Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Generating high-coverage test suites for RISC-V

Author:
Yan Hoi Puk

Supervisor:
Prof. Alastair Donaldson

Second Marker:
Prof. Cristian Cadar

June 21, 2023

Abstract

RISC-V is an open standard instruction set architecture that has gained attraction from developers and manufacturers recently. Early verification of RISC-V implementations is crucial since hidden bugs would become harder to recover later in the development cycle. It incentivizes advanced techniques for generating massive test cases. Therefore, this project aims to take inspiration from coverage-guided fuzzing and fuzzing by proxy to generate a high-coverage test suite that can expose defects in the RISC-V implementations.

In this project, we have successfully developed a RISC-V fuzzer with Spike by first making Spike fuzz-friendly and integrating it with libFuzzer. Then, we created two variations of the RISC-V custom mutator that perform type-aware mutations. Lastly, we performed fuzzing by proxy with the fuzzer-generated test suite to test three other RISC-V implementations: RVEMU, FORVIS and SAIL-RISCV.

We have provided a quantitative analysis of the performance of different variations of fuzzer. The fuzzer with the advanced custom mutator outperforms other variations in most areas. In addition, we have compared the code coverage of Spike with the official RISC-V ISA test suite. The result shows our test suite could provide unique coverage of Spike that could lead to 13.4% increase in total branch coverage and 10% increase in total line coverage. We have also discovered a floating point instruction bug in RVEMU.

Acknowledgements

I would like to thank my supervisor, Prof. Alastair Donaldson, for his constant support throughout the ups and downs of the project. I have learned a lot from you and it has been a pleasure to work with you on this project.

I would also like to thank my parents and my sponsor, Mr. William Louey for their continuous support throughout the degree. I would not have made it this far without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

RISC-V is an open standard Instruction Set Architecture (ISA). It is the fifth generation of Reduced Instruction Set Computer (RISC) architecture developed at the University of California, Berkeley. Due to its open, royalty-free nature, it has gained attraction from developers and manufacturers. A few companies such as SiFive [2] start to offer next-generation RISC-V CPU IPs to their customers.

In the development of ISA, verification is the important process of checking whether the ISA and its implementation meet the design requirements and specifications. Some common verification methods include formal methods and simulation. Formal methods are mathematical techniques that can prove or disprove properties of the ISA, such as correctness or security. Simulation mimics the behaviour of the ISA on a software platform. While formal methods provide high confidence and coverage, it requires high expertise and manual effort. In contrast, simulation provides high objectivity and repeatability but may suffer from low representativeness and diversity [3].

Same as in the development of other ISA implementations, early verification of RISC-V implementations is crucial since hidden bugs would become harder and more costly to recover later in the development cycle. It incentivizes advanced techniques for generating massive test cases: usually binary programs with expected results to ensure the correctness of the implementation. By inputting these test cases into the models or simulators, the defects contained in the implementations could be exposed by comparing the outputs with the expected results. Yet, the manual effort of generating the huge test cases would also significantly increase.

On the other hand, coverage-guided fuzzing has shown great success in finding subtle, security-critical bugs in system-level software [4] while generating a large sample of test inputs. In the context of testing ISA implementations, coverage-guided fuzzing could naturally and randomly produce interesting and diverse test binaries. Meanwhile, it may discover unexpected input binary that can cause crashes which are difficult to be picked up by manually-created tests.

However, coverage-guided fuzzing could be hard to apply to implementation with slow execution speed or implementation that is difficult to collect any coverage information. To take this idea further, we could first perform fuzz testing on an implementation which could be fuzzed at speed. Afterwards, using all the test binary collected during fuzzing, we could fuzz test other implementations 'by proxy' as long as they accept inputs of the same format. Moreover, we could also apply differential testing by cross-checking the outputs of different implementations.

This project aims to take inspiration from coverage-guided fuzzing and fuzzing by proxy to generate a high-coverage test suite that can effectively expose the defects in the RISC-V implementations.

1.2 Objectives & Challenges

The main aim of this project is to create RISC-V fuzzers with Spike, the official RISC-V ISA simulator [5] to generate test cases. Then, we use the generated test cases to perform fuzzing by proxy with implementations that accept RISC-V binary as input. In this project, the following implementations are chosen:

- RVEMU, a RISC-V emulator, written in Rust [6]
- FORVIS, a formal RISC-V ISA Specification, written in Haskell [7]
- SAIL-RISCV, a formal specification of the RISC-V architecture, written in Sail [8]

As the side product of this project, we would be looking for bugs and crashes of Spike and other implementations. Moreover, we would also like to evaluate the effectiveness of structure-aware fuzzing on RISC-V ISA implementation by comparing the performance of different versions of fuzzer. Lastly, any incoherence of final register states of implementations in fuzzing by proxy would be further examined and reported if the behaviour deviates from the specification.

The main challenges involved in this project:

- **Extracting Spike APIs and integrating with libFuzzer**
  Because libFuzzer is an in-process fuzzer, an entry point for the program is required instead of executing the program directly. Without clear instructions or internal knowledge, locating the appropriate API and properly setup the configuration of the simulation could be a difficult task. Moreover, the integration process with libFuzzer could be problematic that prevents it to be fuzzed efficiently. [Chapter 3]
- **Creating RISC-V custom mutators**
  The custom mutator for RISC-V needs to ensure the mutated instructions are still semantically correct. However, there are six types of instructions in RISC-V ISA, each with different composition of fields, simply mutating the opcode could lead to undefined instructions as the bits could be interpreted differently in another type of instruction. [Chapter 4]
- **Building up fuzzing by proxy and examining its output**
  First of all, we have to set up the environment to build and learn the software for three different languages. Furthermore, its automation could also be challenging as the different forms of output need to be read out and parsed correctly for the fair comparison of the register states while still handling situations when the simulation did not terminate. During the process, we also need to ensure the initial execution state of implementations is the same [Chapter 5]. The examination of mismatches in the final register state also requires proficiency in RISC-V architecture and ISA as not all mismatch is caused by a bug in the implementation.

1.3 Contributions

This project introduces a brand new fuzzer that integrates Spike, the golden reference functional Instruction Set Simulator for RISC-V, with libFuzzer. The fuzzer can generate random valid RISC-V binaries for testing the behaviour of Spike and other RISC-V implementations.

The fuzzer applies the idea of structure-aware fuzzing by utilizing the built-in custom mutator API in libFuzzer. The custom mutator covers the whole RV64G extension which includes around 160 unique instructions. The advanced version ensures the mutated instructions are still valid by first decoding the type of instruction and applying suitable mutation operations.

Furthermore, we have also created a script for automating fuzzing by proxy process. Given saved test cases in the test corpus, the script can translate them to assembly programs and compile them into test binaries that satisfy the requirements of other RISC-V implementations: RVEMU, FORVIS and SAIL-RISCV. In the end, the script would produce human-readable output which details the difference of final register states across implementations.

Lastly, by applying differential testing with fuzzing by proxy, we have discovered and reported a bug regarding the incorrect single precision floating point extension implementation of RVEMU [9]. Moreover, the test suite generated from our advanced fuzzer has achieved similar code coverage in Spike compared to the official RISC-V ISA test suite. [Chapter 6]

Chapter 2

Background

2.1 RISC-V

RISC-V is an open-sourced and royalty-free Instruction Set Architecture introduced in 2010. The most basic RISC-V ISA extension is the base Integer Instruction Set, denoted as RV32I with 32 bits registers [1]. It also extends to 64-bit and 128-bit version denoted as RV64I and RV128I. In this project, we focus on the 64-bit ISA. The most common extension includes M(Integer Multiplication and Division), A(Atomic Instruction), F(Single-Precision Floating-Point), D(Double-Precision Floating-Point), etc. Hence, RV64IMAFD would represent a 64-bit processor with all the above-mentioned extensions. Since these extensions are essential for general-purpose computing, G(General) become the shorthand for all the above extension with two extra extensions, *Zifenci* and *Zicsr*. Building a fuzzer that could produce RV64I binary would be the first milestone of this project to demonstrate the potential of this method. Other extensions would be the stretch goal if time permits.

RISC-V belongs to the group of Reduced Instruction Set Computer (RISC), which means the architecture uses a fixed length of instructions. For RISC-V, the instruction length is 32-bit and has 32 integer registers. If the F extension is implemented, the implementation will have extra 32 floating point registers. Finally, there is another type of register called control and status registers (CSR), which are served for different purposes. For example, *mstatus* for displaying and controlling the status of the machine and *fcsr* for displaying and controlling the status of floating point instructions.

There are four core instruction formats in RISC-V, (R/I/S/U) and two variants based on handling of immediate (B/J) shown in Figure 2.1. The RISC-V ISA keeps the source (*rs1* and *rs2*) and destination (*rd*) registers at the same position in all formats to simplify decoding and the different combinations of value in field *opcode*, *funct7* and *funct3* decode into unique instructions in the ISA. The details of each instruction in RV32I, which is the base instruction class of RV64I, could be found in the appendix A.

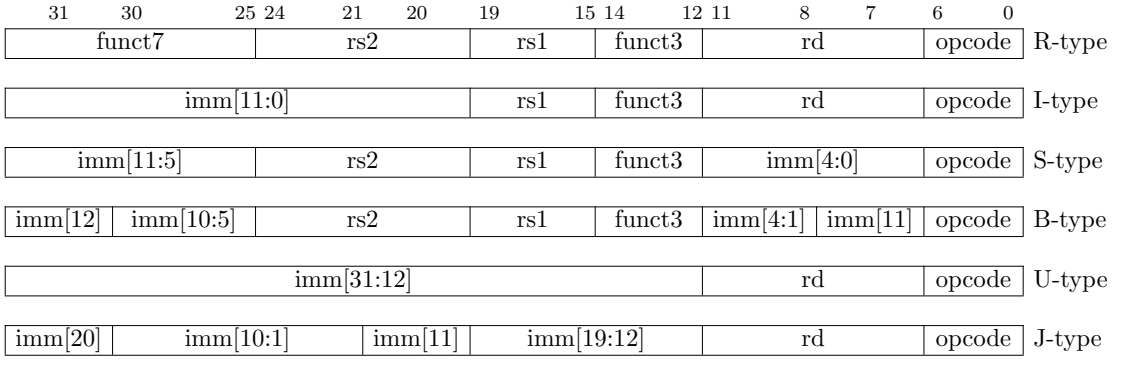

Figure 2.1: RISC-V Instruction types [1]

2.2 Instruction Set Simulators

Instruction Set Simulator (ISS) is a tool that runs on a workstation, called *host machine*, to mimic the behaviour of, or simulate a program running on another machine, called the *target machine*, which either does not yet exist or is not available. It also allows the user to examine the internal state of the target machine, such as register and flag status during the execution of each instruction [10].

ISSs are important tools in the development of conventional computer systems. They help to validate the processor design, and the compiler design, as well as evaluate architectural design decisions. Most common ISS requires the corresponding system application ELF file either compiled from the assembly file or the source file to simulate the model

Compared to other computer system simulators, ISS has faster execution speed with a higher level of abstraction, close to the position of gem5 / Emulator in Figure 2.2. However, the cost of it would be losing all the timing information and the model is not cycle-accurate compared to the RTL simulator. Nonetheless, ISS is still a good fuzzing target that could be fuzzed at speed while capturing all the details in the RISC-V ISA.

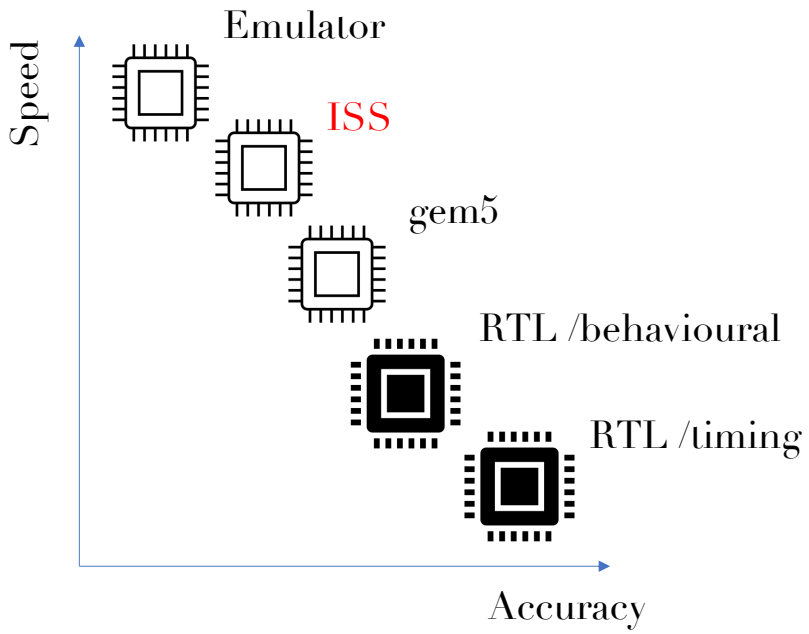

Figure 2.2: Simulation speed vs accuracy for the widely-used methods on different abstraction layers and simulation modes

2.2.1 Spike

Spike is the golden reference functional RISC-V ISS based on C++ [5]. It provides full system emulation and supports multiple ISAs: RV64IMAFDQCV. To run the simulator, one would need to prepare a compiled RISC-V ELF binary and feed it to Spike supported with RISC-V Proxy Kernel. The RISC-V Proxy Kernel(PK) is a lightweight application execution environment to support tethered RISC-V implementation with limited I/O capability and handles I/O-related system calls by proxying them to the host machine [11].

2.2.2 RVEMU

RVEMU is a Rust RISC-V emulator which is available online and in command-line interface [6]. The emulator supports RV64GC ISA. To run the emulator, an ELF file without headers and starts at the address 0x80000000 is needed.

2.2.3 FORVIS

FORVIS is a formal specification for the RISC-V ISA, written in Haskell [7]. It supports the RV64GC extension and executes RISC-V ELF binaries in a sequential fashion: a one-instruction-at-a-time, sequential memory model.

2.2.4 SAIL-RISCV

SAIL-RISCV is a formal specification of the RISC-V architecture, written in Sail [8]. Sail is a language for describing the instruction-set architecture (ISA) semantics of processors [12]. The model builds OCaml and C emulators that can execute RISC-V ELF files. However, only the C emulator supports the F and D extension.

2.3 Fuzzing

Fuzzing (short for fuzz testing) is "an automatic testing technique that covers numerous boundary cases using invalid data (from files, network protocols, application programming interface (API) calls, and other targets) as application input to better ensure the absence of exploitable vulnerabilities" [13]. During the fuzzing process, the program is monitored closely for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

The main advantage of fuzzing includes finding bugs which are impossible to find in pre-defined unit tests which may be biased due to the programmer's perception of the software system and it is much more similar to the real-life situation where the software needs to handle all kinds of user input. Also, it is a relatively simple and easy process to set up in the software development cycle that ensures a higher quality of the published software.

Nowadays, fuzzing is mostly an automated method to expose vulnerabilities in security-critical programs that might be exploited with malicious intent. A well-known real-life example of fuzzing is for Browser Security. Current web browsers such as Chromium of Google Chrome and Microsoft Edge are fuzzed extensively before being published to the stable version [14]. This greatly reduces the chance of a random hacker discovering a zero-day vulnerability in the browsers that could affect millions of users.

2.3.1 Mutation-based Fuzzing vs Generation-based Fuzzing

There are mainly two approaches of generating test inputs to the software. In the first approach, named mutation-based fuzzing, valid data is collected and then modified. The mutation operations could be random or based on heuristics. Examples of the operations would be replacing certain parts of the data or duplicating part of the data to extend the whole data. The second approach, denoted generation-based fuzzing, starts from a specification, which describes the syntax of the input data of software. The main challenge of both approaches is producing effective test cases that exhibit diversity and provoke hidden bugs in the software. Yet, the input data should still be syntactically correct to avoid the target software quickly discarding the input as invalid.

Mutation-based fuzzing requires little to no knowledge of the input format or the software itself. By contrast, generation-based fuzzing requires a significant amount of preparation to understand the specification and manually spell out the grammar of the data. And sometimes the grammar written by the human may limit the variety of input data. Nonetheless, it is often found that generation-based fuzzer creates higher quality test cases with the extra knowledge given to the fuzzer [15].

2.3.2 Structure Aware Fuzzing

Generation-based fuzzers usually target a single input type, generating inputs according to a pre-defined grammar. Mutation-based fuzzers are not restricted to a single input type and do not require grammar definitions. Thus, mutation-based fuzzers are generally easier to set up and use than generation-based fuzzers. But the lack of input grammar can also result in inefficient fuzzing for complex input data structure, where any traditional mutation (e.g. bit flipping) leads to an invalid input rejected by the target software in the early stage of parsing. Hence, structure-aware fuzzing is proposed to incorporate the benefit of generational-based fuzzer into mutation-based fuzzers. Custom mutators, which perform data mutation with respect to the structure of the input

data, are defined by the user in the fuzzer. Carefully implemented custom mutation can ensure the input is syntactically correct even after mutation [16].

More details about custom mutator could be found in later section 2.4

2.3.3 Blackbox and Whitebox Fuzzing

Another component of the fuzzer is its ability to monitor the execution of the software. It could be categorized into three main groups: blackbox, whitebox and greybox [17].

Blackbox fuzzing treats the software as a black box and is unaware of its internal structure. The advantage of blackbox fuzzing is that it is the easiest to set up among these three types of fuzzing since it does not require the source code of the software. Also, the fuzzing process can be relatively fast compared to the other two methods and be easily parallelized. However, blackbox fuzzers may often only scratch the surface and expose "shallow" bugs. For example, in the domain of compiler, blackbox fuzzers are found to be prone to the problem of immunity: after finding and fixing a substantial number of bugs in a particular compiler using the black fuzzer, it fails to generate programs that trigger further bugs in that compiler [18]. A blackbox fuzzer has no way of adapting its generation strategy to find more bugs without any feedback from the execution of the software.

Whitebox fuzzing lies at the other end of the spectrum. The fuzzer can take advantage of information such as the source code and design specification of the software, allowing greater observation of the software structure [17, p. 9]. The extra information often can be used to increase the code coverage and the efficiency of the fuzzing process. Common whitebox testing method includes program analysis, symbolic execution and model-based testing. A whitebox fuzzer can be very effective at exposing hidden bugs. However, the manual effort and time used for analysis could be very costly.

2.3.4 Greybox Fuzzing (Coverage-guided fuzzing)

Greybox fuzzing lies in between blackbox fuzzing and whitebox fuzzing: the fuzzer does not have access to source code yet be able to examine the behaviour of the software using other methods besides observing the input data and the output data. The most common Greybox Fuzzing method leverages instrumentation from the compiler to gather information during the execution of the program and prioritise input data that lead to increased coverage in the software, denoted Coverage-guided fuzzing. The most commonly used coverage-guided fuzzer, *American fuzzy lop* (AFL) [19] and *libFuzzer* [20], utilize lightweight instrumentation to trace basic block transitions exercised by the input data. To perform coverage-guided fuzzing, if the source code is available, the software needs to be compiled with special compiler flags that add extra instrumentation to the compiled binary code. Otherwise, the program could be run in an emulator such as QEMU [21] to allow its execution to be tracked [22]. Although instrumentation induces performance overhead compared to techniques without a feedback loop, more hidden bugs could be discovered with good coverage of the software and relatively low set-up time.

Further detail of libFuzzer would be introduced in the following section 2.4.

2.4 libFuzzer

libFuzzer is a mutation-based coverage-guided in-process fuzzing engine based on LLVM for C and C++ software [20]. Using the libFuzzer requires compiling the software under test (SUT) using a special flag, *-fsanitize=fuzzer*. After linking with the SUT, it feeds the fuzzed inputs to the software via a specific fuzzing entry point (*fuzz target*). The fuzz target is a function that accepts an array of bytes and translates the bytes into the input of the SUT and invokes the SUT. The fuzz target would be called repeatedly by libFuzzer during the fuzzing process with a sequence of mutated bytes. An example of the fuzz target function, *LLVMFuzzerTestOneInput* is shown in the below code snippet 2.1.

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  DoSomethingInterestingWithMySoftware(Data, Size);
  return 0;
}
```

Listing 2.1: Example code for LLVMFuzzerTestOneInput [20]

The tool would look for *initial corpus* provided by the user at the start. The initial corpus should ideally contain a collection of valid input byte arrays although libFuzzer would work without any initial inputs by using a default initial byte sequence. However, it would be less efficient if the input of SUT is complex and structured as it would take extra time for the fuzzer to explore new code paths in the software.

At the same time, if the input structure of the SUT is complex and structured, the default mutator of libFuzzer would be difficult to generate valid input to the SUT. It is because the mutation operations in the default mutator are simple and domain-agnostic, for example, deleting, swapping, changing, or duplicating bytes. It would be challenging for the default mutator to mutate a valid input perfectly into another valid input.

To address these problems presented in Section 2.3.2, libFuzzer also provides the option to incorporate a dictionary into the default mutator or to replace the default mutator with a user-defined custom mutator for the mutation of the bytes. The user-supplied dictionary could be useful in text-based input that the default mutator could insert, delete or substitute tokens instead of individual bytes. In other situations, such as the assembly program for the Instruction Set Simulator, a customer mutator would be the better choice. The custom mutator would take a sequence of bytes and return the mutated byte sequence according to the corresponding input format of the SUT. Using this interface, the users can define their own mutation operations which are the most suitable to the fuzz target input. An example of the built-in custom mutator function, *LLVMFuzzerCustomMutator* is shown in the code snippet 2.2 below.

```
extern "C" size_t LLVMFuzzerCustomMutator(uint8_t *Data, size_t Size,
                                          size_t MaxSize, unsigned int Seed) {
  uint8_t Uncompressed[100];
  size_t UncompressedLen = sizeof(Uncompressed);
  size_t CompressedLen = MaxSize;
  if (Z_OK != uncompress(Uncompressed, &UncompressedLen, Data, Size)) {
    // The data didn't uncompress. Return a dummy...
  }
  UncompressedLen =
      LLVMFuzzerMutate(Uncompressed, UncompressedLen, sizeof(Uncompressed));
  if (Z_OK != compress(Data, &CompressedLen, Uncompressed, UncompressedLen))
    return 0;
  return CompressedLen;
}
```

Listing 2.2: Example code for LLVMFuzzerCustomMutator [16]

The custom mutator API provides two extra arguments comparing to the fuzz target counterpart with *MaxSize* specifies the maximum size of the mutated data and *Seed* is provided to ensure the mutation is reproducible.

The example code above is based on Zlib-compressed data. It would be near to impossible for the default mutator to mutate valid compressed data into another valid compressed data input. Hence in this custom mutator, it first tries to uncompress the data and store it in a temporary byte array, *Uncompressed*. If the uncompression is not successful, it would return a syntactically correct dummy input, much like providing initial test cases in the initial corpus mentioned. Then it mutates *Uncompressed* using the default mutator API, *LLVMFuzzerMutate*. Finally, it re-compresses *Uncompressed* and passes the new compressed data to the original byte array, *Data* with its size, *CompressedLen* to the *LLVMFuzzerTestOneInput* function. In this way, it ensures the mutated data feed into the fuzz target has a valid structure.

The fuzzer tracks the code coverage of the software from the *Clang* compiler and generates new input data to maximize the overall code coverage. The input data which leads to new source code coverage of the simulator, it would feed back to the corpus and would be prioritised for further mutation, which explains its *coverage-guided* property. The fuzzer would terminate when it finds

a data input that would crash the SUT, which is the original intention of the program to discover subtle and unconventional bugs in the SUT. Yet, in this project, we would also utilize the coverage-guided property of libFuzzer to obtain valuable test cases generated during the fuzzing process to build our test suite and perform fuzzing by proxy.

2.5 Fuzzing by proxy

The main assumption of *fuzzing by proxy* is that if a test case provides good code coverage for one RISC-V implementation, these tests are also likely to trigger interesting behaviour in the other RISC-V implementations, and potentially uncover defects. Figure 2.3 tries to visualize the whole *fuzzing by proxy* process. The problem starts from the left red arrow when we want to fuzz test software A. However, there are various reasons that it could be difficult such as not being able to collect coverage information from the software or the execution speed of the software being too slow. Hence, instead of building a fuzzer for software A, we fuzz test software B which has the same input format as software A and is easier to perform fuzzing on, indicated by the green arrow loop on the right. After generating a wide variety of test cases stored in the test corpus from the fuzzing of software B, we use those test cases to test software A to observe if software A would crash or compare its behaviour to software B's.

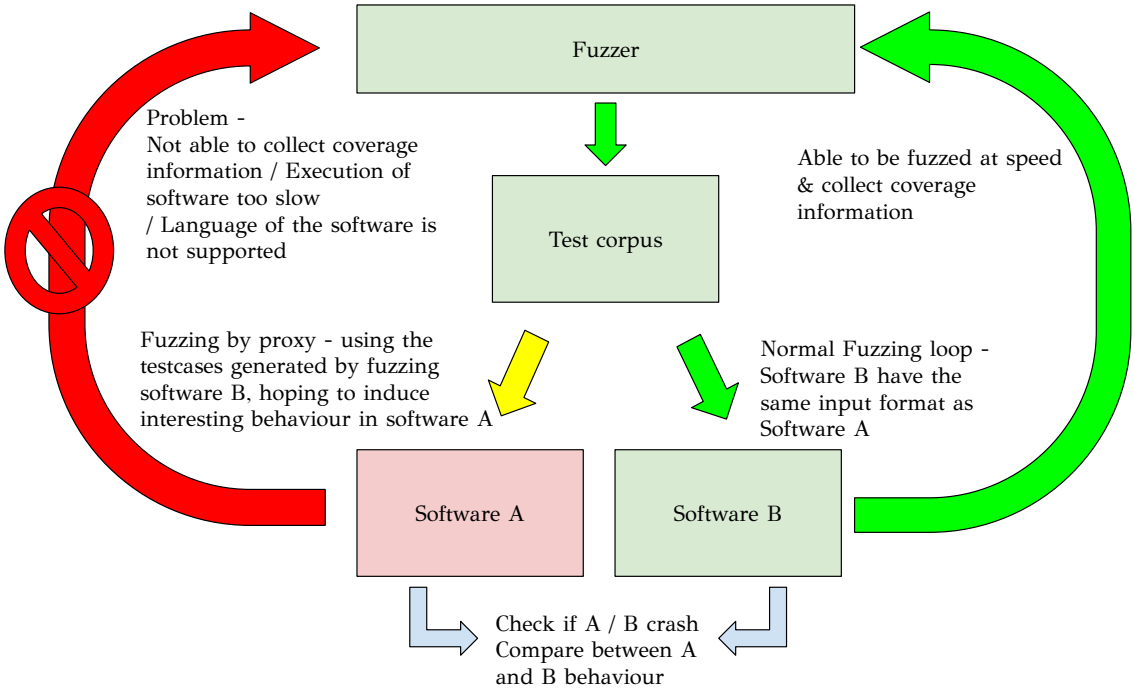

Figure 2.3: Fuzzing by proxy

2.6 Related Work

There has been some previous work on verifying RISC-V ISA implementation / generating RISC-V test suites. Herdt et al. has a similar approach to the problem, using coverage-guided fuzzing to verify ISS [23]. However, in our project, we expanded into the RV64G instruction set and integrated Spike into the libFuzzer instead of the RISC-V Virtual Prototype [24]. Furthermore, we have tested out additional ISS implementations such as SAIL-RISCV and RVEMU. They have also proposed another approach for generating RISC-V ISA test suites using SMT solver with provided instruction constraints and coverage requirements [25].

Structure-aware fuzzing has been a hot research topic since it shows greater potential than plain fuzzing at finding more and deeper bugs as the software becomes "fuzz-immune" with more and

more trivial bugs fixed. Successfully applied structured-input cases include XML and Javascript [26], ELF, JPEG and WAV file [27] and C language [28].

The idea of fuzzing by proxy could be found in a few recent researches. Andrey discovered a way to generate test sets by coverage-guided fuzzing with the Linux kernel USB subsystem and used them to find bugs in the Windows kernel USB subsystem [29]. Trippel et al. have adapted RTL designs into software models and used only open-source coverage-guided tools to create a Hardware Fuzzing Pipeline to fuzz hardware such as RISC-V CPUs by proxy [30]. Google also created a system called SiliFuzz for finding electrical defects in millions of CPU cores by fuzzing software proxies such as CPU simulators and disassemblers [31]. Most recently, this idea is also applied in the fuzzing of the C compiler, GrayC which greatly reduces the manual effort to compile all the test compilers in a manner suitable for greybox fuzzing [28].

Chapter 3

Integrating Spike with libFuzzer

This chapter is going to introduce several problems that we have encountered when integrating Spike with libFuzzer. Section 3.1 details the discovery of the Spike entry point and the technical difficulties we have overcome to make Spike fuzz-friendly. Section 3.2 explain and discuss our approach of transforming mutated data into valid RISC-V test binary.

3.1 Creating a fuzz target for Spike

The first step of building a fuzzer with Spike is to build a fuzz target, which is a function that accepts an array of bytes and translates the bytes into the input of the SUT and invokes the SUT. Therefore, we need to identify the "main" function that could invoke the SUT. Without internal documentation or guidance, it could be a difficult task with a huge code base. However, following the advice of investigating the top-level files, we discovered the potential entry point of Spike in the continuous integration testing file shown in the following code snippet 3.1.

```
int main(){
  std::vector<mem_cfg_t> mem_cfg { mem_cfg_t(0x80000000, 0x10000000) };
  std::vector<int> hartids = {0};
  cfg_t cfg( /*parameters settings...*/ );
  std::vector<std::pair<reg_t, abstract_device_t*>> plugin_devices;
  std::vector<std::string> htif_args {"pk", "hello"};
  debug_module_config_t dm_config = { /*parameters settings...*/ };
  std::vector<std::pair<reg_t, mem_t*>> mems = make_mems(cfg.mem_layout());
  sim_t sim(&cfg, false, mems, plugin_devices, htif_args, m_config,
            nullptr, true, nullptr, false, nullptr);
  sim.run();
}
```

Listing 3.1: "ci-tests/testlib.c" from Spike repository

```
$ spike pk hello
```

Listing 3.2: Command for executing a C program with Spike

Comparing the arguments of the *htif_ args* in 3.1 and the command for running a C program with Spike listed in 3.2, it is obvious that the second argument of *htif_ args* is responsible to pick up the name of the input RISC-V test binary. After setting the execution environment, *sim.run()* starts the actual simulation of instructions execution. Hence, if we could substitute the content in the binary with the new mutated data, we are effectively fuzzing the Spike. Consequently, the problem now lies in how to translate new mutated bytes into RISC-V binary. The solution to the problem would be further explained in section 3.2.

Before tackling the problem of translating bytes into RISC-V binary, we have tried to test run the fuzzer with static RISC-V binary. As we might expect, there are various issues occurred which prevent successful fuzzing. In the following sections, we are going to introduce two major problems we have encountered which demonstrate the limitation of libFuzzer.

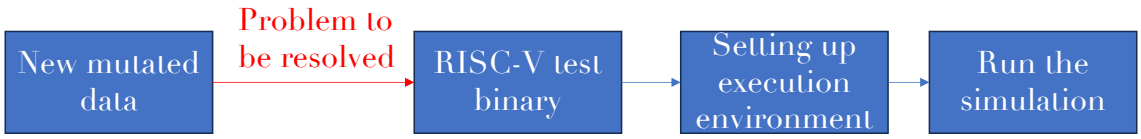

Figure 3.1: Fuzzing Loop v1

3.1.1 Problem: Device Tree Blob compilation

The first problem quickly occurred with a fuzzer crash and the error message, “Child dts process failed”. By searching the code base with the error message outputted, we quickly discovered the problem originated from the function called *dts_compile*, which is called during the initialization of the execution environment.

DTS in this context refers to the Device Tree Source file and its compiled binary form is referred as Device Tree Blob (DTB). The Device Tree is a tree-like data structure that provides a vendor- and architecture-independent description of the system’s hardware components, such as CPUs, memory, peripherals, and their respective configurations and interconnections [32].

Interestingly, we have tested the same setup but without the integration of the libFuzzer and that program completed its execution successfully. Moreover, we have also checked the only argument for *dts_compile* function, the DTS file is the same with and without the integration of libFuzzer. It is anticipated as we have kept the execution environment variable(i.e. memory, CPU cores, peripherals) the same. Therefore, it is most likely the divergence of behaviour occurred during the DTS compilation. Yet, it is difficult to track the behaviour of the function as it involves forking. We suspect the process launched by the function is interfered by libFuzzer which leads to the behaviour difference. Nonetheless, we could not draw a conclusion without tracking the execution of the process.

After spending a considerable amount of time debugging the issue, we have decided to come up with a workaround instead of completely resolving the unexpected error as our main goal of this project is not closely related to this problem. The working principle of the workaround is to bypass the *dts_compile* function by pre-compiling the DTS file into DTB by dumping the DTB output from raw Spike. Afterwards, we supplied the DTB back to the fuzzer for setting up the execution environment without calling the *dts_compile* function. Although we need to pre-compile the DTB every time if we want to change the execution environment, the execution speed of the fuzzing loop would increase since it omits the compilation of the DTS file.

The main execution environment variables are listed below for reference, rest of them could be found in the implementation of fuzzers, "fuzzer_vX.cpp":

- Default ISA: RV64GCV
- Default Privilege: MSU
- Default endianness: little endianness
- Hart (Core): 1
- No plugin device
- Memory base address: 0x80000000
- Memory size: 0x10000000

3.1.2 Problem: Too many Open files

Another significant challenge we have encountered is the "Too many open files" problem. After running the fuzzer for around ten minutes, the compiler(would be introduced in section 3.2) complained about too many files are opened and exited early. This is not ideal as the mutated data is not translated into a new test RISC-V binary. Therefore, the following simulation is meaningless as it would not increase the code coverage of Spike and provide information for the fuzzer. Also, it would be inconvenient to manually rerun the fuzzer for every ten minutes.

When a process opens a file, it requires a file descriptor to access and manipulate that file. The operating system allocates a finite number of file descriptors to each process. If a process exceeds this limit, it will encounter an error. The limit is often set by the operating system and could be increased by using the *ulimit* command [33]. However, based on our observation, the file descriptor count was increasing steadily during the execution of the fuzzer and increasing the limit would only delay the bug from happening.

Hence, we further investigated the Filesystem usage during the execution of the fuzzer using *lsof* and *fs_ usage* commands. We detected the increase of file descriptors originating from the *dup* system call on the */dev/ttys00X* file. The *dup* system call is responsible for allocating a new file descriptor that refers to the same open file description as the given descriptor while the */dev/ttys00X* is a special file that represents the terminal for the current process. The combination of the two suggests an I/O redirection, such as redirecting standard output or standard error to a file.

```
syscall_t::syscall_t(htif_t* htif)
  : htif(htif), memif(&htif->memif()), table(2048)
{
    ...

  stdin_fd = dup(0);
  stdout_fd0 = dup(1);
  stdout_fd1 = dup(1);
  if (stdin_fd < 0 || stdout_fd0 < 0 || stdout_fd1 < 0)
    throw std::runtime_error("could not dup stdin/stdout");

  fds.alloc(stdin_fd); // stdin -> stdin
  fds.alloc(stdout_fd0); // stdout -> stdout
  fds.alloc(stdout_fd1); // stderr -> stdout
}

//Added deconstrutor to release the file descriptor
syscall_t::~syscall_t()
{
  close(stdin_fd);
  close(stdout_fd0);
  close(stdout_fd1);
}
```

Listing 3.3: "fesvr/syscall.cc" from Spike repository

By searching through the code base, we have identified the root of the problem in a constructor function. As shown in the code snippet 3.3, the constructor tries to duplicate the standard input and standard output file descriptor and redirect the standard I/O of another file to them, which verifies our guess. Yet, the duplicated file descriptor is not released at the deconstruction of the variable and leads to the continuous increment of the file descriptor.

To solve the issue, we follow the RAII(Resource Acquisition Is Initialization) design pattern which ties the lifetime of a resource to the lifetime of an object. The duplicated file descriptor is stored as object variables in the constructor and a destructor is added to release the file descriptor when the object is destroyed. It ensures number of the file descriptor stay constant at the end of the execution.

The problem displays one of the difficulties in the integration with the libFuzzer that hidden bugs such as a memory corruption that goes undetected in the fuzz target would accumulate during the repeated runs and cause major problems in the fuzzing cycle. Hence it is recommended to run libFuzzer with sanitisers, such as AddressSanitizer (ASAN) or UndefinedBehaviorSanitizer (UBSAN) to detect most bugs on the spot.

3.2 Translating bytes into RISC-V binary

3.2.1 Method One: ELF template

In the Herdt et al. paper, they suggest creating a RISC-V ELF template with a linker script that generates an empty section in it. Then, using the objcopy utility with the –update-section argument, they could overwrite the empty section with the instruction byte stream and effectively create a new RISC-V binary test case [23].

3.2.2 Method Two: Assembly template

However, unfamiliarity with the linker script becomes the main obstacle for the ELF template approach. Hence, we have switched to a more manageable approach, the Assembly template. By compiling a simple C source file with a placeholder assembly section using the inline assembly feature, we could obtain a skeleton assembly file.

The problem now becomes the transformation between bytes and text assembly, which is exactly the job of a disassembler. Through further code inspection, we discovered the Spike built-in disassembler and its API. The disassembler API allows one 32-bit instruction to be disassembled at a time. And by grouping the bytes into a group of 4 to form 32-bit instructions, we could obtain the text assembly and insert them into the placeholder section in the assembly template. After compiling the modified assembly template, we obtain the test binary which allows us to finish the rest of the operation in the fuzzing loop.

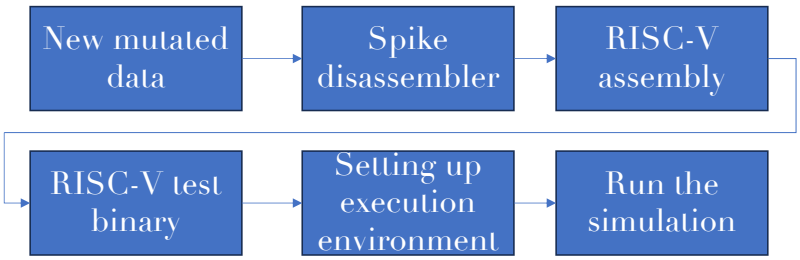

Figure 3.2: Final Fuzzing Loop

In the integration with the Spike disassembler, we encountered a few parsing problems from the disassembled instructions. The most significant problem is related to the branch and jump instructions. For example, a branch instruction could be disassembled as follows:

```
bnez a0, pc - 8
```

Yet, the compiler would complain about not being able to find *pc* as a referenced label and the compilation fails as it requires the current address of the instruction to calculate the branching address. A simple solution is to add the *pc* label in front of the instruction. Moreover, a unique number identifier is attached to each label to avoid cross-referencing when the assembly has multiple branch and jump instructions.

```
pc0: bnez a0, pc0 - 8
```

Compared with method one, the assembly template method allows us to fuzz test the Spike disassembler at the same time since its coverage is also collected during fuzzing. Also, for method one, the mutated data needs to be semantically correct otherwise it would be rejected early in the Spike decoding process. However, with this method, we could filter out unknown instructions classified by the Spike disassembler. However, the fuzzing speed is slower due to the additional processing of the Spike disassembler and the compilation of the modified assembly.

Chapter 4

Custom Mutators for RISC-V

This chapter introduces one of the most important features of the fuzzer, the custom mutator. Section 4.1 further discusses the operations of the default mutator and the problem it generates when directly applied to RISC-V fuzzing. Section 4.2 presents the basic version of the RISC-V custom mutator with the ability to exchange opcode. Section 4.3 displays several improvements based on the basic version, such as better type-aware and more variety of mutation operations.

4.1 Version 1: Default mutator

To implement the fuzzer with the default mutator, simply compile the fuzzing loop 3.2 with the *-fsanitize=fuzzer* flag with no extra code. The code is implemented in "fuzzer_v1.cpp".

As mentioned in section 2.3.2, the default mutator could apply various mutation operations, such as

- bitwise flipping, an operation that changes the value of individual bits within a binary representation of data.
- block insertion / deletion, inserting / deleting new blocks of data into the existing input. The inserted blocks can contain random data, data copied from other parts of the input, or a combination of both while the size and location of the deleted block could also be random.

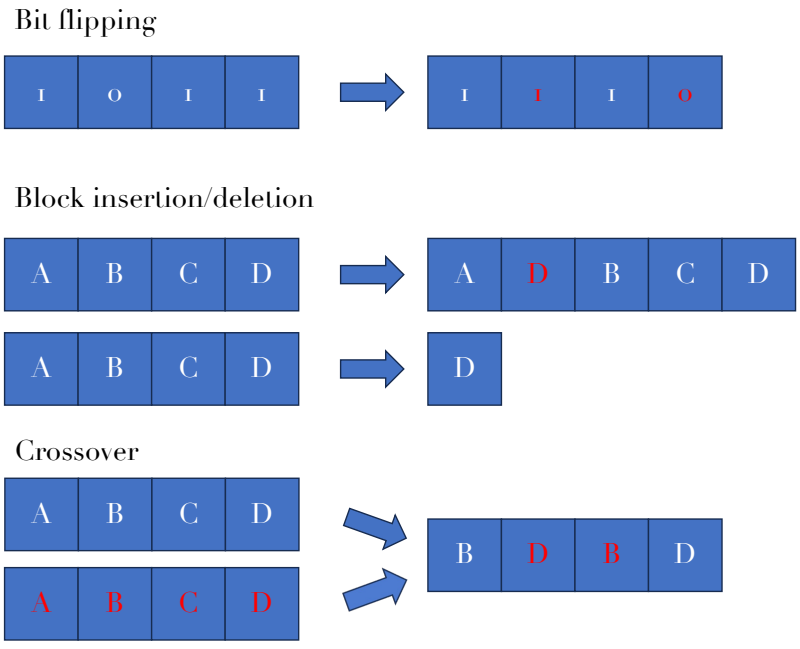

Figure 4.1: Mutation operation of default mutator

In addition to mutation, libFuzzer also supports cross-over operations. It takes two inputs and combines parts of their contents to create a new input. This process mimics the cross-over operation in genetic algorithms, where characteristics from two parents are combined to produce offspring with new traits.

However, this system is not suitable for the RISC-V instruction. This can be illustrated briefly by the bit-flipping operation. The opcode of ADD instruction is 0110011. If the default mutator decides to flip the third and fourth bit of the opcode, which becomes 0101011. The opcode is not owned by any instruction in the G extension and it would be recognized as an unknown instruction and got rejected by Spike. On the other side, it is possible that the "unknown" mutated instruction trigger crashes in Spike. However, only the ELF template method mentioned in section 3.2.1 would be able to include these "unknown" instructions into the test binary while the assembly would not be assembled in the compiling stage for the assembly template method. Moreover, it is unlikely that the test case that crashes Spike with invalid instruction is useful in testing other implementations in fuzzing by proxy as the invalid instruction is more likely to be filtered out before triggering any unintended behaviour.

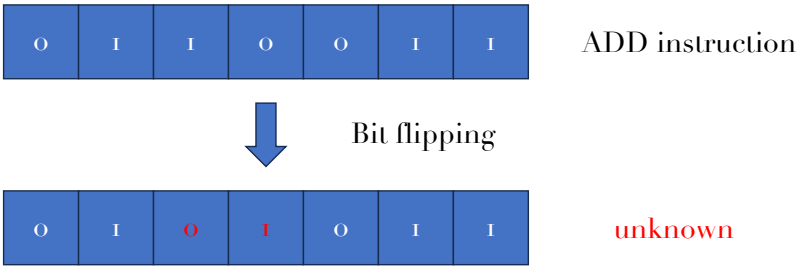

Figure 4.2: Problem with default mutator

In line with our expectation, most of the mutated instructions are invalid with numerous instructions classified as unknown or as compressed instructions. Compressed instructions are short 16-bit instruction encodings for common operations as the C extension of the RISC-V. Since the disassembler is not able to decode these data as 32-bit instructions, it tries to decode the data as two 16-bit compressed instructions. And due to reduced opcode space, they have a higher chance to be successfully decoded.

During the fuzzing with the default mutator, we also encountered the timeout problem. The fuzzing loop would freeze and stuck at the simulation stage. By inspecting the test case causing the freeze, we speculate that the jump and branch instructions are the root of this problem. Normally, these instructions would not be committed as the condition is not met. However, there are still chances that the condition is met or the instruction is unconditional. To address this situation, libFuzzer provides the option for timeout which allows the process to be treated as a failure case if the input takes longer than a specified time. For this project, we have set the timeout to 30 seconds which is more than enough to run the test cases which are normally less than 20 lines. Moreover, since the fuzzer would stop after encountering a timeout test case, we have used a while loop to wrap the fuzzer, which reruns the fuzzer until a specified period has passed. The detailed implementation could be inspected in the bash script *run_fuzzer.sh*.

4.2 Version 2: Simple custom mutator

Different from version 1 of fuzzer, this version utilizes the custom mutator API provided by libFuzzer to develop the simplest type of custom mutator. The manual effort in the implementation is minimal to demonstrate the impact of the custom mutator under limited human input.

At the start of the custom mutator, we first seed the random number generator by using the *srand* function. It ensures the mutation by the custom mutator is reproducible. Next, we check whether the size of the data can at least compose one 32-bit instruction to perform mutation. If not, we assess whether the maximum size, *Max Size* allows us to accommodate one or more instructions. If so, we would generate one by using the *genInstr* function. The *genInstr* can generate all six type of instruction with equal probability by randomly choosing one of the *genXInstr* that generate X-type of instruction. Lastly, by using the random number generator, the mutator decides on which instruction to mutate.

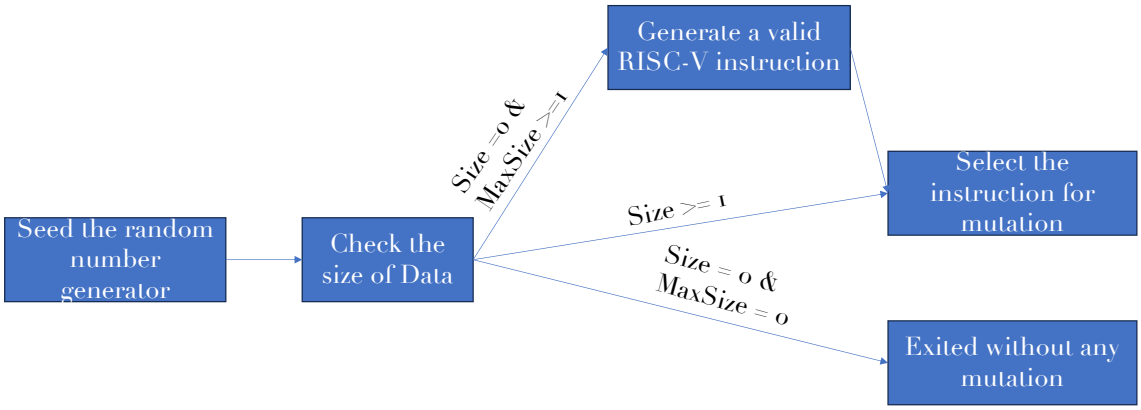

Figure 4.3: Setup of custom mutator (Size and MaxSize in the unit of 32-bit instruction)

The custom mutator is able to perform four types of mutation:

- Opcode mutation: Mutating one opcode (bit 0 - 7) to another valid opcode
- "Immediate" mutation: Replacing the bit 8 - 31 with random values
- Instruction insertion: Inserting a random valid instruction into random position if *MaxSize* allows
- Instruction deletion: Deleting a random instruction at a random position

The first type of mutation directly addresses the invalid opcode problem due to random bit flipping mentioned in section 4.1. Instead of replacing individual bits randomly in the instruction, the mutation replaces the opcode section as a whole. The second type of mutation is a simple extension of the first one to the remaining bits, which usually accepts all possible values.

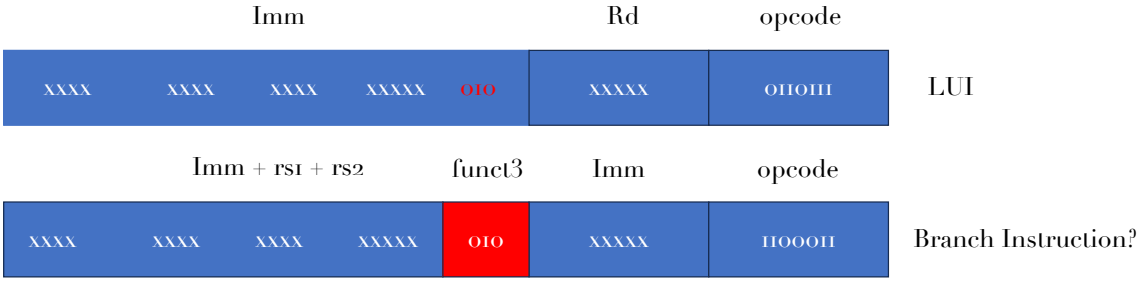

Figure 4.4: Problem of direct opcode replacement

Although the opcode mutation method solves the bit-flipping problem, it does not guarantee the mutated instruction is semantically correct. For example, shown in Figure 4.4, the opcode of the LUI instruction, a U-type instruction, is replaced by another opcode of the branch instruction, a B-type instruction. Hence, the bits are interpreted differently. Some of the field value is compatible such as the destination register value(*Rd*) of LUI and part of the Immediate value(*Imm*) of the branch instruction since the maximum value of *Rd* is 31, same as the maximum encoded value in a 5-bit field. However, the funct3 value for the branch instruction does not allow the value of 0b010 while it is possible for the last three bits of immediate field of the LUI instruction to be 0b010. The structural difference of instructions requires more attention to the instruction type during mutation which leads us to the advanced custom mutator.

4.3 Version 3: Advanced custom mutator

To tackle the lack of type awareness mentioned in section 4.2, the advanced custom mutator includes the *decode* function that acts like the decoder in the CPU. The function can recognize

the opcode of the selected instruction and classify it into one of the six types of instruction. To accompany the function, an enumeration data type for instruction, *opcode_type* is introduced for code clarity. As a kind of safety net, if the *decode* function does not recognize the opcode, the instruction would be classified as *unknown* and that instruction would later be replaced by a valid instruction before performing any mutation.

With the information of the type of instruction, *mutate_opcode_v3* can perform a more precise mutation by only swapping with the opcode of the same instruction type. Therefore, it reduces the chances of instructions mutating into invalid instructions due to structural differences. A similar idea is also applied to the "immediate" mutation and is captured by the *mutate_imm* function in *instruction.hpp* file. To facilitate the implementation of the function while maintaining the original functionality, the *genXInstr* function is reworked to allow an opcode argument using function overloading. To illustrate, code snippet 4.1 displays the function *genSInstr*. *S_opcode* is a vector container that contains all possible opcodes of S-type instructions. The first function generates S-type instruction with respect to the opcode argument. It would be called by *mutate_imm* function for generating new "immediate" bits while maintaining the structure of the S-type instruction. The second function is to preserve the *genSInstr* original functionality, generating a random S-type instruction. It is done by first randomly choosing an opcode from *S_opcode* and calling back to the first function for generating the remaining part of the instructions.

```
uint32_t genSInstr(uint8_t opcode)
{
    int index = opcode_ind(S_opcode, opcode);
    //Generate instruction with respect to the type and opcode
    return res;
}

uint32_t genSInstr()
{
    int index = rangeSelector(0, S_opcode.size()-1);
    uint8_t opcode = S_opcode[index];
    return genSInstr(opcode);
}
```

Listing 4.1: genSInstr

The advanced custom mutator is also able to perform three additional types of mutation:

- Instruction swap : Swapping the position of two instructions
- Instruction duplication: Duplicating a random instruction and place it at a random position if *MaxSize* allows
- Instruction replacement: Replacing a random instruction

The additional types of mutation are basically the reprints of the default mutator mutations but working at the unit of instructions. The introduction of these mutation operations would bring more variety in the mutated data and be able to create interesting test case in less mutation iteration. However, there are certain situations that these mutations can not be performed:

- Instruction insertion / duplication that would exceed the *MaxSize*
- Instruction swap / deletion when there is only one instruction

In those cases, the default advanced opcode mutation is performed instead.

Several minor enhancements are also implemented in this iteration of the mutator. Due to the unbalanced distribution of instruction type of RISC-V instruction with more than 100 R-type instructions and less than 10 for each U, B, and J-type instruction, generating each type of instruction with equal probability lead to mutated instructions biased towards U, B, J-type instructions. Hence, the generation probability is readjusted to account for the population of each type of instruction.

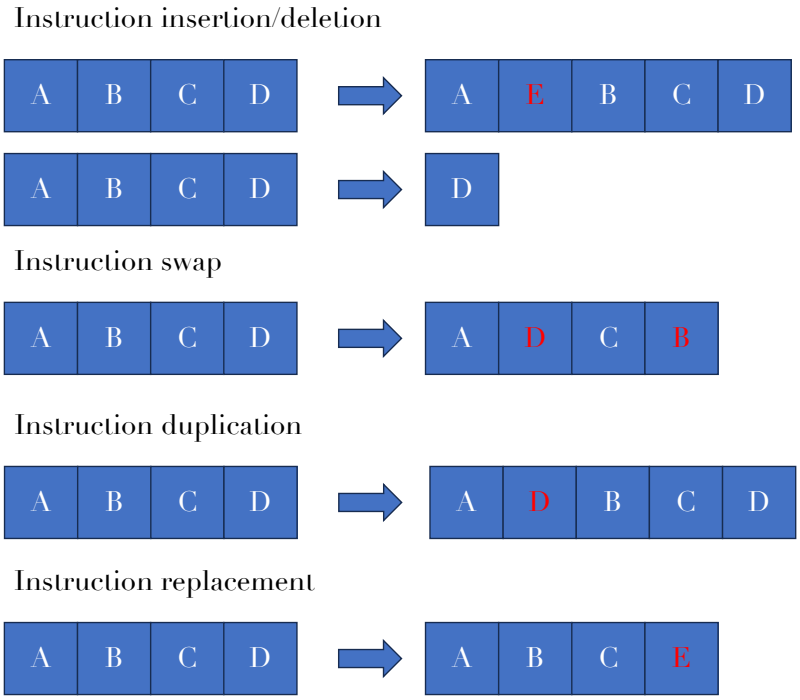

Figure 4.5: Mutation operations of advanced custom mutator

Chapter 5

Fuzzing by Proxy

The section below describes the final part of the project, fuzzing by proxy. Section 5.1 introduces overall structure of the operation and illustrates the details in the implementation. Section 5.2 present an unexpected challenge faced in analysing the output.

5.1 Overall design

After obtaining the test cases by fuzzing Spike, we can fuzz test the remaining implementations: RVEMU, FORVIS, SAIL-RISCV as described in section 2.5. First and foremost, the test cases which are in bytes are disassembled into assembly instructions using the program, *disasm*. Compared with the fuzzer, *disasm* only contains the Spike disassembler feature and directly fills in the instruction sequence into four different assembly templates for each implementation. The variation of the template is mainly due to some instructions in the original template are not compatible with specific implementations. Hence, they are replaced or modified to adapt to the implementation. After each version of the assembly files is compiled into binaries, they are executed in the simulation of each implementation.

The bash script, *read_IMP_reg.sh* will record whether the simulation crash and the ending state of the machine. The following table summarises the available information provided by each implementation for state comparison:

Available output value	Spike	RVEMU	FORVIS	SAIL-RISCV
Integer registers	✓	✓	✓	✓
Floating point registers	Decimal	Decimal	Hex	Hex
Control and Status Register	✓	✓	✓	
Memory	✓			

Table 5.1: State information provided by each implementation

In this project, the state of integer registers and floating point registers are selected for the comparison as they are the only information that is available across all implementations. The method of obtaining the register value for each implementation is described below:

- Spike: A text file, *read_reg_commmand.txt* is provided as the debug argument when running Spike. The file contains a list of debugging commands. The first command instructs Spike to run until the end of the test case using the address of the last instruction obtained via objdump disassembly. Then, we extract the register value using *reg* and *fregd* commands and their value would be displayed through standard error. More details regarding the debugging command are described in Appendix B.

- RVEMU: The final state of the machine is shown after the simulation. Hence, we simply need to extract the register state from it. An example raw output of RVEMU is shown in Appendix C.1

- FORVIS: Similar to RVEMU, the implementation produces the state of the machine at each instructions. Therefore, to obtain the final register state, we need to parse the last state of the machine only. The example raw output of FORVIS is shown in Appendix C.2.
- SAIL-RISCV: Different from FORVIS and RVEMU, it only shows the update of register value at each instruction. Hence, we need to track the latest value update of each register to come up with the final state of each register. An example raw output log of SAIL-RISCV could be seen in Appendix C.3.

All the register values are stored in *IMP_reg_output.txt*.

However, the simulation could also end in other ways: the simulation crashes, the simulation timeout, and the simulation exited early due to exceptions. Any crashes and early exits would be recorded by the script and displayed in the final comparison log. In the case where the simulation exits early, there may be differences in output behaviour and the data is handled differently. For example, Spike would display the last value of integer registers only in the error log. Therefore, in that case, the value of floating point registers is not obtainable and would be assumed as zero during the comparison phase. A sample error output is shown in Appendix D

Finally, the comparison of register values is performed using the program, *cmp_reg_output*. The value of the integer registers is compared across all implementations while the value of the floating point register is compared in pairs (SPIKE/RVEMU, FORVIS/SAIL-RISCV) due to differences in the output format. Any difference in register value would be displayed with the corresponding test instruction sequence in a human-readable format. Those cases would be further examined and discussed in the evaluation section [Section 6.3]. An example output of *cmp_reg_output* is shown in Appendix E.1.

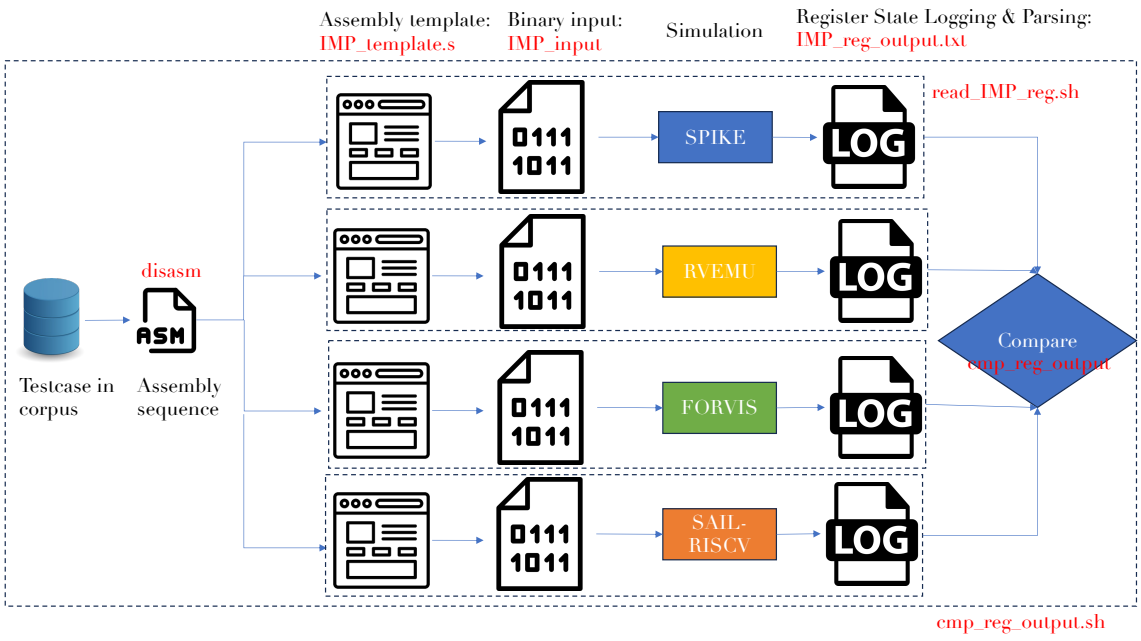

Figure 5.1: Fuzzing by proxy flow chart (IMP: name of implementation)

The complete Fuzzing by Proxy flow chart is shown in Figure 5.1.

5.2 Challenge: Standardizing the initial state

Early testing using this fuzzing by proxy approach revealed a problem: the execution of test cases does not start from an identical initial state in different implementations. It results in meaningless comparisons between the final registers states. One of the situations we encounter is the floating point register value not changed at all in the SAIL-RISCV implementation compared to other implementations when the test case contains floating point instructions. Yet, SAIL-RISCV is compatible with RV64F(floating point) extension. Further inspection of the output log reveals that

these floating point instructions are trapped in the SAIL model instead of treated as unrecognized instructions. Hence, we suspect a certain bit of the CSR is not set properly. After extensive investigation of the RISC-V specification, we have determined that the *FS* field of the *mstatus* causes the trap. The *FS* field encodes the status of the floating-point unit state, including the floating-point registers *f0–f31* and the CSRs *fcsr*, *frm*, and *fflags*. When an extension's status is set to Off, any instruction that attempts to read or write the corresponding state will cause an illegal instruction exception. When the status is Initial, the corresponding state should have an initial constant value [1].

Status	FS Meaning
0	Off
1	Initial
2	Clean
3	Dirty

Table 5.2: mstatus.FS status and meaning

Therefore, by changing the status of *mstatus.FS* to Initial, the floating point instruction should be properly handled by the SAIL model. To conclude, the issues arises since the default state of other implementations is different from the default state of SAIL-RISCV. It alarms the necessity to standardize the initial state of the machine as its difference may vastly affect the result of the final comparison.

To address the problem, the concept of a reset vector is introduced. Usually, in the test assembly, the reset vector refers to the memory location where the program execution begins after a system reset. But in this context, we treat it as the instruction sequence to standardize the initial state of the machine. The reset vector is inserted and executed before the test instruction sequence.

The first two instructions in the reset vector of SAIL-RISCV shown in Appendix F serve the purpose of disabling the floating point instruction trap discussed above. The next instruction, "fscsr zero", aligns the dynamic rounding mode of floating point instruction to "Round to Nearest, ties to Even". The last two sets of instructions, "li" and "fcvt.d.w" ensures that the initial value of integer and floating point registers is zero.

Chapter 6

Evaluation

The chapter below evaluates the performance of the RISC-V fuzzers and details the testing method. Section 6.1 and 6.2 attempts to quantitatively measure the performance of different versions of fuzzer and compare it to the official ISA test suite. Section 6.3 qualitatively examines the test case generated from the fuzzer and analyses its ability to discover bugs.

All the testing was performed on the 2018 MacBook Pro Model with 2.3GHz Quad-Core Intel Core i5 and 8GB of RAM.

6.1 Fuzzer Comparison

The following section describes the evaluation metric, experimental setup and the result of the experiments on the three versions of fuzzer detailed in Chapter 4.

6.1.1 Experimental setup

For each version of the fuzzer, a fixed set of 10 randomly generated valid test cases was provided as the initial corpus. The fuzzing campaign lasted for five hours and the statistics were collected for every hour. The campaign was repeated three times in total to reduce the variance of the result.

6.1.2 Evaluation metrics

The following list details the evaluation metrics we considered and the method of obtaining them.

- Test case generated. More test cases generated usually indicate a higher possibility of detecting more bugs with the test suite. It was obtained by tracking the number of files in the corpus.
- Instruction covered in the G extension. It tracks whether the mutations cover all testable instructions, similar to functional coverage. The statistics are obtained by capturing all the instructions in the corpus using *disasm* and filtering all the pseudo-instructions which could be generated by the Spike dissembler and lead to an inaccurate count.
- Successful compilation rate. In the fuzzer loop, the return code of the assembly compilation was recorded. Return code 0 would be treated as successful compilation while return code 256 would be treated as failed compilation due to unrecognized instructions from the disassembly of invalid data. It measures the competence of the fuzzer for performing valid mutation.
- Total branch and line coverage. It measures how much Spike was tested and the representativeness of the test suite. To obtain the coverage information, we first compiled the Spike library with '-fprofile-instr-generate -fcoverage-mapping' coverage flag. Next, during the linkage between the fuzzer and library, we need to provide a stand-alone fuzz target main function for accepting the test cases from the corpus as the input of the *LLVMFuzzerTestOneInput* function. The stand-alone main function is referenced from the official llvm repository example [34]. Then, after running the binary, a raw profile file containing the coverage information is generated and needs to be indexed by using *llvm-profdata merge* command. Afterwards, the statistics were collected from the generated coverage report using *llvm-cov report*. All the

steps are detailed in LLVM Source-based Code Coverage website [35] and all the commands are contained in *get_fuzz_coverage.sh*.

In addition to total branch and line coverage, we would like to obtain the unique branch and line coverage to discover how complementary different version test suites are. The process of obtaining unique coverage is similar to obtaining the total coverage. Instead of generating the coverage report, we exported the data using *llvm-cov export* to a file format, 'lcov'. Then, we used a Python script, *compare_coverage.py* to extract the unique line number and branch number pairs from the covered branches (branch coverage) and covered line numbers (line coverage). The unique identification allows us to deduce the unique coverage for each fuzzer and also their intersection.

6.1.3 Test case generation

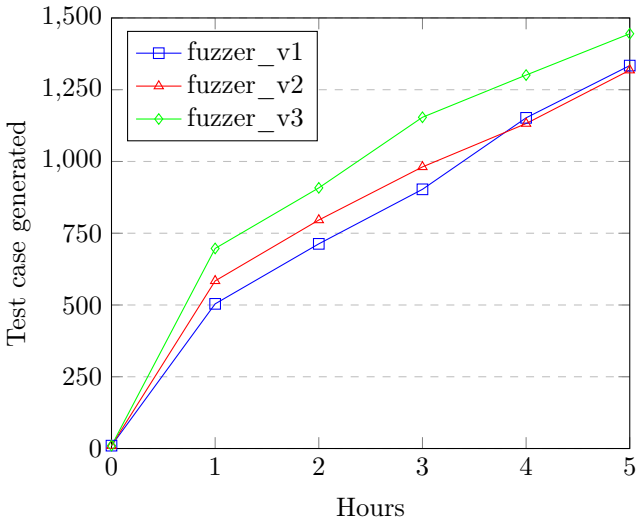

Figure 6.1: Test case generated by different version of fuzzer over 5 hours

As we could see in Figure 6.1, version 3 has the most test case generated while version 1 and version 2 has a similar growing trend. It may suggest version 3 fuzzer with more valid mutation operations successfully generates more variety of test cases compared to the other two versions.

6.1.4 Instruction coverage

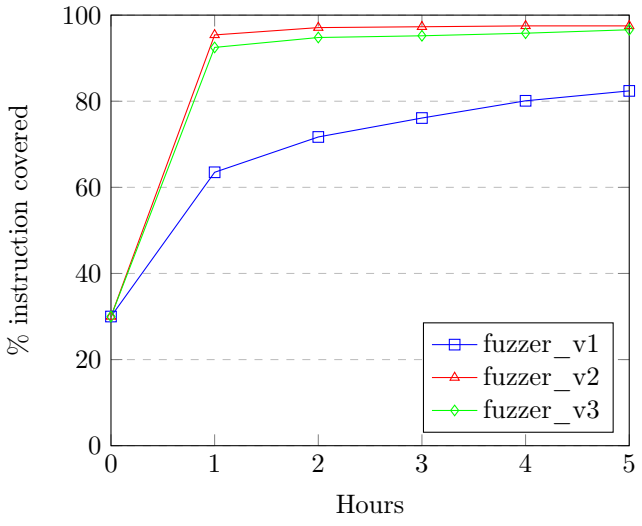

Figure 6.2: % instruction covered in G extension by different version of fuzzer over 5 hours

Presented in Figure 6.2, both version 2 and 3 have over 95% of instruction coverage while version 1 has around 82%. Surprisingly, version 1 could obtain quite a high instruction coverage given no extra human input except for the initial corpus. Nonetheless, the custom mutator in version 2 and version 3 allows the fuzzer to test more instructions than version 1 in the same time constraint.

6.1.5 Successful compilation rate

Fuzzer	Success	Fail	Success %
v1	85783	97803	47
v2	65869	15045	81
v3	69785	3953	90

Table 6.1: Successful compilation rate of different version of fuzzers

As shown in table 6.1, version 3 has the highest successful compilation rate of 90 % while version 1 has the lowest successful compilation rate of 81%. The introduction of the custom mutator significantly improves the quality of the test case illustrated by the large increase in success rate(34%) from version 1 to version 2. This is expected as the custom mutation increases the probability that the mutated inputs will be valid. The more type-aware custom mutator from version 3 further enhances the validity of the test case but is less significant compared to the previous version.

6.1.6 Line and branch coverage

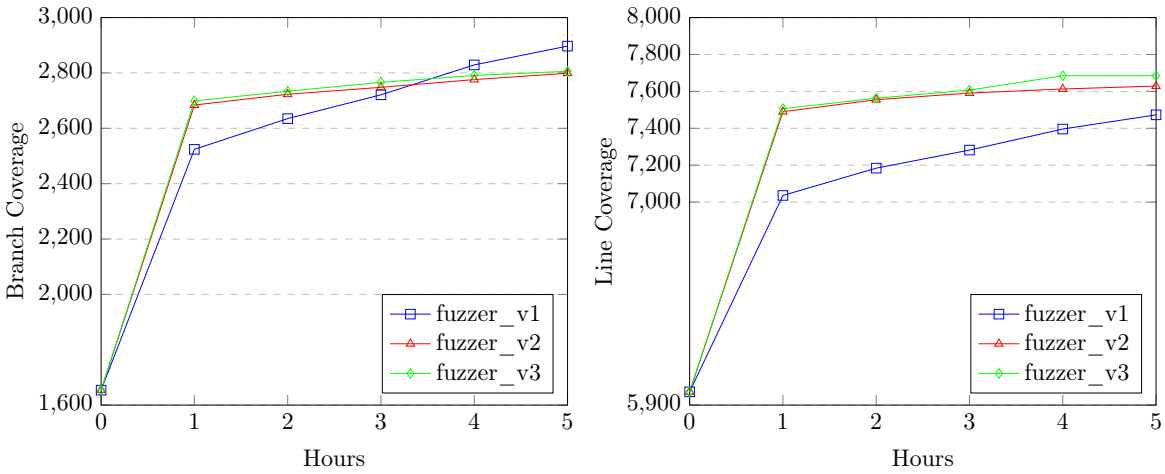

Figure 6.3: Total Line and Branch Coverage of different version of fuzzer over 5 hours

As shown in Figure 6.1.6, the line coverage of version 2 and version 3 is similar, around 7600 after 5 hours while version 1 is slightly lower at around 7400. It may suggest that the extra mutation operations and enhancement in version 3 fuzzer are not very effective to increase coverage compared to version 2. Also, we can observe saturation after the first hour of fuzzing for version 2 and version 3 while version 1 steadily increases its branch and line coverage. Most surprisingly, version 1 overtakes version 2 and version 3 in branch coverage at the fourth hour. It prompts us for a more careful investigation of the coverage on individual files. Then, we discovered version 1 has 180 and 220 more branch coverage than version 2 and 3 respectively on the file 'disasm.cc', which is used to disassemble the mutated data. However, since version 1 has no custom mutator and restrictions on mutation, the mutated data from version 1 would be more likely to trigger branches of outputting other extension instructions. Hence, it is unfair to include the extra coverage from this file and also from other unrelated instruction implementation files. Therefore, we have recalculated the total

branch and line coverage that excluded those unrelated files, shown in Table 6.2. As expected, version 3 has the highest branch and line coverage out of three versions of fuzzers.

In the following discussion, we would only include the coverage information for selected files. The complete list of included files could be found in Appendix G.

Fuzzer	Branch coverage	Line coverage
v1	2160	8248
v2	2228	8441
v3	2299	8498

Table 6.2: Total Line and Branch Coverage of different version of fuzzer after 5 hours for selected files (by excluding disassembly files and instruction implementation files that are not included in G extension)

6.1.7 Unique coverage

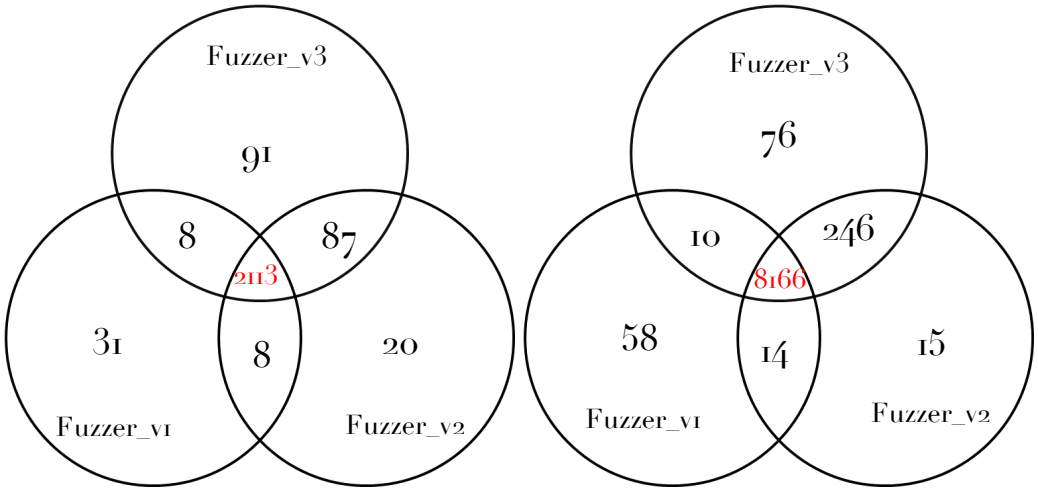

Figure 6.4: Venn diagrams of comparison between different version test suite of branch(left) and line(right) coverage after 5 hours for selected files (same selection criteria mentioned in Table 6.2)

The Venn diagrams in Figure 6.4 shows that there are 2113 common branch coverage and 8166 common line coverage for version 1, 2 and 3. It coverts to over 90 % of coverage overlap. Also, version 3 has the most unique branch and line coverage. Therefore, it suggests less complementarity between versions of the test suite and would be sufficient to pick version 3 to represent all of them. Hence, version 3 test suite will be chosen to compare with the official ISA test suite in the following section.

6.2 Comparison between fuzzer and official test suite

The official test suite chosen for this comparison comes from riscv-test repository [36]. It contains 230 test files filtered to only test the G extension instructions. Version 3 test suite is selected for comparison with the official test suite. The fuzzer was run for 24 hours and the test suite generated contains 5454 test cases covering 156 instructions.

6.2.1 Line and branch coverage

As shown in table 6.3, version 3 test suite achieves slightly lower total branch coverage (2578 vs 2518) but higher total line coverage (8986 vs 9360) compared to the official test suite. We can also group the files included into three main file groups: Instruction implementation, Spike and External library. The official test suite has better coverage in the instruction implementation and

	Official test suite		v3 test suite	
	Branch coverage	Line coverage	Branch coverage	Line coverage
Instruction implementation	641	2071	603	2080
Spike	1034	4336	1084	4732
External library	903	2579	831	2548
Total	2578	8986	2518	9360

Table 6.3: Total line and branch coverage of official ISA test suite and version 3 test suite for selected files (same selection criteria mentioned in Table 6.2)

external library file group while version 3 test suite has better coverage in the Spike file group. The result reveals that the manually created test suite is more capable of testing different kinds of corner cases than the fuzzer-created ones but by a small margin. Detail coverage of file groups and individual files could be found in Appendix G.

During the investigation of the coverage information, we discovered that the official test suite has no coverage on instructions 'lr.d' and 'sc.d' which is included in the RV64A extension. The author of the test suite may consider the 32-bit version of the instruction, 'lr.w' and 'sc.w', which are covered in the test suite, would behave the same compared to the 64-bit version. However, to be on the safe side, we suggest including the test cases for 'lr.d' and 'sc.d' in the official test suite.

6.2.2 Unique coverage

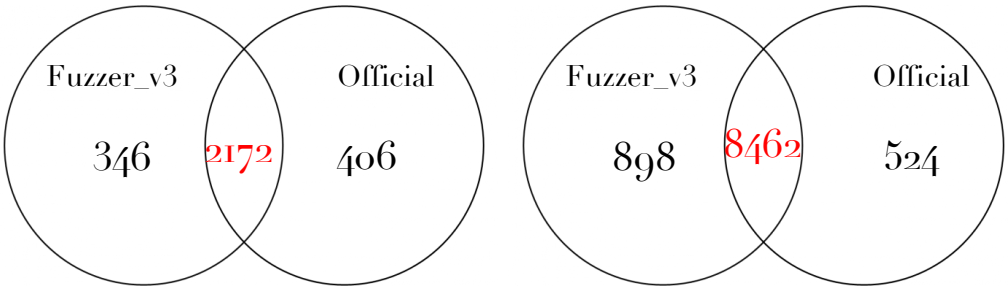

Figure 6.5: Venn diagram of comparison between version 3 and official ISA test suite of branch (left) and line (right) coverage for selected files (same selection criteria mentioned in Table 6.2)

Shown in Figure 6.5, version 3 test suite has 346 unique branch coverage and 898 unique line coverage while the official ISA test suite has higher unique branch coverage of 406 but lower unique line coverage of 524. This also displays complementarity between the methods: if the official ISA test suite includes the test suite generated from version 3 fuzzer, it would lead to 13.4% increase in total branch coverage and 10% increase in total line coverage.

6.3 Test case Examination

This section of evaluation focuses on test case examination and assessing the bug-finding ability of the fuzzer and its generated test cases. More specifically, the advanced fuzzer [Section 4.3] and its generated test cases are being evaluated. The fuzzer is run for 24 hours before the test cases are collected to perform fuzzing by proxy [Chapter 5].

To summarize, no crashes were found in any of the tested implementations. Furthermore, no bugs were discovered for Spike, FORVIS and SAIL-RISCV. However, we discovered and reported a bug regarding the incorrect handling of floating point instructions in RVEMU.

In the last part of this section, we would like to discuss some of the rejected test cases.

6.3.1 Bug: RVEMU-F Extension Instructions

The first and only bug detected in the fuzzing by proxy stage is related to floating point operations handling in RVEMU. The same type of difference in floating point register value was found multiple

times in various test cases. The most simplified test cases containing the bug are displayed in the following code snippet 6.1 and 6.2.

```
Asm:
fmsub.s ft1, ft9, fa6, ft0

Difference of floating point register
values:
Reg_name Spike            RVEMU
f1/ft1   nan              0
         FORVIS
         ffffffff7fc00000
         SAIL
         ffffffff7fc00000
```

Listing 6.1: RVEMU FMSUB.S

```
Asm:
fclass.s s0, fa3

Difference of register values:
Reg_name Spike            RVEMU
x8/s0/fp 200              4
         FORVIS           SAIL
         200              200
```

Listing 6.2: RVEMU bug output cont.

The instructions in test case 6.1 test the floating point arithmetic operation of the implementation. Specifically, *FMSUB.S rd, rs1, rs2, rs3* performs single-precision fused multiply addition. The implementation could be described by the following equation:

$$f[rd] = f[rs1] * f[rs2] - f[rs3]$$

The end register state of *f1*, the destination register of the instruction is NaN for Spike, FORVIS and SAIL-RISCV as 0xFFFF FFFF 7FC0 0000 represents quiet NaN for double precision value. And only RVEMU's *f1* shows zero. The correct behaviour described in the specification involved the concept of NaN-boxing [1]. Given the single-precision instruction is considered as a 32-bit operation and the double-precision instruction is considered as 64-bit:

> Apart from transfer operations described in the previous paragraph, all other floating-point operations on narrower n-bit operations, n< FLEN(width of floating point registers, 64-bit in our case) , check if the input operands are correctly NaN-boxed, i.e., all upper FLEN - n bits are 1. If so, the n least-significant bits of the input are used as the input value, otherwise the input value is treated as an n-bit canonical NaN.

As we have set the value of all the floating point registers to 0x0 before the execution of the test case using the reset vector, the input operand should be treated as NaN as mentioned above. Therefore, the value of the destination register, *f1* should be NaN instead of zero.

Similar bugs could be found in the following arithmetic F-extension instructions:

- fadd/sub.s
- fmul.s
- fmadd/sub.s
- fnmadd/sub.s

Test case 6.1 shows the error of another type of RVEMU floating point instruction. In this case, *FCLASS.S* examines the value in the floating point register, *fa3* and writes to integer register *s0*, a 10-bit mask that indicates the class of the floating point number. Table 6.4 shows the format of the relevant mask shown in the output. The corresponding bit will be set if the property is true and clear otherwise.

rd bit	Meaning
2	rs1 is a negative subnormal number
9	rs1 is a quiet NaN

Table 6.4: FCLASS rd bit meaning

RVEMU claims that the value in *fa3* is a negative subnormal number while Spike, FORVIS and SAIL-RISCV disagree and treat the value as quiet NaN. Similar to the case above, the value

of *fa3* have been set to zero in double precision. Therefore, by applying the NaN-boxing rule, it should be observed as quiet NaN for the input value of single precision instruction .

A GitHub issue was created to report the found bug [9]. However, at the time of writing, the author of RVEMU has not responded to the issue yet.

6.3.2 Rejected test case

```
Asm:
sc.w    s1, s4, (s10)
remu    s0, s5, a7
subw    tp, t5, a0
amominu.w a7, sp, (tp)

Difference of register values:
Reg_name Spike           RVEMU           FORVIS          SAIL
x9/s1    0               1               0               1
```

Listing 6.3: SC.W

However, not every test case that displays differences is considered as a bug. For example, in test case 6.3, *x9* register value is one for RVEMU and SAIL but zero for Spike and FORVIS. The difference in result could be traced back to the *SC.W* instruction with the destination register of *x9*.

SC.W rd, rs1, rs2 write a word in *rs2* to the address in *rs1*, provided a valid reservation still exists on that address. If the write is successful, *rd* would be written with zero. Otherwise, a non-zero code value be written instead. It is usually used after *LR.W* which loads a word from the address in a register and registers a reservation on the memory address. The combination of two atomic extension instructions allows RISC-V harts atomically read-modify-write memory to support synchronization between multiple of them running in the same memory space.

As there is no reservation of memory address by *LR.W* before *SC.W* in the test case, the instruction is expected to return a non-zero value to the *rd* register. However, Spike and FORVIS did not agree and returned zero instead. With further inspection, *SC.W* is trapped by both implementations.

The trap in FORVIS is caused by an invalid address as we have initialized the value of registers to be zero, which is not included in the memory region of FORVIS. By replacing the value of the store address in the *rs1* with a valid one, we retrieved a non-zero return from FORVIS.

The investigation of Spike is much more challenging as simply changing the memory address does not resolve the issue. After investigating the issue online, we discovered a reply from the contributor of RISC-V Proxy Kernel (PK), the execution environment used in the Spike fuzzing, that 'PK has a demand-paged virtual memory system, and so the addresses in the program don't directly correspond to physical addresses.'[37] This explains why Spike still traps the instruction even with different storing addresses. To investigate the behaviour of Spike without traps, we decided to use a test case from the official test suite since it would be difficult to test the behaviour with the PK environment. The test case we utilized is from test 2 of "rv64ua/lrsc.S"

```
//make sure that sc without a
    reservation fails.
TEST_CASE( 2, a4, 1, \
  la a0, foo; \
  li a5, 0xdeadbeef; \
  sc.w a4, a5, (a0); \
)
```

Listing 6.4: Test 2 of rv64ua/lrsc.S in riscv-tests [38]

```
Asm:
auipc   tp, 0x7abf0

Difference of register values:
Reg_name Spike           RVEMU
x4/tp    7ac00276        fabf00d0
         FORVIS          SAIL
         fabf00ce        fabf00d8
```

Listing 6.5: AUIPC

As shown in code snippet 6.4, the test assembly is similar to our test case and is responsible for testing the failure of *SC.W* by checking the *rd* register, *a4* has the value one after the test assembly. After compiling the test case, we can feed the binary directly to Spike without PK since the official test suite test program would set up its virtual testing environment. We obtained the

value of *a4* through the debug commands similar to the method detailed in section 5.1 and found out Spike behaves as expected.

Another rejected test case is shown in code snippet 6.5. The output value of *x4* for RVEMU, FORVIS and SAIL-RISCV is quite close while Spike's value is much lower. The instruction in the test case, *AUIPC rd, imm* forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the program counter, then places the result in register *rd*. The implementation could be described by the following equation:

$$x[rd] = pc + sign\_extend(immediate[31:12] \ll 12)$$

The main problem of this test case is the result of instruction depends on the value of the program counter at the execution time, which effectively is the address of the instruction. However, the starting address of the test case in the binary input for each implementation is different due to the varying size of the reset vector. Moreover, only the starting address of the text section, which stores the instructions of the program, of the Spike binary is not specified during the compilation stage. Therefore, it causes a large gap between the result of Spike and other implementations.

A possible solution would be to equalize the length of the reset vector using multiple *nop* and standardize the starting address of the text section of Spike binary by passing the compiler flag '-Wl,-Ttext=0x80000000'.

These rejected test cases again display the difficulty in standardizing the execution environment, which is briefly discussed in section 5.2 and the possibility of future improvement on the fuzzing by proxy system.

Chapter 7

Conclusion

In this project, we have successfully created a RISC-V fuzzer with Spike by first making Spike fuzz-friendly and integrating it with libFuzzer. Then, we created two variations of the custom mutator for RISC-V: a basic version which is capable of basic mutation such as switching the opcode of the instruction and an advanced version which has the capability of a decoder to perform more type-aware mutations. Including the default mutator variation, we developed three variations of the RISC-V fuzzer in total. Next, we performed fuzzing by proxy by utilizing the test cases generated during the fuzzing process to test the other three RISC-V implementations: RVEMU, FORVIS and SAIL-RISCV which accept the same binary input format. The whole process of fuzzing by proxy is automated by a bash script and the test result is presented in a human-readable format.

The first part of the evaluation compares the performance of different variations of fuzzer. As expected, the fuzzer with the advanced custom mutator outperforms other variations in terms of the number of test cases generated, instruction coverage, successful compilation rate and code coverage. However, the performance gain in some of the areas may not justify the manual effort put into building the advanced custom mutator. We are also surprised by the performance of the fuzzer with the default mutator since it is built with little manual input. In addition, we have compared the code coverage of Spike between the official RISC-V ISA test suite and the test suite generated by our RISC-V fuzzer. Although the result shows the test suite generated by our fuzzer covers slightly fewer branches in Spike compared with the official test suite, we discovered a potential missed instruction test case for the official test suite and unique coverage of our test suite that could lead to 13.4% increase in total branch coverage and 10% increase in total line coverage by combining the two test suite. The second part of the evaluation introduces the single-precision floating point instruction bug found in RVEMU during the fuzzing by proxy stage. Additionally, we also reflected on the test case that we rejected due to differences in the execution environment.

In summary, this project demonstrates the bug-finding potential of the fuzzer-generated test suite with a similar code coverage level compared to the official test suite. Moreover, we suggest using fuzzer generated test suite as a supplement to the official test suite to increase the chance of discovering more bugs. Fuzzing by proxy has also shown to be effective in exposing more bugs with differential testing. However, the setup of the testing has proven to be difficult and remains a challenge to be solved.

7.1 Future Work

- **Better opcode mutation and custom crossover mutation**
  Although in the evaluation section, the fuzzer with advanced custom mutator achieves over 95% instruction coverage in G extension, it would be ideal to achieve 100% instruction coverage to ensure every instruction is tested. One method would be to add a restriction in the instruction insertion mutation to insert every instruction in the set before randomly generating new instructions. Another method would be to revamp the custom mutator as the current model simply consider the 'opcode' section dictates the instruction. However, it could also be controlled by 'funct3' and 'funct7' sections depending on the type of instructions. By accounting for those sections, the fuzzer may achieve higher instruction coverage. Furthermore, the crossover mutation for RISC-V has not been implemented in the custom mutator. libFuzzer provides an API called *LLVMFuzzerCustomCrossOver* that allows the users to de-

fine their custom crossover mutation. It would be interesting to investigate whether this new mutation could lead to more coverage or bugs discovered.

- **Custom mutator: other RISC-V extensions**.
  Currently, the custom mutator only generates instructions in the G extension. However, compressed instruction(C) extension and vector(V) extension instruction are becoming more and more popular with the increasing requirement of high-performance computing and embedded system. It will be useful to automatically generate the test suite that covers instructions in both extensions using fuzzers and custom mutator. It would be also interesting to investigate the performance of the default mutator and the custom mutator since the vector extension adds around 400 potential instructions on top of existing instruction sets(around 160 instructions). The manual cost of extending the custom mutator may not justify its performance improvement.

- **Better Fuzzing by proxy**
  In fuzzing by proxy, we usually would only integrate one implementation with the fuzzer for generating test cases to test other implementations. Yet, we could also consider integrating multiple implementations instead to generate better test suites. For example, in this project, besides Spike, SAIL-RISCV builds its emulator in C. Hence, we could integrate SAIL-RISCV with libFuzzer and generate test cases. By merging the test case generated by each fuzzer, we may obtain a better test suite to test RISC-V implementations including Spike and SAIL-RISCV themselves.

- **Standardization of execution environment**
  As discussed in section 5.2 and evaluation, standardization of the execution environment has been a huge challenge in evaluating the output difference from fuzzing by proxy. We later discovered the official test suite code base [36] might already contain useful functions, macros and templates for building standardized RISC-V ISA unit tests. Unfortunately, due to time limitations, the option is not fully explored. It would be a great improvement for the fuzzing by proxy system if the templates help to solve the problem.

7.2 Ethical Consideration

This project aims to generate high-coverage test suites for the RISC-V ISA to strengthen the verification process of RISC-V Implementation. However, in the process, it may lead to the discovery of critical bugs in the software that are used for fuzzing by proxy, as well as in ISA descriptions. Therefore, it is possible for bugs and exploits discovered by these test cases to be misused by a malicious attacker. If these bugs are related to software used for fuzzing by proxy, the attacker may only be able to exploit computers which are running these softwares. However, if these bugs are related to RISC-V ISA, an attacker may be able to implement them to attack any devices with RISC-V cores and cause a much greater impact. Thus, it is important that vendors patch the bug promptly to avoid targeted attacks.

Generating test suites using fuzzing tools may have an environmental impact with a prolonged period of fuzzing. The "brute-force" approach could result in high energy usage on a large scale as it involves generating and computing with an enormous number of data inputs. Hence, the test case generation and testing process should be as efficient as possible to avoid wasting energy.

Appendix A

RV32I Instruction Set

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Figure A.1: RV32I Instruction Set [1]

Appendix B

Spike Debugging command

The basic usage of the Spike debugging command could be found in Spike Github Readme [5].
The value *end_pc* is replaced by the address of the last nop instruction of the binary, which indicate the end of test case.
reg command outputs the integer register value with the hartid and the register name as the arugment.
Similarly, *freg{s|d}* command outputs the floating point register value. *s* interprets the value as single precision floating point while *d* interprets the value as double precision floating point.
q quits the debugging terminal.

```
until pc 0 end_pc
reg 0 ra
reg 0 sp
reg 0 gp
reg 0 tp
reg 0 t0
reg 0 t1
reg 0 t2
reg 0 s0
reg 0 s1
reg 0 a0
reg 0 a1
reg 0 a2
reg 0 a3
reg 0 a4
reg 0 a5
reg 0 a6
reg 0 a7
reg 0 s2
reg 0 s3
reg 0 s4
reg 0 s5
reg 0 s6
reg 0 s7
reg 0 s8
reg 0 s9
reg 0 s10
reg 0 s11
reg 0 t3
reg 0 t4
reg 0 t5
reg 0 t6
```

Listing B.1: Spike Debugging command

```
fregd 0 ft0
fregd 0 ft1
fregd 0 ft2
fregd 0 ft3
fregd 0 ft4
fregd 0 ft5
fregd 0 ft6
fregd 0 ft7
fregd 0 fs0
fregd 0 fs1
fregd 0 fa0
fregd 0 fa1
fregd 0 fa2
fregd 0 fa3
fregd 0 fa4
fregd 0 fa5
fregd 0 fa6
fregd 0 fa7
fregd 0 fs2
fregd 0 fs3
fregd 0 fs4
fregd 0 fs5
fregd 0 fs6
fregd 0 fs7
fregd 0 fs8
fregd 0 fs9
fregd 0 fs10
fregd 0 fs11
fregd 0 ft8
fregd 0 ft9
fregd 0 ft10
fregd 0 ft11
q
```

Listing B.2: Spike Debugging command cont.

Appendix C

Example output of RISC-V Implementations

C.1 RVEMU

Example output of RVEMU: first part show the value of integer registers in hexadecimal, followed by floating point registers in decimal and control and status registers in hexadecimal.

```
pc: 0x0, trap Fatal
--------------------------------------------------------------------------------

x00(zero)=            0x0 x01( ra )=           0x0 x02( sp )=           0x0
x03( gp )=            0x0
x04( tp )=            0x0 x05( t0 )=           0x0 x06( t1 )=           0x0
x07( t2 )=            0x0
x08( s0 )=            0x0 x09( s1 )=           0x0 x10( a0 )=           0x0
x11( a1 )=            0x0
x12( a2 )=            0x0 x13( a3 )=           0x0 x14( a4 )=           0x4
x15( a5 )=            0x0
x16( a6 )=            0x0 x17( a7 )=           0x0 x18( s2 )=           0x0
x19( s3 )=            0x0
x20( s4 )=            0x0 x21( s5 )=           0x0 x22( s6 )=           0x0
x23( s7 )=            0x0
x24( s8 )=            0x0 x25( s9 )=           0x0 x26( s10)=           0x0
x27( s11)=            0x0
x28( t3 )=            0x0 x29( t4 )=           0x0 x30( t5 )=           0x0
x31( t6 )=            0x0
--------------------------------------------------------------------------------
f00( ft0)=      0.00000000 f01( ft1)=     0.00000000 f02( ft2)=     0.00000000
f03( ft3)=      0.00000000
f04( ft4)=      0.00000000 f05( ft5)=     0.00000000 f06( ft6)=     0.00000000
f07( ft7)=      0.00000000
f08( fs0)=      0.00000000 f09( fs1)=     0.00000000 f10( fa0)=     0.00000000
f11( fa1)=      0.00000000
f12( fa2)=      0.00000000 f13( fa3)=     0.00000000 f14( fa4)=     0.00000000
f15( fa5)=      0.00000000
f16( fa6)=      0.00000000 f17( fa7)=     0.00000000 f18( fs2)=     0.00000000
f19( fs3)=      0.00000000
f20( fs4)=      0.00000000 f21( fs5)=     0.00000000 f22( fs6)=     0.00000000
f23( fs7)=      0.00000000
f24( fs8)=      0.00000000 f25( fs9)=     0.00000000 f26(fs10)=     0.00000000
f27(fs11)=      0.00000000
f28( ft8)=      0.00000000 f29( ft9)=     0.00000000 f30(ft10)=     0.00000000
f31(ft11)=      0.00000000
--------------------------------------------------------------------------------
```

```
mstatus=          0x1800  mtvec=             0x0   mepc=             0x4
 mcause=             0x1 medeleg=            0x0 mideleg=            0x0
sstatus=             0x0  stvec=             0x0   sepc=             0x0
 scause=             0x0 sedeleg=            0x0 sideleg=            0x0
ustatus=             0x0  utvec=             0x0   uepc=             0x0
 ucause=             0x0
--------------------------------------------------------------------------------
pc: 0x0
```

Listing C.1: RVEMU example output(last column of register output is wrapped)

C.2 FORVIS

Example output of FORVIS: first part show the value of integer registers, followed by floating point registers and control and status registers in hexadecimal.

```
inum:74 pc 0x800000cc instr 0x_0a8d_0d13 priv 3 ADDI 26 26 168
  RV64 pc:800000d0 priv:3
  x0      ...._...._...._...0 ra ...._...._...._...0 sp ...._...._...._...0
  gp  ...._...._...._...0
  x4/tp   ...._...._...._...0 t0 ...._...._...._...0 t1 ...._...._...._...0
  t2  ...._...._...._...0
  x8/s0/fp ...._...._...._...0 s1 ...._...._...._...0 a0 ...._...._...._...0
  a1  ...._...._...._...0
  x12/a2  ...._...._...._...0 a3 ...._...._...._...0 a4 ...._...._...._...0
  a5  ...._...._...._...0
  x16/a6  ...._...._...._...0 a7 ...._...._...._...0 s2 ...._...._...._...0
  s3  ...._...._...._...0
  x20/s4  ...._...._...._...0 s5 ...._...._...._...0 s6 ...._...._...._...0
  s7  ...._...._...._...0
  x24/s8  ...._...._...._...0 s9 ...._...._...._...0 s10 ...._...._8000_00a8
  s11 ...._...._...._...0
  x28/t3  ...._...._...._...0 t4 ...._...._...._...0 t5 ...._...._...._...0
  t6  ...._...._...._...0
  f0/ft0  ...._...._...._...0 ft1  ...._...._...._...0 ft2  ...._...._...._...0
  ft3    ...._...._...._...0
  f4/ft4  ...._...._...._...0 ft5  ...._...._...._...0 ft6  ...._...._...._...0
  ft7    ...._...._...._...0
  f8/fs0  ...._...._...._...0 fs1  ...._...._...._...0 f10/fa0 ...._...._...._...0
  fa1    ...._...._...._...0
  f12/fa2 ...._...._...._...0 fa3  ...._...._...._...0 fa4  ...._...._...._...0
  fa5    ...._...._...._...0
  f16/fa6 ...._...._...._...0 fa7  ...._...._...._...0 f18/fs2 ...._...._...._...0
  fs3    ...._...._...._...0
  f20/fs4 ...._...._...._...0 fs5  ...._...._...._...0 fs6  ...._...._...._...0
  fs7    ...._...._...._...0
  f24/fs8 ...._...._...._...0 fs9  ...._...._...._...0 fs10 ...._...._...._...0
  fs11   ...._...._...._...0
  f28/ft8 ...._...._...._...0 ft9  ...._...._...._...0 ft10 ...._...._...._...0
  ft11   ...._...._...._...0
    mstatus:a00002000 mie:0 mip:880 medeleg:0 mideleg:0 mtvec:0 mepc:0 mcause:0
    mtval:0 mscratch:0 minstret:4a minstreth:0 mcycle:4a mcycleh:0 mcounteren:0
        mvendorid:0
    marchid:0 mimpid:0 mhartid:0 misa:800000000000112d tselect:0 data1:0 data2:0 data3:0
    dcsr:0 dpc:0 dscratch:0
    sstatus:200002000 sie:0 sip:0 sedeleg:0 sideleg:0 stvec:0 sepc:0 scause:0
    stval:0 sscratch:0 satp:0 scounteren:0
    ustatus:200002000 uie:0 uip:0 time:0 timeh:0 utvec:0 uepc:0 ucause:0
    utval:0 uscratch:0 cycle:4a cycleh:0 instret:4a instreth:0 fflags:0 frm:0
```

```
   fcsr:0
 Run_State_Running
```

Listing C.2: FORVIS example output(last column of register output is wrapped)

C.3 SAIL-RISCV

Example output of SAIL-RISCV: For each instruction execution, the output log will display the bytes read from the memory, the instruction decoded and the resulting operation (which is usually writing values into the destination register).

```
mem[X,0x00000000800000D0] -> 0xF8C7
mem[X,0x00000000800000D2] -> 0xBB9F
[74] [M]: 0x00000000800000D0 (0xBB9FF8C7) fmsub.d fa7, ft11, fs9, fs7, dyn
f17 <- 0x0000000000000000


mem[X,0x00000000800000D4] -> 0xF113
mem[X,0x00000000800000D6] -> 0xBD5F
[75] [M]: 0x00000000800000D4 (0xBD5FF113) andi sp, t6, 3029
x2 <- 0x0000000000000000


mem[X,0x00000000800000D8] -> 0xFC33
mem[X,0x00000000800000DA] -> 0x0324
[76] [M]: 0x00000000800000D8 (0x0324FC33) remu s8, s1, s2
x24 <- 0x0000000000000000


mem[X,0x00000000800000DC] -> 0x9753
mem[X,0x00000000800000DE] -> 0xE003
[77] [M]: 0x00000000800000DC (0xE0039753) fclass.s a4, ft7
x14 <- 0x0000000000000200


mem[X,0x00000000800000E0] -> 0xF8C7
mem[X,0x00000000800000E2] -> 0xBB9F
[78] [M]: 0x00000000800000E0 (0xBB9FF8C7) fmsub.d fa7, ft11, fs9, fs7, dyn
f17 <- 0x0000000000000000
```

Listing C.3: SAIL-RISCV example output

Appendix D

Spike error output

Example Spike error output with only the integer register value and program counter value shown. The exception raised is shown in the last line of the error log, 'User store segfault'.

```
z  0000000000000000 ra 0000000000000000 sp 0000000000000000 gp 0000000000000000
tp 0000000000000000 t0 0000000000000000 t1 0000000000000000 t2 0000000000000000
s0 0000000000000000 s1 0000000000000000 a0 0000000000000000 a1 0000000000000000
a2 0000000000000000 a3 0000000000000000 a4 0000000000000000 a5 0000000000000000
a6 0000000000000000 a7 0000000000000000 s2 0000000000000000 s3 0000000000000000
s4 0000000000000000 s5 0000000000000000 s6 0000000000000000 s7 0000000000000000
s8 0000000000000000 s9 0000000000000000 sA 0000000000000000 sB 0000000000000000
t3 0000000000000000 t4 0000000000000000 t5 0000000000000000 t6 0000000000000000
pc 000000000001026e va/inst ffffffffffffcc4 sr 8000000200006020
User store segfault @ 0xffffffffffffcc4
```

Listing D.1: Spike error output

Appendix E

Example output of *cmp_reg_output*

Example output of *cmp_reg_output*. The output includes:

- Filename
- Disassembled instructions
- Error output from implementation which facilitate the examination process
- Difference in integer register value (if any)
- Difference in floating point register value (if any)

```
Reading file: testcases/05_26/191f542e020cefd78989d684d14c315f28cadcda
Asm:
fnmsub.d ft4, fs3, ft4, ft10
fnmsub.s ft1, ft9, fa6, ft0
fdiv.d fs10, ft6, fs7
fnmsub.s ft1, ft9, fa6, ft0
fnmsub.d ft4, fs3, ft4, ft10
fmsub.s ft1, ft9, fa6, ft0
fnmsub.s ft1, ft9, fa6, ft0
fnmsub.d ft4, fs3, ft4, ft10
bbl loader
SAIL: Stopping program execution.
Difference of register values:
Reg_name Spike           RVEMU           FORVIS          SAIL
----------------------------------------
Difference of floating point register values:
Reg_name Spike           RVEMU           FORVIS          SAIL
f1/ft1   nan             0               ffffffff7fc00000 ffffffff7fc00000
----------------------------------------
```

Listing E.1: *cmp_reg_output* example output

Appendix F

Reset vector for FORVIS

Reset vector for FORVIS. For detailed explanation, please refer to section 5.2.

```
    li      a0, 8192
    csrw    mstatus, a0
    fscsr   zero
    li    x0,       0
    li    x1,       0
    li    x2,       0
    li    x3,       0
    li    x4,       0
    li    x5,       0
    li    x6,       0
    li    x7,       0
    li    x8,       0
    li    x9,       0
    li    x10,      0
    li    x11,      0
    li    x12,      0
    li    x13,      0
    li    x14,      0
    li    x15,      0
    li    x16,      0
    li    x17,      0
    li    x18,      0
    li    x19,      0
    li    x20,      0
    li    x21,      0
    li    x22,      0
    li    x23,      0
    li    x24,      0
    li    x25,      0
    li    x26,      0
    li    x27,      0
    li    x28,      0
    li    x29,      0
    li    x30,      0
    li    x31,      0
```

Listing F.1: Reset vector

```
    fcvt.d.w   f0, x0
    fcvt.d.w   f1, x0
    fcvt.d.w   f2, x0
    fcvt.d.w   f3, x0
    fcvt.d.w   f4, x0
    fcvt.d.w   f5, x0
    fcvt.d.w   f6, x0
    fcvt.d.w   f7, x0
    fcvt.d.w   f8, x0
    fcvt.d.w   f9, x0
    fcvt.d.w   f10, x0
    fcvt.d.w   f11, x0
    fcvt.d.w   f12, x0
    fcvt.d.w   f13, x0
    fcvt.d.w   f14, x0
    fcvt.d.w   f15, x0
    fcvt.d.w   f16, x0
    fcvt.d.w   f17, x0
    fcvt.d.w   f18, x0
    fcvt.d.w   f19, x0
    fcvt.d.w   f20, x0
    fcvt.d.w   f21, x0
    fcvt.d.w   f22, x0
    fcvt.d.w   f23, x0
    fcvt.d.w   f24, x0
    fcvt.d.w   f25, x0
    fcvt.d.w   f26, x0
    fcvt.d.w   f27, x0
    fcvt.d.w   f28, x0
    fcvt.d.w   f29, x0
    fcvt.d.w   f30, x0
    fcvt.d.w   f31, x0
```

Listing F.2: Reset vector cont.

Appendix G

Coverage comparison between fuzzer-generated test suite and official test suite

ISA represents the coverage of the official RISC-V ISA test suite
Fuzz represents the coverage of fuzzer
ISA-U represents the unique coverage of the official RISC-V ISA test suite
Fuzz-U represents the unique coverage of the fuzzer
Inter represents the intersection between the two test suites

G.1 Instruction implementation

Table G.1: Coverage comparison of instruction implementation

Filename	Branch Coverage					Line Coverage				
	ISA	**Fuzz**	**ISA-U**	**Fuzz-U**	**Inter**	**ISA**	**Fuzz**	**ISA-U**	**Fuzz-U**	**Inter**
add.cc	0	0	0	0	0	9	9	0	0	9
add.h	0	0	0	0	0	2	2	0	0	2
addi.cc	0	0	0	0	0	9	9	0	0	9
addi.h	0	0	0	0	0	2	2	0	0	2
addiw.cc	0	0	0	0	0	9	9	0	0	9
addiw.h	0	0	0	0	0	3	3	0	0	3
addw.cc	0	0	0	0	0	9	9	0	0	9
addw.h	0	0	0	0	0	3	3	0	0	3
amoadd_d.cc	1	1	0	0	1	9	9	0	0	9
amoadd_d.h	0	0	0	0	0	4	4	0	0	4
amoadd_w.cc	1	1	0	0	1	9	9	0	0	9
amoadd_w.h	0	0	0	0	0	3	3	0	0	3
amoand_d.cc	1	1	0	0	1	9	9	0	0	9
amoand_d.h	0	0	0	0	0	4	4	0	0	4
amoand_w.cc	1	1	0	0	1	9	9	0	0	9
amoand_w.h	0	0	0	0	0	3	3	0	0	3
amomax_d.cc	1	1	0	0	1	9	9	0	0	9
amomax_d.h	0	0	0	0	0	4	4	0	0	4
amomax_w.cc	1	1	0	0	1	9	9	0	0	9
amomax_w.h	0	0	0	0	0	3	3	0	0	3
amomaxu_d.cc	1	1	0	0	1	9	9	0	0	9
amomaxu_d.h	0	0	0	0	0	4	4	0	0	4
amomaxu_w.cc	1	1	0	0	1	9	9	0	0	9
amomaxu_w.h	0	0	0	0	0	3	3	0	0	3

Continued on next page

Filename	Branch Coverage					Line Coverage				
	ISA	Fuzz	ISA-U	Fuzz-U	Inter	ISA	Fuzz	ISA-U	Fuzz-U	Inter
amomin_d.cc	1	1	0	0	1	9	9	0	0	9
amomin_d.h	0	0	0	0	0	4	4	0	0	4
amomin_w.cc	1	1	0	0	1	9	9	0	0	9
amomin_w.h	0	0	0	0	0	3	3	0	0	3
amominu_d.cc	1	1	0	0	1	9	9	0	0	9
amominu_d.h	0	0	0	0	0	4	4	0	0	4
amominu_w.cc	1	1	0	0	1	9	9	0	0	9
amominu_w.h	0	0	0	0	0	3	3	0	0	3
amoor_d.cc	1	1	0	0	1	9	9	0	0	9
amoor_d.h	0	0	0	0	0	4	4	0	0	4
amoor_w.cc	1	1	0	0	1	9	9	0	0	9
amoor_w.h	0	0	0	0	0	3	3	0	0	3
amoswap_d.cc	1	1	0	0	1	9	9	0	0	9
amoswap_d.h	0	0	0	0	0	4	4	0	0	4
amoswap_w.cc	1	1	0	0	1	9	9	0	0	9
amoswap_w.h	0	0	0	0	0	3	3	0	0	3
amoxor_d.cc	1	1	0	0	1	9	9	0	0	9
amoxor_d.h	0	0	0	0	0	4	4	0	0	4
amoxor_w.cc	1	1	0	0	1	9	9	0	0	9
amoxor_w.h	0	0	0	0	0	3	3	0	0	3
and.cc	0	0	0	0	0	9	9	0	0	9
and.h	0	0	0	0	0	2	2	0	0	2
andi.cc	0	0	0	0	0	9	9	0	0	9
andi.h	0	0	0	0	0	2	2	0	0	2
auipc.cc	0	0	0	0	0	9	9	0	0	9
auipc.h	0	0	0	0	0	2	2	0	0	2
beq.cc	2	2	0	0	2	8	8	0	0	8
beq.h	0	0	0	0	0	3	3	0	0	3
bge.cc	2	2	0	0	2	8	8	0	0	8
bge.h	0	0	0	0	0	3	3	0	0	3
bgeu.cc	2	2	0	0	2	8	8	0	0	8
bgeu.h	0	0	0	0	0	3	3	0	0	3
blt.cc	2	2	0	0	2	8	8	0	0	8
blt.h	0	0	0	0	0	3	3	0	0	3
bltu.cc	2	2	0	0	2	8	8	0	0	8
bltu.h	0	0	0	0	0	3	3	0	0	3
bne.cc	2	2	0	0	2	8	8	0	0	8
bne.h	0	0	0	0	0	3	3	0	0	3
csrrc.cc	3	4	0	1	3	9	9	0	0	9
csrrc.h	0	0	0	0	0	8	8	0	0	8
csrrci.cc	3	3	1	1	2	9	9	0	0	9
csrrci.h	0	0	0	0	0	8	6	2	0	6
csrrs.cc	4	4	0	0	4	9	9	0	0	9
csrrs.h	0	0	0	0	0	8	8	0	0	8
csrrsi.cc	3	4	0	1	3	9	9	0	0	9
csrrsi.h	0	0	0	0	0	8	8	0	0	8
csrrw.cc	2	2	0	0	2	9	9	0	0	9
csrrw.h	0	0	0	0	0	5	5	0	0	5
csrrwi.cc	2	2	0	0	2	9	9	0	0	9
csrrwi.h	0	0	0	0	0	5	5	0	0	5
div.cc	7	4	3	0	4	9	9	0	0	9
div.h	0	0	0	0	0	10	10	0	0	10
divu.cc	3	3	0	0	3	9	9	0	0	9
divu.h	0	0	0	0	0	8	8	0	0	8
divuw.cc	3	3	0	0	3	9	9	0	0	9

Continued on next page

Filename	Branch Coverage					Line Coverage				
	ISA	**Fuzz**	**ISA-U**	**Fuzz-U**	**Inter**	**ISA**	**Fuzz**	**ISA-U**	**Fuzz-U**	**Inter**
divuw.h	0	0	0	0	0	9	9	0	0	9
divw.cc	3	3	0	0	3	9	9	0	0	9
divw.h	0	0	0	0	0	9	9	0	0	9
ebreak.cc	8	0	8	0	0	5	0	5	0	0
ebreak.h	0	0	0	0	0	8	0	8	0	0
ecall.cc	8	4	4	0	4	5	5	0	0	5
ecall.h	0	0	0	0	0	10	5	5	0	5
fadd_d.cc	9	8	1	0	8	9	9	0	0	9
fadd_d.h	0	0	0	0	0	6	6	0	0	6
fadd_s.cc	9	9	1	1	8	9	9	0	0	9
fadd_s.h	0	0	0	0	0	6	6	0	0	6
fclass_d.cc	3	3	0	0	3	9	9	0	0	9
fclass_d.h	0	0	0	0	0	4	4	0	0	4
fclass_s.cc	3	3	0	0	3	9	9	0	0	9
fclass_s.h	0	0	0	0	0	4	4	0	0	4
fcvt_d_l.cc	6	7	0	1	6	9	9	0	0	9
fcvt_d_l.h	0	0	0	0	0	7	7	0	0	7
fcvt_d_lu.cc	7	7	0	0	7	9	9	0	0	9
fcvt_d_lu.h	0	0	0	0	0	7	7	0	0	7
fcvt_d_s.cc	7	7	0	0	7	9	9	0	0	9
fcvt_d_s.h	0	0	0	0	0	6	6	0	0	6
fcvt_d_w.cc	6	6	0	0	6	9	9	0	0	9
fcvt_d_w.h	0	0	0	0	0	6	6	0	0	6
fcvt_d_wu.cc	6	6	0	0	6	9	9	0	0	9
fcvt_d_wu.h	0	0	0	0	0	6	6	0	0	6
fcvt_l_d.cc	11	8	3	0	8	9	9	0	0	9
fcvt_l_d.h	0	0	0	0	0	7	7	0	0	7
fcvt_l_s.cc	11	9	2	0	9	9	9	0	0	9
fcvt_l_s.h	0	0	0	0	0	7	7	0	0	7
fcvt_lu_d.cc	11	9	2	0	9	9	9	0	0	9
fcvt_lu_d.h	0	0	0	0	0	7	7	0	0	7
fcvt_lu_s.cc	11	9	2	0	9	9	9	0	0	9
fcvt_lu_s.h	0	0	0	0	0	7	7	0	0	7
fcvt_s_d.cc	7	8	0	1	7	9	9	0	0	9
fcvt_s_d.h	0	0	0	0	0	6	6	0	0	6
fcvt_s_l.cc	6	7	0	1	6	9	9	0	0	9
fcvt_s_l.h	0	0	0	0	0	7	7	0	0	7
fcvt_s_lu.cc	7	7	0	0	7	9	9	0	0	9
fcvt_s_lu.h	0	0	0	0	0	7	7	0	0	7
fcvt_s_w.cc	6	7	0	1	6	9	9	0	0	9
fcvt_s_w.h	0	0	0	0	0	6	6	0	0	6
fcvt_s_wu.cc	7	7	0	0	7	9	9	0	0	9
fcvt_s_wu.h	0	0	0	0	0	6	6	0	0	6
fcvt_w_d.cc	11	9	2	0	9	9	9	0	0	9
fcvt_w_d.h	0	0	0	0	0	6	6	0	0	6
fcvt_w_s.cc	11	9	2	0	9	9	9	0	0	9
fcvt_w_s.h	0	0	0	0	0	6	6	0	0	6
fcvt_wu_d.cc	11	9	2	0	9	9	9	0	0	9
fcvt_wu_d.h	0	0	0	0	0	6	6	0	0	6
fcvt_wu_s.cc	11	9	2	0	9	9	9	0	0	9
fcvt_wu_s.h	0	0	0	0	0	6	6	0	0	6
fdiv_d.cc	9	8	1	0	8	9	9	0	0	9
fdiv_d.h	0	0	0	0	0	6	6	0	0	6
fdiv_s.cc	9	9	0	0	9	9	9	0	0	9
fdiv_s.h	0	0	0	0	0	6	6	0	0	6

Continued on next page

Filename	Branch Coverage					Line Coverage				
	ISA	**Fuzz**	**ISA-U**	**Fuzz-U**	**Inter**	**ISA**	**Fuzz**	**ISA-U**	**Fuzz-U**	**Inter**
fence_i.cc	0	0	0	0	0	8	8	0	0	8
fence_i.h	0	0	0	0	0	2	2	0	0	2
fence.cc	0	0	0	0	0	8	8	0	0	8
feq_d.cc	6	5	1	0	5	9	9	0	0	9
feq_d.h	0	0	0	0	0	5	5	0	0	5
feq_s.cc	6	5	1	0	5	9	9	0	0	9
feq_s.h	0	0	0	0	0	5	5	0	0	5
fld.cc	1	1	0	0	1	9	9	0	0	9
fld.h	0	0	0	0	0	4	4	0	0	4
fle_d.cc	6	5	1	0	5	9	9	0	0	9
fle_d.h	0	0	0	0	0	5	5	0	0	5
fle_s.cc	6	6	0	0	6	9	9	0	0	9
fle_s.h	0	0	0	0	0	5	5	0	0	5
flt_d.cc	6	5	1	0	5	9	9	0	0	9
flt_d.h	0	0	0	0	0	5	5	0	0	5
flt_s.cc	6	6	0	0	6	9	9	0	0	9
flt_s.h	0	0	0	0	0	5	5	0	0	5
flw.cc	1	1	0	0	1	9	9	0	0	9
flw.h	0	0	0	0	0	4	4	0	0	4
fmadd_d.cc	10	11	1	2	9	9	9	0	0	9
fmadd_d.h	0	0	0	0	0	6	6	0	0	6
fmadd_s.cc	10	11	1	2	9	9	9	0	0	9
fmadd_s.h	0	0	0	0	0	6	6	0	0	6
fmax_d.cc	35	31	7	3	28	9	9	0	0	9
fmax_d.h	0	0	0	0	0	10	10	0	0	10
fmax_s.cc	35	37	1	3	34	9	9	0	0	9
fmax_s.h	0	0	0	0	0	10	10	0	0	10
fmin_d.cc	29	26	8	5	21	9	9	0	0	9
fmin_d.h	0	0	0	0	0	9	10	0	1	9
fmin_s.cc	29	37	0	8	29	9	9	0	0	9
fmin_s.h	0	0	0	0	0	9	10	0	1	9
fmsub_d.cc	10	9	1	0	9	9	9	0	0	9
fmsub_d.h	0	0	0	0	0	6	6	0	0	6
fmsub_s.cc	10	10	1	1	9	9	9	0	0	9
fmsub_s.h	0	0	0	0	0	6	6	0	0	6
fmul_d.cc	9	8	1	0	8	9	9	0	0	9
fmul_d.h	0	0	0	0	0	6	6	0	0	6
fmul_s.cc	9	8	1	0	8	9	9	0	0	9
fmul_s.h	0	0	0	0	0	6	6	0	0	6
fmv_d_x.cc	1	1	0	0	1	9	9	0	0	9
fmv_d_x.h	0	0	0	0	0	5	5	0	0	5
fmv_w_x.cc	1	1	0	0	1	9	9	0	0	9
fmv_w_x.h	0	0	0	0	0	4	4	0	0	4
fmv_x_d.cc	1	1	0	0	1	9	9	0	0	9
fmv_x_d.h	0	0	0	0	0	5	5	0	0	5
fmv_x_w.cc	1	1	0	0	1	9	9	0	0	9
fmv_x_w.h	0	0	0	0	0	4	4	0	0	4
fnmadd_d.cc	10	9	1	0	9	9	9	0	0	9
fnmadd_d.h	0	0	0	0	0	6	6	0	0	6
fnmadd_s.cc	10	9	1	0	9	9	9	0	0	9
fnmadd_s.h	0	0	0	0	0	6	6	0	0	6
fnmsub_d.cc	10	9	1	0	9	9	9	0	0	9
fnmsub_d.h	0	0	0	0	0	6	6	0	0	6
fnmsub_s.cc	10	9	1	0	9	9	9	0	0	9
fnmsub_s.h	0	0	0	0	0	6	6	0	0	6

Continued on next page

Filename	Branch Coverage					Line Coverage				
	ISA	Fuzz	ISA-U	Fuzz-U	Inter	ISA	Fuzz	ISA-U	Fuzz-U	Inter
fsd.cc	1	1	0	0	1	8	8	0	0	8
fsd.h	0	0	0	0	0	4	4	0	0	4
fsgnj_d.cc	5	5	0	0	5	9	9	0	0	9
fsgnj_d.h	0	0	0	0	0	4	4	0	0	4
fsgnj_s.cc	5	5	0	0	5	9	9	0	0	9
fsgnj_s.h	0	0	0	0	0	4	4	0	0	4
fsgnjn_d.cc	5	5	0	0	5	9	9	0	0	9
fsgnjn_d.h	0	0	0	0	0	4	4	0	0	4
fsgnjn_s.cc	5	5	0	0	5	9	9	0	0	9
fsgnjn_s.h	0	0	0	0	0	4	4	0	0	4
fsgnjx_d.cc	6	6	0	0	6	9	9	0	0	9
fsgnjx_d.h	0	0	0	0	0	4	4	0	0	4
fsgnjx_s.cc	6	6	0	0	6	9	9	0	0	9
fsgnjx_s.h	0	0	0	0	0	4	4	0	0	4
fsqrt_d.cc	8	8	0	0	8	9	9	0	0	9
fsqrt_d.h	0	0	0	0	0	6	6	0	0	6
fsqrt_s.cc	8	7	1	0	7	9	9	0	0	9
fsqrt_s.h	0	0	0	0	0	6	6	0	0	6
fsub_d.cc	9	8	1	0	8	9	9	0	0	9
fsub_d.h	0	0	0	0	0	6	6	0	0	6
fsub_s.cc	9	8	1	0	8	9	9	0	0	9
fsub_s.h	0	0	0	0	0	6	6	0	0	6
fsw.cc	1	1	0	0	1	8	8	0	0	8
fsw.h	0	0	0	0	0	4	4	0	0	4
jal.cc	0	0	0	0	0	9	9	0	0	9
jal.h	0	0	0	0	0	4	4	0	0	4
jalr.cc	0	0	0	0	0	9	9	0	0	9
jalr.h	0	0	0	0	0	4	4	0	0	4
lb.cc	0	0	0	0	0	9	9	0	0	9
lb.h	0	0	0	0	0	2	2	0	0	2
lbu.cc	0	0	0	0	0	9	9	0	0	9
lbu.h	0	0	0	0	0	2	2	0	0	2
ld.cc	0	0	0	0	0	9	9	0	0	9
ld.h	0	0	0	0	0	3	3	0	0	3
lh.cc	0	0	0	0	0	9	9	0	0	9
lh.h	0	0	0	0	0	2	2	0	0	2
lhu.cc	0	0	0	0	0	9	9	0	0	9
lhu.h	0	0	0	0	0	2	2	0	0	2
lr_d.cc	0	1	0	1	0	0	9	0	9	0
lr_d.h	0	0	0	0	0	0	4	0	4	0
lr_w.cc	1	1	0	0	1	9	9	0	0	9
lr_w.h	0	0	0	0	0	3	3	0	0	3
lui.cc	0	0	0	0	0	9	9	0	0	9
lui.h	0	0	0	0	0	2	2	0	0	2
lw.cc	0	0	0	0	0	9	9	0	0	9
lw.h	0	0	0	0	0	2	2	0	0	2
lwu.cc	0	0	0	0	0	9	9	0	0	9
lwu.h	0	0	0	0	0	3	3	0	0	3
mul.cc	2	2	0	0	2	9	9	0	0	9
mul.h	0	0	0	0	0	3	3	0	0	3
mulh.cc	2	2	0	0	2	9	9	0	0	9
mulh.h	0	0	0	0	0	5	5	0	0	5
mulhsu.cc	2	2	0	0	2	9	9	0	0	9
mulhsu.h	0	0	0	0	0	5	5	0	0	5
mulhu.cc	2	2	0	0	2	9	9	0	0	9

Continued on next page

Filename	Branch Coverage					Line Coverage				
	ISA	Fuzz	ISA-U	Fuzz-U	Inter	ISA	Fuzz	ISA-U	Fuzz-U	Inter
mulhu.h	0	0	0	0	0	5	5	0	0	5
mulw.cc	2	2	0	0	2	9	9	0	0	9
mulw.h	0	0	0	0	0	4	4	0	0	4
or.cc	0	0	0	0	0	9	9	0	0	9
or.h	0	0	0	0	0	2	2	0	0	2
ori.cc	0	0	0	0	0	9	9	0	0	9
ori.h	0	0	0	0	0	2	2	0	0	2
rem.cc	7	4	3	0	4	9	9	0	0	9
rem.h	0	0	0	0	0	10	10	0	0	10
remu.cc	3	3	0	0	3	9	9	0	0	9
remu.h	0	0	0	0	0	8	8	0	0	8
remuw.cc	3	3	0	0	3	9	9	0	0	9
remuw.h	0	0	0	0	0	9	9	0	0	9
remw.cc	3	3	0	0	3	9	9	0	0	9
remw.h	0	0	0	0	0	9	9	0	0	9
sb.cc	0	0	0	0	0	8	8	0	0	8
sb.h	0	0	0	0	0	2	2	0	0	2
sc_d.cc	0	1	0	1	0	0	9	0	9	0
sc_d.h	0	0	0	0	0	0	5	0	5	0
sc_w.cc	1	1	0	0	1	9	9	0	0	9
sc_w.h	0	0	0	0	0	4	4	0	0	4
sd.cc	0	0	0	0	0	8	8	0	0	8
sd.h	0	0	0	0	0	3	3	0	0	3
sh.cc	0	0	0	0	0	8	8	0	0	8
sh.h	0	0	0	0	0	2	2	0	0	2
sll.cc	0	0	0	0	0	9	9	0	0	9
sll.h	0	0	0	0	0	2	2	0	0	2
slli.cc	1	1	0	0	1	9	9	0	0	9
slli.h	0	0	0	0	0	3	3	0	0	3
slliw.cc	0	0	0	0	0	9	9	0	0	9
slliw.h	0	0	0	0	0	3	3	0	0	3
sllw.cc	0	0	0	0	0	9	9	0	0	9
sllw.h	0	0	0	0	0	3	3	0	0	3
slt.cc	0	0	0	0	0	9	9	0	0	9
slt.h	0	0	0	0	0	2	2	0	0	2
slti.cc	0	0	0	0	0	9	9	0	0	9
slti.h	0	0	0	0	0	2	2	0	0	2
sltiu.cc	0	0	0	0	0	9	9	0	0	9
sltiu.h	0	0	0	0	0	2	2	0	0	2
sltu.cc	0	0	0	0	0	9	9	0	0	9
sltu.h	0	0	0	0	0	2	2	0	0	2
sra.cc	0	0	0	0	0	9	9	0	0	9
sra.h	0	0	0	0	0	2	2	0	0	2
srai.cc	1	1	0	0	1	9	9	0	0	9
srai.h	0	0	0	0	0	3	3	0	0	3
sraiw.cc	0	0	0	0	0	9	9	0	0	9
sraiw.h	0	0	0	0	0	3	3	0	0	3
sraw.cc	0	0	0	0	0	9	9	0	0	9
sraw.h	0	0	0	0	0	3	3	0	0	3
srl.cc	0	0	0	0	0	9	9	0	0	9
srl.h	0	0	0	0	0	2	2	0	0	2
srli.cc	1	1	0	0	1	9	9	0	0	9
srli.h	0	0	0	0	0	3	3	0	0	3
srliw.cc	0	0	0	0	0	9	9	0	0	9
srliw.h	0	0	0	0	0	3	3	0	0	3

Continued on next page

Filename	Branch Coverage					Line Coverage				
	ISA	**Fuzz**	**ISA-U**	**Fuzz-U**	**Inter**	**ISA**	**Fuzz**	**ISA-U**	**Fuzz-U**	**Inter**
srlw.cc	0	0	0	0	0	9	9	0	0	9
srlw.h	0	0	0	0	0	3	3	0	0	3
sub.cc	0	0	0	0	0	9	9	0	0	9
sub.h	0	0	0	0	0	2	2	0	0	2
subw.cc	0	0	0	0	0	9	9	0	0	9
subw.h	0	0	0	0	0	3	3	0	0	3
sw.cc	0	0	0	0	0	8	8	0	0	8
sw.h	0	0	0	0	0	2	2	0	0	2
xor.cc	0	0	0	0	0	9	9	0	0	9
xor.h	0	0	0	0	0	2	2	0	0	2
xori.cc	0	0	0	0	0	9	9	0	0	9
xori.h	0	0	0	0	0	2	2	0	0	2
Total	641	603	72	34	569	2071	2080	20	29	2051

G.2 Spike

Filename	Branch Coverage					Line Coverage				
	ISA	**Fuzz**	**ISA-U**	**Fuzz-U**	**Inter**	**ISA**	**Fuzz**	**ISA-U**	**Fuzz-U**	**Inter**
abstract_device.h	0	0	0	0	0	1	1	0	0	1
abstract_interrupt _controller.h	0	0	0	0	0	1	1	0	0	1
arith.h	19	15	4	0	15	45	45	0	0	45
cfg.cc	4	4	0	0	4	9	9	0	0	9
cfg.h	0	0	0	0	0	2	2	0	0	2
clint.cc	4	15	0	11	4	18	38	0	20	18
common.h	0	0	0	0	0	2	2	0	0	2
csrs.cc	187	168	30	11	157	604	568	60	24	544
csrs.h	0	0	0	0	0	15	15	0	0	15
debug_module.cc	11	11	0	0	11	68	68	0	0	68
debug_rom_defines.h	0	0	0	0	0	1	1	0	0	1
decode_macros.h	4	4	0	0	4	100	127	4	31	96
decode.h	2	2	0	0	2	70	73	0	3	70
devices.cc	11	14	0	3	11	29	35	0	6	29
devices.h	0	0	0	0	0	1	1	0	0	1
dts.cc	38	38	0	0	38	108	108	0	0	108
encoding.h	0	0	0	0	0	1605	1808	2	205	1603
execute.cc	29	29	0	0	29	71	68	3	0	68
insn_macros.h	0	0	0	0	0	1	2	0	1	1
isa_parser.cc	134	134	0	0	134	156	156	0	0	156
isa_parser.h	0	0	0	0	0	11	11	0	0	11
log_file.h	2	2	0	0	2	4	4	0	0	4
memtracer.h	1	1	0	0	1	6	6	0	0	6
mmu.cc	162	158	12	8	150	262	256	13	7	249
mmu.h	170	176	3	9	167	155	161	1	7	154
ns16550.cc	1	49	0	48	1	25	125	0	100	25
opcodes.h	0	0	0	0	0	16	16	0	0	16
overlap_list.h	0	0	0	0	0	12	12	0	0	12
p_ext_macros.h	0	0	0	0	0	0	10	0	10	0
platform.h	0	0	0	0	0	3	3	0	0	3
plic.cc	5	46	0	41	5	21	124	0	103	21
processor.cc	155	129	28	2	127	536	512	25	1	511
processor.h	18	12	6	0	12	66	62	4	0	62
rom.cc	1	1	0	0	1	7	7	0	0	7
sim.cc	54	54	2	2	52	179	181	2	4	177
sim.h	0	0	0	0	0	4	4	0	0	4
simif.h	0	0	0	0	0	3	3	0	0	3
tracer.h	0	0	0	0	0	2	2	0	0	2
trap.h	0	0	0	0	0	19	18	1	0	18
triggers.cc	15	15	0	0	15	43	33	10	0	33
triggers.h	0	0	0	0	0	5	4	1	0	4
vector_unit.cc	7	7	0	0	7	42	42	0	0	42
vector_unit.h	0	0	0	0	0	8	8	0	0	8
Total	1034	1084	85	135	949	4336	4732	126	522	4210

Table G.2: Coverage comparison of Spike

G.3 External library

Table G.3: Coverage comparison of external library

Filename	Branch Coverage					Line Coverage				
	ISA	**Fuzz**	**ISA-U**	**Fuzz-U**	**Inter**	**ISA**	**Fuzz**	**ISA-U**	**Fuzz-U**	**Inter**
softfloat/										
f32_add.c	2	2	0	0	2	16	16	0	0	16
f32_classify.c	63	45	18	0	45	23	23	0	0	23
f32_div.c	8	14	5	11	3	47	63	13	29	34
f32_eq.c	18	11	7	0	11	19	17	2	0	17
f32_le.c	11	14	3	6	8	20	20	0	0	20
f32_lt_quiet.c	19	16	4	1	15	24	22	2	0	22
f32_lt.c	11	8	4	1	7	20	20	0	0	20
f32_mul.c	12	11	7	6	5	62	51	21	10	41
f32_mulAdd.c	0	0	0	0	0	15	15	0	0	15
f32_sqrt.c	10	8	7	5	3	47	29	23	5	24
f32_sub.c	1	2	0	1	1	15	16	0	1	15
f32_to_f64.c	5	7	1	3	4	28	36	0	8	28
f32_to_i32.c	7	7	1	1	6	25	25	0	0	25
f32_to_i64.c	9	6	4	1	5	33	33	0	0	33
f32_to_ui32.c	7	7	1	1	6	25	25	0	0	25
f32_to_ui64.c	9	6	4	1	5	33	33	0	0	33
f64_add.c	2	2	0	0	2	20	20	0	0	20
f64_classify.c	63	47	17	1	46	23	23	0	0	23
f64_div.c	10	5	9	4	1	66	42	33	9	33
f64_eq.c	18	7	11	0	7	19	16	3	0	16
f64_le.c	11	5	7	1	4	21	15	6	0	15
f64_lt_quiet.c	19	7	12	0	7	25	17	8	0	17
f64_lt.c	11	2	9	0	2	21	15	6	0	15
f64_mul.c	11	5	9	3	2	62	42	25	5	37
f64_mulAdd.c	0	0	0	0	0	15	15	0	0	15
f64_sqrt.c	10	11	4	5	6	54	51	10	7	44
f64_sub.c	1	2	0	1	1	18	20	0	2	18
f64_to_f32.c	5	6	0	1	5	28	31	0	3	28
f64_to_i32.c	7	7	1	1	6	23	23	0	0	23
f64_to_i64.c	10	5	5	0	5	31	25	6	0	25
f64_to_ui32.c	7	7	1	1	6	23	23	0	0	23
f64_to_ui64.c	9	7	3	1	6	31	31	0	0	31
i32_to_f32.c	5	5	0	0	5	12	12	0	0	12
i32_to_f64.c	4	4	0	0	4	19	19	0	0	19
i64_to_f32.c	4	7	1	4	3	17	24	0	7	17
i64_to_f64.c	3	5	0	2	3	9	12	0	3	9
internals.h	0	0	0	0	0	10	10	0	0	10
primitives.h	30	29	15	14	15	54	34	20	0	34
s_addMagsF32.c	7	17	1	11	6	42	56	0	14	42
s_addMagsF64.c	5	11	3	9	2	35	37	13	15	22
s_approxRecip-Sqrt32_1.c	2	2	1	1	1	23	23	0	0	23
s_countLeading-Zeros64.c	6	6	0	0	6	20	20	0	0	20
s_mul64To128.c	0	0	0	0	0	18	0	18	0	0
s_mulAddF32.c	14	19	10	15	4	80	70	43	33	37
s_mulAddF64.c	17	8	15	6	2	86	51	53	18	33
s_normRound-PackToF32.c	4	4	0	0	4	12	12	0	0	12

Continued on next page

Filename	Branch Coverage					Line Coverage				
	ISA	**Fuzz**	**ISA-U**	**Fuzz-U**	**Inter**	**ISA**	**Fuzz**	**ISA-U**	**Fuzz-U**	**Inter**
s_normRoundPackToF64.c	4	4	0	0	4	12	12	0	0	12
s_normSubnormalF32Sig.c	0	0	0	0	0	8	8	0	0	8
s_propagateNaNF32UI.c	0	2	0	2	0	0	4	0	4	0
s_propagate NaNF64UI.c	0	2	0	2	0	0	4	0	4	0
s_roundPackToF32.c	12	13	1	2	11	38	40	0	2	38
s_roundPackToF64.c	6	6	0	0	6	28	28	0	0	28
s_roundToI32.c	19	9	10	0	9	31	25	6	0	25
s_roundToI64.c	14	7	8	1	6	21	14	7	0	14
s_roundToUI32.c	17	9	8	0	9	27	19	8	0	19
s_roundToUI64.c	14	6	9	1	5	20	12	8	0	12
s_shiftRightJam128.c	1	0	1	0	0	12	0	12	0	0
s_subMagsF32.c	7	14	3	10	4	44	57	7	20	37
s_subMagsF64.c	7	12	3	8	4	41	47	10	16	31
softfloat_raiseFlags.c	0	0	0	0	0	3	3	0	0	3
specialize.h	0	0	0	0	0	18	12	6	0	12
ui32_to_f32.c	3	4	0	1	3	9	12	0	3	9
ui32_to_f64.c	1	2	0	1	1	13	14	0	1	13
ui64_to_f32.c	4	5	0	1	4	19	19	0	0	19
ui64_to_f64.c	3	4	0	1	3	11	14	0	3	11
Subtotal	599	505	243	149	356	1724	1577	369	222	1355
fdt/										
fdt_addresses.c	6	6	0	0	6	24	24	0	0	24
fdt_ro.c	86	86	0	0	86	210	210	0	0	210
fdt.c	78	78	0	0	78	139	139	0	0	139
fdt.h	0	0	0	0	0	11	11	0	0	11
libfdt_env.h	0	0	0	0	0	6	6	0	0	6
libfdt_internal.h	0	0	0	0	0	10	10	0	0	10
libfdt.h	0	0	0	0	0	28	28	0	0	28
Subtotal	170	170	0	0	170	428	428	0	0	428
fesvr/										
byteorder.h	0	0	0	0	0	12	13	0	1	12
device.cc	5	5	0	0	5	33	33	0	0	33
device.h	0	0	0	0	0	6	7	0	1	6
elf.h	0	0	0	0	0	6	6	0	0	6
elfloader.cc	38	39	0	1	38	75	75	0	0	75
htif.cc	61	61	4	4	57	153	152	4	3	149
htif.h	1	2	0	1	1	19	25	0	6	19
memif.cc	18	20	0	2	18	55	72	0	17	55
memif.h	0	0	0	0	0	6	6	0	0	6
syscall.cc	9	25	2	18	7	53	134	5	86	48
term.cc	2	4	0	2	2	9	20	0	11	9
Subtotal	134	156	6	28	128	427	543	9	125	418
Total	903	831	249	177	654	2579	2548	378	347	2201

Bibliography

[1] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, December 2019.

[2] Sifive. URL https://www.sifive.com/.

[3] What are the best practices and tools for isa verification and testing? URL https://www.linkedin.com/advice/0/what-best-practices-tools-isa-verification.

[4] Kostya Serebryany Max Moroz. Guided in-process fuzzing of chrome components. URL https://security.googleblog.com/2016/08/guided-in-process-fuzzing-of-chrome.html.

[5] RISCV-software. Spike risc-v isa simulator, 2019. URL https://github.com/riscv-software-src/riscv-isa-sim.

[6] rvemu: Risc-v emulator, . URL https://github.com/d0iasm/rvemu.

[7] Niraj Nayan Sharma Rishiyur S. Nikhil. Forvis: A formal risc-v isa specification. URL https://github.com/rsnikhil/Forvis_RISCV-ISA-Spec.

[8] Rishiyur S. Nikhil Prashanth Mundkur. Riscv sail model. URL https://github.com/riscv/sail-riscv.

[9] Single precision instruction on double precision register value, . URL https://github.com/d0iasm/rvemu/issues/25.

[10] Jianwen Zhu and D.D. Gajski. An ultra-fast instruction set simulator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 10(3):363–373, 2002. doi: 10.1109/TVLSI.2002.1043339.

[11] Risc-v proxy kernel and boot loader, 2013. URL https://github.com/riscv-software-src/riscv-pk.

[12] Sail. URL https://www.cl.cam.ac.uk/~pes20/sail/.

[13] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58–62, 2005. doi: 10.1109/MSP.2005.55.

[14] Michele Orrù Markus Vervier. Browser security white paper. Technical report, X41 D-SEC GmbH, 2017.

[15] Charlie Miller and Zachary N. J. Peterson. Analysis of mutation and generation-based fuzzing. Technical report, Independent Security Evaluators, 2007.

[16] Google. structure-aware-fuzzing, 2019. URL https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md.

[17] Duncan Grove Richard McNally, Ken Yiu and Damien Gerhardy. Fuzzing: The state of the art. Technical report, Command, Control, Communications and Intelligence Division Defence Science and Technology Organisation, Department of Defence, Australian Government, 2012.

[18] Alex Groce and John Regehr. The saturation effect in fuzzing, 2020. URL https://blog.regehr.org/archives/1796.

[19] Michal Zalewski. american fuzzy lop. URL https://github.com/google/AFL.

[20] libfuzzer – a library for coverage-guided fuzz testing. URL https://llvm.org/docs/LibFuzzer.html.

[21] Qemu: A generic and open source machine emulator and virtualizer. URL https://www.qemu.org/.

[22] Thuan Pham. Code coverage-guided fuzzing, 2022. URL https://swen90006.github.io/notes/Coverage-Guided-Fuzzing.html.

[23] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. Verifying instruction set simulators using coverage-guided fuzzing. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 360–365, 2019. doi: 10.23919/DATE.2019.8714912.

[24] Risc-v based virtual prototype (vp), . URL https://github.com/agra-uni-bremen/riscv-vp.

[25] Vladimir Herdt, Daniel Große, and Rolf Drechsler. Towards specification and testing of risc-v isa compliance⋆. *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 995–998, 2020.

[26] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735, 2019. doi: 10.1109/ICSE.2019.00081.

[27] V. Pham, M. Bohme, A. E. Santosa, A. Caciulescu, and A. Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47(09):1980–1997, sep 2021. ISSN 1939-3520. doi: 10.1109/TSE.2019.2941681.

[28] Karine Even Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. *GrayC: Greybox Fuzzing of Compilers and Analysers for C*. ACM, May 2023.

[29] Andrey Konovalov. Coverage-guided usb fuzzing with syzkaller, 2019. URL https://www.youtube.com/watch?v=1MD5JV6LfxA&ab_channel=OffensiveCon.

[30] 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, Boston, MA, August 2022. USENIX Association. ISBN 978-1-939133-31-1. URL https://www.usenix.org/conference/usenixsecurity22/presentation/trippel.

[31] Kostya Serebryany, Maxim Lifantsev, Konstantin Shtoyk, Doug Kwan, and Peter Hochschild. Silifuzz: Fuzzing cpus by proxy. *CoRR*, abs/2110.11519, 2021. URL https://arxiv.org/abs/2110.11519.

[32] Device tree what it is. URL https://elinux.org/Device_Tree_What_It_Is.

[33] Kunal Verma. Fixing the "too many open files" error in linux. URL https://www.tutorialspoint.com/fixing-the-too-many-open-files-error-in-linux.

[34] Standalonefuzztargetmain.c. URL https://github.com/llvm/llvm-project/blob/main/compiler-rt/lib/fuzzer/standalone/StandaloneFuzzTargetMain.c.

[35] Source-based code coverage. URL https://clang.llvm.org/docs/SourceBasedCodeCoverage.html.

[36] riscv-tests, . URL https://github.com/riscv-software-src/riscv-tests.

[37] Is pk preventing me from accessing certain address in simulation by spike? URL https://github.com/riscv-software-src/riscv-pk/issues/289.

[38] lrsc.s. URL https://github.com/riscv-software-src/riscv-tests/blob/master/isa/rv64ua/lrsc.S.