rd field, CVA6 throws an exception.

5.2.2 CSR State Related Bugs

CSR state bugs require first setting the CSR to a specific state and then inducing the buggy behavior through instruction sequences. MorFuzz is able to generate instruction sequences that meet the above requirements with the guidance of sequence patterns.

Bug B2. In Rocket, the custom extension illegal signal incorrectly uses vector extension status. Due to this bug, the valid custom extension instruction may fail to execute.

Bug B3. `vsstatus.xs` field is writable in Rocket. The `xs` field summarizes the extension context status, and according to the specification, it is read-only.

Bug B7. If we set the `frm` to DYN (or an invalid value), any floating-point instruction whose `rm` field is set to DYN should raise an illegal instruction exception. Nonetheless, BOOM executes these instructions without raising an exception.

Bug B11. When the `mstatus.fs` field is set to dirty, the `mstatus.sd` field in CVA6 does not update immediately. This bug may cause the contents of the floating-point registers to be lost during the context switch.

Bug B12. CVA6 writes the binary instruction of the `ebreak` to the `mtval/stval` register when it executes an `ebreak`. According to the specification, `mtval/stval` should contain the faulting virtual address if it is written with a non-zero value when a breakpoint exception occurs. And the `ecall` also for the same reason.

Bug B18. Spike's `mcontrol.action` component contains an incorrect mask, which is `0x3f`, while this field only has 4 bits width. If users attempt to set the `sizelo` field next to it, an illegal action will be saved, forcing the program simulation to crash abruptly.

5.2.3 Complex Logic Bugs

The remaining bugs are not concentrated in specific hardware functional modules and require numerous instructions with specific semantics to prepare a buggy environment, we collectively call them logic bugs. MorFuzz monitors the internal runtime states of the DUT to dynamically morph instructions and randomize operands, greatly enhancing the semantics of

Table 2: A list of bugs discovered by MorFuzz.

Processor	Bug Description	CVE/Issue ID	CWE	New Bug	Confirmed	Fixed
Rocket	**B1**: Treat `aes64ksli` with rcon greater than `0xA` as valid	CVE-2022-34632	CWE-327	✓	✓	✓
	B2: Error in condition of the `rocc_illegal` signal	Issue #2980	CWE-1281	✓	✓	✓
	B3: The `vsstatus.xs` is writable	CVE-2022-34627	CWE-732	✓	✓	✓
BOOM	**B4**: Incorrect exception type when a PMA violation	CVE-2022-34636	CWE-1202	✓	✓	
	B5: Incorrect exception type when a PMP violation	CVE-2022-34641	CWE-1198	✓	✓	
	B6: Floating-point instruction with invalid `rm` field does not raise exception	Issue #458	CWE-391		✓	
	B7: Floating-point instruction with invalid `frm` does not raise exception	Issue #492	CWE-391		✓	
CVA6	**B8**: Crafted or incorrectly formatted `sfence.vma` instructions are executed	CVE-2022-34633	CWE-1242	✓	✓	✓
	B9: Crafted or incorrectly formatted `dret` instructions are executed	CVE-2022-34634	CWE-1242	✓	✓	✓
	B10: Non-standard `fence` instructions are treated as illegal	CVE-2022-34639	CWE-1209	✓	✓	✓
	B11: The `mstatus.sd` field does not update immediately	CVE-2022-34635	CWE-1199	✓	✓	
	B12: The value of `mtval`/`stval` after `ecall`/`ebreak` is incorrect	CVE-2022-34640	CWE-755	✓	✓	
	B13: Incorrect exception type when a PMA violation	CVE-2022-34636	CWE-1202	✓	✓	
	B14: Incorrect exception type when a PMP violation	CVE-2022-34641	CWE-1198	✓	✓	✓
	B15: Incorrect exception type when accessing an illegal virtual address	CVE-2022-34637	CWE-754	✓	✓	
	B16: Improper physical PC truncate	Issue #901	CWE-222	✓	✓	
	B17: Incorrect `lr` exception type	CVE-2022-37182	CWE-754	✓	✓	
Spike	**B18**: The component `mcontrol.action` contains the incorrect mask	CVE-2022-34642	CWE-787	✓	✓	✓
	B19: Incorrect exception priotrity when accessing memory	CVE-2022-34643	CWE-754	✓	✓	✓

Table 3: Comparison of the average time to reproduce bugs.

Bug ID	Elapsed Time		
	riscv-torture	DifuzzRTL	MorFuzz
B7	118h	20.3h	10.4m
B8	✗	✗	6.5s

✗ means failed to reproduce the bug.

the input. For complex operations that are difficult to generate randomly, like modifying the page table, MorFuzz can use `ecall` with specific parameters to invoke the page table randomization function in the fuzzing execution environment.

Bug B4, B13. MorFuzz found that the exception type of the physical memory attribute (PMA) violation during the address translation is incorrect. The processor needs to raise an access-fault exception corresponding to the original access type if accessing PTE violates a PMA check. Both the exception types for BOOM and CVA6 are incorrect.

Bug B5, B14. We perform a store operation with a special virtual address whose non-leaf PTE is out of the physical memory protection (PMP) range. BOOM and CVA6 implement the incorrect exception type when a PMP violation occurs.

Bug B15. Bits 63 to 39 of 64-bit virtual addresses in Sv39 must all equal bit 38. When accessing an address that does not satisfy this requirement, CVA6 throws an access fault, while according to the specification, it should be a page fault.

Bug B16. In CVA6, an implicit address truncation is applied to any physical address access. Specifically, the highest 8 bits for instruction addresses and the highest 32 bits for data addresses are ignored.

Bug B17. The exception type of failed `lr` instruction is incorrect in CVA6. When we use `lr` to access a page that has not yet been mapped, CVA6 throws a store page fault.

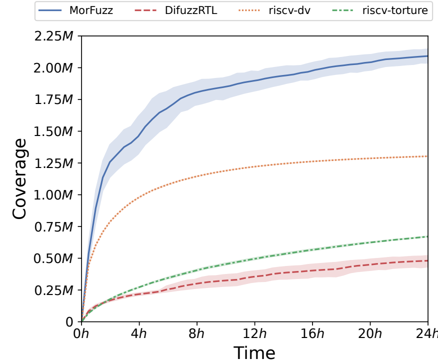


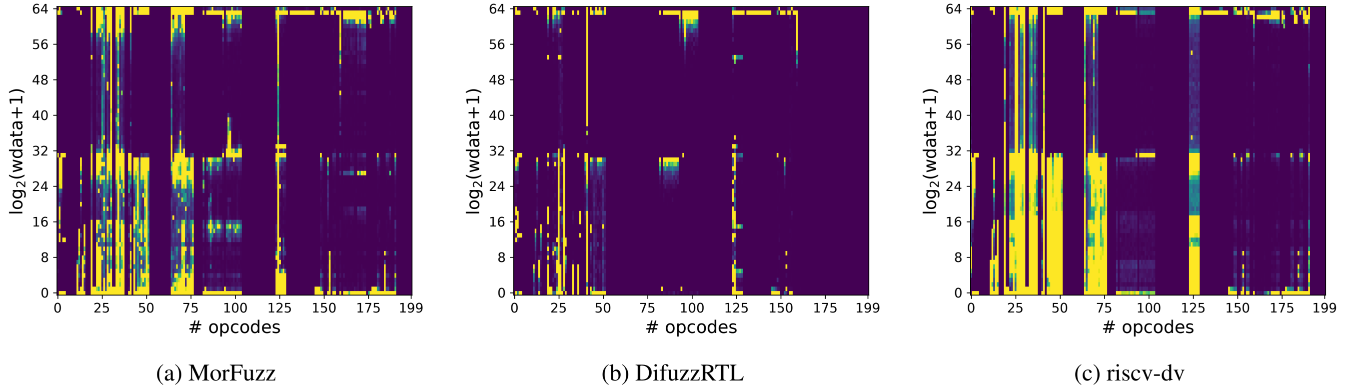
Figure 8: State coverage of MorFuzz and prior works on Rocket over the 24-hour fuzzing time. The shaded area represents the 95% confidence interval.

Bug B19. Spike implements the incorrect exception priority when accessing memory. In the specification, the breakpoint exception has a higher priority than the address-misaligned and access-fault exceptions, which is the opposite of the spike's implementation.

5.3 Exploring the State of Processors

We first demonstrate that MorFuzz can achieve higher state coverage than the state-of-the-art processor fuzzer. We choose the currently available fuzzer, DifuzzRTL[30], for evaluation. We also compare MorFuzz with traditional simulation-based dynamic verification methods. We select riscv-torture [48], a

Since the open-source implementation of DifuzzRTL uses a different RTL simulator, we replay all test cases generated by DifuzzRTL in our environment. In this way, both fuzzers share the same evaluation environment, and the results are comparable.



(a) MorFuzz (b) DifuzzRTL (c) riscv-dv

Figure 9: Instruction diversity of MorFuzz, DifuzzRTL and riscv-dv in one round of 24-hour fuzzing. The highlighted area indicates the number of committed instructions with the corresponding write-back data.

random instruction generator widely used in the community, and use the UVM-based risv-dv [26] to represent industrial solutions. These instruction generators do not use coverage but hand-written constraints to guide random instruction generation. The riscv-torture uses simple register analysis to generate instructions, and the riscv-dv uses manually crafted test templates to generate high-quality test cases. We evaluate the above verification approaches with default configurations on the BaseConfig Rocket because other processors suffer from pending bugs.

Figure 8 presents the evaluation result, which indicates that MorFuzz achieves higher coverage and better efficiency than other methods. Since riscv-torture and riscv-dv use fixed constraints and do not generate random control transfer instructions, they generate deterministic inputs for a specific input space, and therefore their state coverage curves have tighter confidence intervals. MorFuzz and DifuzzRTL have larger fluctuations due to the randomly generated seeds and the control transfer instructions. MorFuzz eventually explores $4.4\times$ more coverage than DifuzzRTL, $3.1\times$ more coverage than riscv-torture, and $1.6\times$ more coverage than riscv-dv. Moreover, MorFuzz is far more efficient than the state-of-the-art processor fuzzer DifuzzRTL. DifuzzRTL takes 24 hours to reach 480K coverage, while MorFuzz obtains the same coverage in only bout 30 minutes. And MorFuzz uses about 2.4 hours to achieve the coverage that riscv-dv takes 24 hours to complete. The remarkable result indicates that MorFuzz can explore processor states effectively and efficiently.

One interesting aspect is that ten years old riscv-torture outperforms DifuzzRTL. To further highlight the impact of input control flow on mutation effectiveness (§2.3), we statistics the test point execution rate of DifuzzRTL. To our surprise, the result is only about 4%. Since DifuzzRTL blindly inserts control flow instructions and lacks exception handlers, most inputs are not completely executed. Such a low execution rate means that DifuzzRTL spends most of its time executing meaningless initialization functions and makes the coverage insufficient to reflect the mutation quality. This may also explain why DifuzzRTL performs better than riscv-torture in the first few hours, but as time grows, the fuzzer is gradually misguided. In contrast, MorFuzz achieves an 86% testing block execution rate with the help of instruction morphing and state synchronization. As a result, MorFuzz is able to execute inputs more thoroughly and mutate them more effectively, enabling efficient exploration of the processor's state.

5.4 Instruction Diversity

To illustrate that instruction morphing generates valid and diverse instruction streams, we visualize each committed valid instruction and its write-back data during the fuzzing. We assess the diversity of instructions in two dimensions: the `opcode` (i.e., the function of the instruction) and the `wdata` (i.e., the result written back). The diversity of `opcode` indicates the number of data paths a fuzzer can test, while the diversity of `wdata` suggests the test completeness for a specific data path. We evaluate the instruction diversity of both MorFuzz, DifuzzRTL and riscv-dv. We plot the result as heat maps (Figure 9), using the `opcode` as the x-axis, the logarithm of the `wdata` as the y-axis, and the brightness as the number of committed instructions corresponding to the `opcode` and `wdata` pair.

Since not all instructions generate the full range of 64-bit write-back data, some areas in the heat map are always dark. For example, the branch and store instructions do not have destination registers, and word operations never generate data larger than 32 bits. Comparing these figures, we find that riscv-dv has the most bright areas (Figure 9c). Under manually crafted constraints, riscv-dv can generate valid instructions with uniformly distributed operands, representing the upper bound on the quality of randomly generated instructions. On the contrary, DifuzzRTL has the least highlighted regions, indicating its limited input diversity (Figure 9b). Figure 9a suggests that MorFuzz is capable of generating more valid

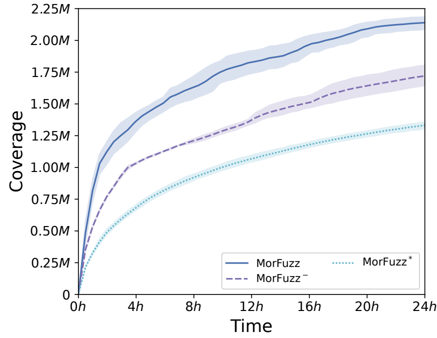


Figure 10: State coverage of MorFuzz and its variants.

and diverse fuzzing inputs than DifuzzRTL, and the quality of the generated instructions is comparable to that of riscv-dv.

5.5 Component Analysis

To measure the effect of each component of MorFuzz, we first create two variants of MorFuzz that disables part of its components: (i) MorFuzz$^-$ disables instruction morphing, which means the DUT directly executes statically generated random instructions without morphing. (ii) MorFuzz* disables both instruction morphing and state synchronization. This variant means the fuzzer not only executes the static instructions but also terminates the simulation immediately when it detects a mismatched state. Similarly, we evaluate MorFuzz and its variants on the BaseConfig Rocket for 24 hours.

Figure 10 shows the results of the experiment. The gap in coverage between the two variants shows that false positives caused by the model implementation differences can cause the DUT to terminate the simulation prematurely and fail to execute the input comprehensively. MorFuzz$^-$ effectively eliminates implementation differences through state synchronization, allowing the fuzzer to touch more deep states. And by comparing MorFuzz and MorFuzz$^-$, MorFuzz significantly increases the speed of coverage growth with instruction morphing. The instruction morphing technique uses run-time contextual information to generate more diverse and meaningful instruction streams, thus dramatically increasing the effectiveness of the fuzzing.

6 Discussion and Limitations

Requirement of ISA Simulator. MorFuzz and other processor fuzzing works [6, 30, 32, 33] use the ISA simulator as the golden reference model to verify the behavior of processors. Usually, there are many available simulators for different instruction set architectures, e.g., Bochs [38], QEMU [3], and Dromajo [8]. One possible problem is that the implementations between different simulators are variable and may lead to false positives. To mitigate this, MorFuzz proposes the synchronizable co-simulation framework to automatically eliminate the implementation differences between the DUT and the simulator.

Restriction on Functional Bug. Currently, MorFuzz can only use the functional model provided by the simulator for state verification, such as memory and registers. For details not available in the simulator, such as caches and branch predictors, MorFuzz cannot directly verify the correctness of their behavior and can only detect bugs when they propagate into the architectural registers.

State Sync vs. False Positive. State synchronization would only eliminate legal differences in the architectural state without modifying implementation details. From the simulator's view, it merely replaces the operands in the general-purpose registers with those of the DUT. If the DUT executes the synchronized instruction again, it will still get a mismatch and then trigger state synchronization to eliminate it. Therefore, state synchronization itself does not introduce false positives. On the contrary, it eliminates false positives caused by implementation differences at the architectural level in time, allowing the fuzzer to explore more deep states.

Complex Bug Pinpoint. First, MorFuzz still requires the user to dive deep into the specification and circuit to analyze the root cause to determine if the mismatch is a false positive caused by implementation differences or an actual bug. Second, diverse instruction streams make MorFuzz more efficient in terms of coverage while also making it difficult for users to pinpoint the bug. Because the morphed instructions and control flow information generated by the morpher do not exist in the stimulus template, users need to save additional runtime information to assist in the analysis.

FPGA Emulations. RFUZZ [37] and DifuzzRTL [30] can use FPGA to accelerate the simulation process by sacrificing verification accuracy. MorFuzz uses the ISA simulator to co-simulate with the DUT to provide more accurate verification. However, the ISA simulator is a software model that cannot be mapped directly onto FPGA, so MorFuzz can only simulate them via the RTL simulator.

7 Related Work

In this section, we describe the existing hardware fuzzing works and introduce how MorFuzz differs from them, as summarized in Table 4.

RFUZZ [37] introduced the concept of hardware fuzzing and first proposed a coverage-guided hardware fuzzing framework for general RTL designs. To match the various interfaces of the targets, RFUZZ generates input directly for the hardware ports at cycle granularity. Unlike RFUZZ, MorFuzz uses a test harness to convert the compiled assembly programs into bus transactions, ensuring that the input's hardware semantics are legal, thereby more efficiently fuzzing processor designs. And several succeeding works [7, 39] extended RFUZZ to improve performance by analyzing circuit information (e.g.,

Table 4: Comparison with the prior hardware fuzzers.

Fuzzer	Fuzzing Target	Coverage Matrix	Mutation Dimension	Verification Preknowledge	Coverage Comparison	Performance Comparison	New Bugs
RFUZZ [37]	RTL designs	Multiplexer	Binary	N/A	Baseline	Baseline	0
DirectFuzz [7]	RTL designs	Multiplexer	Instance distance	N/A	Same to RFUZZ	2.23× faster than RFUZZ	0
Trippel et al. [58]	RTL designs	Software	Custom grammar	SVA	26.70% more than RFUZZ	N/A	0
DifuzzRTL [30]	Processor	Control register	Instruction	Not required	N/A	40× faster than RFUZZ	16
Kabylkas et al. [32]	Processor	N/A	N/A	Not required	N/A	N/A	13
TheHuzz [33]	Processor	Hardware behavior	Instruction field	Not required	2.86% more than DifuzzRTL	3.33× faster than DifuzzRTL	8
MorFuzz	Processor	Control register	Processor state, instruction field, program semantic	Not required	4.4× more than DifuzzRTL	48× faster than DifuzzRTL	17

module distance, symbolic execution) to optimize input. However, these efforts only focus on maximizing the hardware coverage and do not give solutions to verifying hardware behavior, thus they are ineffective in finding bugs.

Trippel et al. [58] use the famous software fuzzer AFL [67] to fuzz the host-executable binary file generated by the RTL simulator. Moreover, the authors use SystemVerilog assertion (SVA) to check design violations. Unfortunately, SVA has the following two drawbacks [24]. First, SVA requires prior manual instrumentation by the developer. Thus it asserts known bugs rather than exploring unknown bugs. Second, SVA cannot constrain the buggy behavior of complex processors well because bugs usually result from multi-cycle actions. SVA has difficulty constraining the behavior of multiple modules over multiple cycles. MorFuzz uses a co-simulation based differential testing approach. By comparing the state differences between the DUT and the reference model after each instruction is executed, MorFuzz can accurately and automatically identify potential bugs without any predefined assertions.

DifuzzRTL [30] and TheHuzz [33] are hardware fuzzing frameworks exclusively for processors and are the most relevant works to MorFuzz. DifuzzRTL proposes a cycle-sensitive control register coverage matrix, and TheHuzz uses features provided by commercial EDA tools to capture more intrinsic hardware behaviors. As opposed to previous efforts that focus on designing fine-grained coverage matrix, MorFuzz aims to verify processors more effectively and efficiently. First, MorFuzz designs the stimulus template to efficiently explore the input space from the processor state, instruction field, and program semantics levels. Second, MorFuzz uses instruction morphing to dynamically mutate the template instructions. By collecting runtime information to generate meaningful instruction streams, MorFuzz significantly improves the effectiveness of fuzzing. Third, MorFuzz uses state synchronization to eliminate the implementation differences between the DUT and the reference model so that the simulation can continue to execute. Thus MorFuzz can penetrate more deep states of the processor. Additionally, MorFuzz does not rely on commercial tools and is also compatible with the traditional verification processes in the semiconductor industry and the open-source community.

Kabylkas et al. [32] introduced the Logic Fuzzer, a small piece of logic injected into the circuit to trigger atypical scenarios. However, we consider that the Logic Fuzzer has a limited effect. First, the bugs triggered by the Logic Fuzzer cannot be reproduced by software. Therefore the Logic Fuzzer may violate the designer's intent, e.g., a properly working Branch Target Buffer will not generate invalid branch addresses. Second, the Logic Fuzzer can only work in specific hardware modules (e.g., FIFO, memory) that do not affect the processor's functionality. MorFuzz increases processor test pressure by injecting the morpher into the decoder, which is a general design, and all reported bugs are software triggerable.

In addition to the above fuzzers for processor RTL code at the pre-silicon stage, there are also some fuzzers designed to detect undocumented instructions [18, 61] and hidden model-specific registers [20, 36] in manufactured processors to disclose the hardware backdoors [19]. Since the custom extensions in different models are not identical, the differential testing based MorFuzz is also able to detect these undocumented hidden features. During our evaluation, we did find some custom instructions [9] and their related bug (Bug B2).

8 Conclusion

This paper proposed MorFuzz, a coverage-guided processor fuzzer that can detect software triggerable hardware bugs efficiently. As opposed to prior fuzzers, MorFuzz uses instruction morphing to dynamically mutate instructions at runtime to generate diverse and meaningful inputs and efficiently guide mutations. In addition, MorFuzz designs stimulus templates to provide multi-level runtime mutation primitives and develops the synchronizable co-simulation framework to eliminate implementation differences. We evaluate MorFuzz on three popular open-source RISC-V processors and achieve at most 4.4× and 1.6× more state coverage than the state-of-the-art fuzzer, DifuzzRTL, and the famous constrained instruction

generator, riscv-dv, respectively. Moreover, MorFuzz discovered a total of 17 new bugs (with 13 CVEs assigned), demonstrating its effectiveness in detecting unknown bugs in real-world processors.

Acknowledgments

We thank all anonymous reviewers and our shepherd for their valuable comments and suggestions, which significantly improved this paper. We also appreciate the developers in the open-source RISC-V processor communities for their helpful responses to our bug reports. They are Andrew Waterman, Jerry Zhao and Jiuyang Liu from Rocket/BOOM, Florian Zaruba and Guillaume Chauvon from CVA6, Scott Johnson and Tim Newsome from Spike. Additionally, we acknowledge Zhiheng He and Zhenxia Mo from Pengcheng Laboratory and Kun Yang from Zhejiang University for sharing their insights on the state of the industry. The authors are partially supported by the National Key R&D Program of China (No. 2022YFE0113200), the National Natural Science Foundation of China (NSFC) under Grant U21A20464, as well as the Research Grants Council (Hong Kong) under Grants RFS2122-1S04, C2004-21GF, R1012-21, and R6021-20F. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

References

[1] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design automation conference 2012*, pages 1212–1221. IEEE, 2012.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. Califor-nia, USA, 2005.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[5] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. {ÆPIC} leak: Architecturally leaking uninitialized data from the microarchitecture. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3917–3934, 2022.

[6] Niklas Bruns, Vladimir Herdt, Daniel Große, and Rolf Drechsler. Efficient cross-level processor verification using coverage-guided fuzzing. In *Proceedings of the Great Lakes Symposium on VLSI 2022*, pages 97–103, 2022.

[7] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 529–534. IEEE, 2021.

[8] Chipsalliance. dromajo. https://github.com/chipsalliance/dromajo.

[9] Chipsalliance. Non-standard opcode 0x30500073 in rocket. https://github.com/chipsalliance/rocket-chip/issues/1868.

[10] AMD Corporation. Revision guide for amd family 10h processors revision 3.92. 2012.

[11] Intel Corporation. Pentium processor specification update. 1999.

[12] Intel Corporation. 7th and 8th generation intel core processor family specification update revision 011. 2019.

[13] Intel Corporation. 12th generation intel core processor specification update revision 008. 2022.

[14] Synopsys Corporation. Chronologic VCS Simulator. https://www.synopsys.com/verification/simulation/vcs.html.

[15] National Vulnerability Database. CVE-2012-0217. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0217.

[16] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. {HardFails}: Insights into {Software-Exploitable} hardware bugs. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 213–230, 2019.

[17] C Domas. The memory sinkhole-unleashing an x86 design flaw allowing universal privilege escalation. *Black-Hat, Las Vegas, USA*, 2015.

[18] Christopher Domas. Breaking the x86 isa. *Black Hat*, 2017.

[19] Christopher Domas. Hardware backdoors in x86 cpus. *Black Hat*, pages 1–14, 2018.

[20] Christopher Domas. The ring 0 façade: Awakening the processors inner demons. *DEF CON*, 2018.

[21] Stephanie Drzevitzky. Proof-carrying hardware: Runtime formal verification for secure dynamic reconfiguration. In *2010 International Conference on Field Programmable Logic and Applications*, pages 255–258. IEEE, 2010.

[22] Shai Fine and Avi Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Proceedings of the 40th annual Design Automation Conference*, pages 286–291, 2003.

[23] Harry Foster. 2020 wilson research group functional verification study: Ic/asic functional verification trend report. *Wilson Research Group and Mentor, A Siemens Business, White Paper*, 2020.

[24] Weimin Fu, Orlando Arias, Yier Jin, and Xiaolong Guo. Fuzzing hardware: Faith or reality? In *2021 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pages 1–6. IEEE, 2021.

[25] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[26] Google. riscv-dv. https://github.com/google/riscv-dv.

[27] Xiaolong Guo, Raj Gautam Dutta, Yier Jin, Farimah Farahmandi, and Prabhat Mishra. Pre-silicon security verification and validation: A formal perspective. In *Proceedings of the 52nd annual design automation conference*, pages 1–6, 2015.

[28] Finn Haedicke, Hoang M Le, Daniel Große, and Rolf Drechsler. Crave: An advanced constrained random verification environment for systemc. In *2012 International Symposium on System on Chip (SoC)*, pages 1–7. IEEE, 2012.

[29] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song, and Insik Shin. Enhancing memory error detection for large-scale applications and fuzz testing. In *Network and Distributed Systems Security (NDSS) Symposium 2018*, 2018.

[30] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, 2021.

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

[32] Nursultan Kabylkas, Tommy Thorn, Shreesha Srinath, Polychronis Xekalakis, and Jose Renau. Effective processor verification with logic fuzzer enhanced co-simulation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 667–678, 2021.

[33] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. Thehuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities.

[34] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0ltpwn: Attacking x86 processor integrity from software. In *Proceedings of the 29th USENIX Conference on Security Symposium*, pages 1445–1461, 2020.

[35] Nathan Kitchen and Andreas Kuehlmann. Stimulus generation for constrained random simulation. In *2007 IEEE/ACM International Conference on Computer-Aided Design*, pages 258–265. IEEE, 2007.

[36] Andreas Kogler, Daniel Weber, Martin Haubenwallner, Moritz Lipp, Daniel Gruss, and Michael Schwarz. Finding and exploiting cpu features using msr templating. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1474–1490. IEEE, 2022.

[37] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: Coverage-directed fuzz testing of rtl on fpgas. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[38] Kevin P Lawton. Bochs: A portable pc emulator for unix/x. *Linux Journal*, 1996(29es):7–es, 1996.

[39] Tun Li, Hongji Zou, Dan Luo, and Wanxia Qu. Symbolic simulation enhanced coverage-directed fuzz testing of rtl design. In *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2021.

[40] Lingyi Liu and Shabha Vasudevan. Star: Generating input vectors for design validation by static analysis of rtl. In *2009 IEEE International High Level Design Validation and Test Workshop*, pages 32–37. IEEE, 2009.

[41] Ashok B Mehta. Constrained random verification (crv). In *ASIC/SoC Functional Design Verification*, pages 65–74. Springer, 2018.

[42] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *Proceeding of the 29th USENIX Security Symposium*, 2020.

[43] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482. IEEE, 2020.

[44] Shisong Qin, Chao Zhang, Kaixiang Chen, and Zheming Li. idev: exploring and exploiting semantic deviations in arm instruction processing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 580–592, 2021.

[45] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaking sgx by software-controlled voltage-induced hardware faults. In *2019 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 1–6. IEEE, 2019.

[46] Jeyavijayan Rajendran, Vivekananda Vedula, and Ramesh Karri. Detecting malicious modifications of data in third-party intellectual property cores. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015.

[47] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[48] UC Berkeley Architecture Research. riscv-torture. https://github.com/ucb-bar/riscv-torture.

[49] RISC-V. riscv-crypto. https://github.com/riscv/riscv-crypto.

[50] Bruno Sá, José Martins, and Sandro Pinto. A first look at risc-v virtualization from an embedded systems perspective. *IEEE Transactions on Computers*, 71(9):2177–2190, 2021.

[51] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. {kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[52] Wilson Snyder. Verilator. https://github.com/verilator/verilator.

[53] RISC-V Software. riscv-isa-sim. https://github.com/riscv-software-src/riscv-isa-sim.

[54] RISC-V Software. riscv-tests. https://github.com/riscv-software-src/riscv-tests.

[55] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *2019 Network and Distributed Systems Security Symposium (NDSS)*, pages 1–15. Internet Society, 2019.

[56] Giovanni Squillero. Microgp—an evolutionary assembly program generator. *Genetic programming and evolvable machines*, 6(3):247–263, 2005.

[57] Sycuricon. Starship soc generator. https://github.com/sycuricon/starship.

[58] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3237–3254. USENIX Association, 2022.

[59] Ilya Wagner, Valeria Bertacco, and Todd Austin. Stresstest: an automatic approach to test generation via activity monitors. In *Proceedings of the 42nd annual Design Automation Conference*, pages 783–788, 2005.

[60] Fanchao Wang, Hanbin Zhu, Pranjay Popli, Yao Xiao, Paul Bodgan, and Shahin Nazarian. Accelerating coverage directed test generation for functional verification: A neural network-based framework. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pages 207–212, 2018.

[61] Guang Wang, Ziyuan Zhu, Shuan Li, Xu Cheng, and Dan Meng. Differential testing of x86 instruction decoders with instruction operand inferring algorithm. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 196–203. IEEE, 2021.

[62] Andrew Waterman, Krste Asanovic, and CS Division. The RISC-V instruction set manual volume I: Unprivileged isa.

[63] Andrew Waterman, Krste Asanovic, John Hauser, and CS Division. The RISC-V instruction set manual volume II: Privileged architecture.

[64] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated

discovery of microarchitectural side channels. In *30th USENIX Security Symposium (USENIX Security'21)*, pages 1–18, 2021.

[65] Stephen Williams. Icarus verilog. https://github.com/steveicarus/iverilog.

[66] Jun Yuan, Carl Pixley, Adnan Aziz, and Ken Albin. A framework for constrained functional verification. In *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No. 03CH37486)*, pages 142–145. IEEE, 2003.

[67] Michal Zalewski. American fuzzy lop. https://lcamtuf.coredump.cx/afl.

[68] F. Zaruba and L. Benini. The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11):2629–2640, Nov 2019.

[69] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. End-to-end automated exploit generation for validating the security of processor designs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 815–827. IEEE, 2018.

[70] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. May 2020.

[71] Yanhong Zhou, Tiancheng Wang, Huawei Li, Tao Lv, and Xiaowei Li. Functional test generation for hard-to-reach states using path constraint solving. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(6):999–1011, 2015.

USENIX'23 Artifact Appendix: MorFuzz: Fuzzing Processor via Runtime Instruction Morphing enhanced Synchronizable Co-simulation

Jinyan Xu
Zhejiang University
phantom@zju.edu.cn

Yiyuan Liu
Zhejiang University
yiyuanliu@zju.edu.cn

Sirui He
City University of Hong Kong
sol.he@my.cityu.edu.hk

Haoran Lin
Zhejiang University
haoran_lin@zju.edu.cn

Yajin Zhou
Zhejiang University
yajin_zhou@zju.edu.cn

Cong Wang
City University of Hong Kong
congwang@cityu.edu.hk

A Artifact Appendix

A.1 Abstract

As introduced in the paper, MorFuzz discovers several new bugs across open-source RISC-V processors with different microarchitectures and significantly improves the efficiency and effectiveness of processor fuzzing. Our artifact provides binaries and scripts to reproduce those results. This appendix describes the steps to set up our prototype and run our evaluation experiments.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact is available at `https://github.com/sycuricon/MorFuzz/releases/tag/usenix23`. Both MorFuzz pre-built binaries and the inputs generated by DifuzzRTL that we replayed are available at `https://zenodo.org/record/8055696`.

A.2.3 Hardware dependencies

MorFuzz uses commercial EDA software, so an x86 processor is required. We evaluate MorFuzz on a 48-core dual Intel Xeon Silver 4214 server with 256GB RAM. In addition, at least 280 GB of storage is required. Storing the input generated by DifuzzRTL requires 270 GB, and the evaluation also consumes about 10 GB of storage.

A.2.4 Software dependencies

Our prototype contains three components: an instruction generator, a co-simulation library, and a top-level fuzzing framework. We release the instruction generator and the co-simulation library of MorFuzz as pre-built binaries, they are compiled with GCC 10.2.1 on CentOS 7.9.2009. In order to run MorFuzz the same operating system and compiler are required. MorFuzz uses the Synopsys VCS, a commercial RTL simulator, to simulate processor designs. You need to purchase licenses from Synopsys to use VCS. In addition, in order to cross-compile RISC-V programs, a RISC-V toolchain is also required, which is available at the official riscv-gnu-toolchain repository on the GitHub.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

First, MorFuzz requires the following dependencies, and in order to run the experiment you also need to set up dependencies from riscv-dv and riscv-torture:

```
sudo yum -y groupinstall "Development Tools"
sudo yum -y install redhat-lsb libXScrnSaver
    centos-release-scl dtc
sudo yum -y install devtoolset-10
```

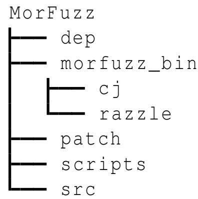
Second, clone the repository and execute the setup script.

```
git clone https://github.com/sycuricon/MorFuzz.git
cd MorFuzz
git checkout usenix23
git submodule update --init --recursive
export ARTIFACT_ROOT=$(pwd)
./scripts/setup.sh
```

Next, download the MorFuzz pre-built binaries `morfuzz_bin.zip` from `https://zenodo.org/record/8055696` and unzip it. You also need to place the decompressed `morfuzz_bin` directory under the root directory of the MorFuzz repository.

Finally, download the input sets `difuzzrtl_[0-4].zip` generated by DifuzzRTL from https://zenodo.org/record/8055696 and unzip them. You do not need to copy them into the repository, making the `DIFUZZRTL_INPUT` environment variable point to one of the input sets is enough.

The final directory structure of the project is as follows:



A.3.2 Basic Test

Before executing each experiment, you need to point the `ARTIFACT_ROOT` environment variable to the directory where the MorFuzz repository was cloned and execute the `env.sh` script to set up the other environment variables.

```
export ARTIFACT_ROOT=#absolute path to MorFuzz#
cd $ARTIFACT_ROOT
source ./scripts/env.sh
```

After executing the script if there are no complaints about missing dependencies, you can execute the basic test script. The script invokes Rocket, BOOM, CVA6, and Spike in turn to execute a normal test case, and if the test passes the "`*** PASSED ***`" message will appear on the terminal.

```
./scripts/basic.sh
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): MorFuzz is compatible with different microarchitectures and identified new bugs. This claim is supported by experiment (E1).

(C2): MorFuzz can efficiently achieve better coverage than DifuzzRTL and other techniques. This claim is supported by experiment (E2).

(C3): MorFuzz is capable of generating more diverse inputs than DifuzzRTL, and is comparable to riscv-dv. This claim is supported by experiment (E3).

(C4): Instruction morphing and state synchronization can help MorFuzz achieve better coverage. This claim is supported by experiment (E4).

A.4.2 Experiments

(E1): Executing test cases that trigger discovered bugs on the corresponding processors to prove that MorFuzz can be used on different microarchitectures, for details see `src/table2/README.md`.

Estimated time: less than 5 human-minute, and less than 5 compute-minute.

Preparation: Go to the MorFuzz repository root directory, set up `ARTIFACT_ROOT` variable and execute `scripts/env.sh`.

Execution: Execute `scripts/tab2.sh`.

Results: Trigger bugs in Table 2. For a detailed analysis of each result, please refer to `src/table2/README.md`.

(E2): Evaluating coverage to prove that MorFuzz achieves better coverage, for details see `src/figure8/README.md`.

Estimated time: less than 5 human-minute, and 24 compute-hour.

Preparation: Go to the MorFuzz repository root directory, set up `ARTIFACT_ROOT` and `DIFUZZRTL_INPUT` variables and execute `scripts/env.sh`.

Execution: Execute `scripts/fig8.sh`.

Results: Reproduce Figure 8, you can find the figure at `scripts/output/fig8.pdf`.

(E3): Evaluating instruction diversity to prove that MorFuzz generates instructions with good diversity, for details see `src/figure9/README.md`.

Estimated time: less than 5 human-minute, and 24 compute-hour.

Preparation: Go to the MorFuzz repository root directory, set up `ARTIFACT_ROOT` and `DIFUZZRTL_INPUT` variables, and execute `scripts/env.sh`.

Execution: Execute `scripts/fig9.sh`.

Results: Reproduce Figure 9, you can find three heatmaps named `heatmap_<name>.pdf` in the `scripts/output` directory.

(E4): Evaluating coverage to prove that MorFuzz's subcomponents can help MorFuzz achieve better coverage, for details see `src/figure10/README.md`.

Estimated time: less than 5 human-minute, and 24 compute-hour.

Preparation: Go to the MorFuzz repository root directory, set up `ARTIFACT_ROOT` variable and execute `scripts/env.sh`.

Execution: Execute `scripts/fig10.sh`.

Results: Reproduce Figure 10, you can find the figure at `scripts/output/fig10.pdf`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.