Closing the RISC-V Compliance Gap: Looking from the Negative Testing Side*

Vladimir Herdt[1] Daniel Große[1,2] Rolf Drechsler[1,2]
[1]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{vherdt,grosse,drechsle}@informatik.uni-bremen.de

***Abstract*—Compliance testing for RISC-V is very important. Therefore, an official hand-written compliance test-suite is being actively developed. However, besides requiring significant manual effort, it focuses on positive testing (the implemented instructions work as expected) only and neglects negative testing (consider illegal instructions to also ensure that no additional/unexpected behavior is accidentally added). This leaves a large gap in compliance testing.**

In this paper we propose a fuzzing-based test-suite generation approach to close this gap. We found new bugs in several RISC-V simulators including *riscvOVPsim* from Imperas which is the official reference simulator for compliance testing.

I. Introduction

An *Instruction Set Architecture* (ISA) defines the interface between the *Hardware* (HW) of a processor and the *Software* (SW). While, as a consequence, the format of a SW binary running on a processor is clearly defined by the ISA, nothing is specified on how to implement the processor[1]. An ISA which has become very popular is the RISC-V ISA [1]. Driven by the ideas from open source SW, the RISC-V ISA is open, royalty-free, and maintained by the non-profit RISC-V foundation [2]. The major goal of the RISC-V ISA is to provide a path to a new era of processor innovation via open standard collaboration. Around RISC-V an ecosystem is rapidly emerging. Staring from the base ISA, a big plus of the RISC-V ISA is the availability of modular standard extensions. In addition, extensibility has been designed into the ISA allowing for custom instructions. While this flexibility offers significant advantages (free selection of what is needed from the standard extensions and addition of dedicated custom instructions for optimization of the target application), also a major challenge is posed: fragmentation. The above mentioned cooperation driving the ecosystem will fail, if different RISC-V CPU implementations do not comply with the ISA specification. Therefore, the *compliance* of each RISC-V CPU to the ISA specification has to be validated. This is the task of *compliance testing*. More precisely, compliance testing checks whether registers are missing, modes are not there, instructions are absent, as well as the presence of only those instructions which are part of the selected ISA [3], [4]. If the compliance test passes for a CPU, the HW/SW contract is maintained and the SW will be portable between implementations. Note that compliance testing is *not* design verification. In contrast to compliance testing, the goal of verification is to find errors in the CPU implementation.

The importance of compliance testing has been recognized very early by the RISC-V foundation and therefore the compliance task group has been formed [5]. The compliance task group actively develops the official hand-written compliance test-suite. The individual test-cases are designed to compute an in-memory signature, that represents the output of the test result and is dumped at the end of the test execution. For compliance testing, these signatures are compared against golden reference signatures (obtained by running the test-suite on a reference simulator). A separate sub test-suite is developed for the RISC-V base ISA as well as for each standard ISA extension. Besides the significant manual effort for the maintenance, the compliance test-suite focuses on positive testing only, i.e. to show that the implemented instructions work as expected. However, it neglects negative testing, i.e. to consider illegal instructions to also ensure that no additional/unexpected behavior is accidentally added. This leaves open a large gap in compliance testing.

Contribution: In this paper we propose a fuzzing-based test-suite generation approach to close this gap. We leverage state-of-the-art fuzzing techniques (based on LLVM *libFuzzer*) to iteratively generate test-cases which are executed on a RISC-V simulator and guide the fuzzing process through the observed code coverage of the simulator. A filter is integrated between fuzzer and simulator to conservatively remove test-cases with infinite loops and platform specific details, to avoid spurious signature mismatches and to enable automated compliance testing. To further improve the fuzzing effectiveness, we incorporate a custom coverage metric and fuzzing mutator. Our approach is very effective for negative testing and thus complements the official compliance test-suite. We found new bugs in several RISC-V simulators including *riscvOVPsim* from Imperas, which is the official reference simulator for compliance testing (i.e. used to generate reference signatures).[2]

II. Related Work

For the purpose of verification, several approaches to test program generation have been proposed. In particular model-based approaches, which separate the test generator from the architecture description, have a long history. Prominent examples using constraint solving techniques are [6], [7]. An optimized test generation framework is presented in [8]. It propagates constraints among multiple instructions in an effective manner. The test program generator of [9] includes a coverage model that holds constraints describing execution paths of individual instructions. Other approaches integrate coverage-guided test generation based on bayesian networks [10] and other machine learning techniques [11] as well as fuzzing [12].

Recently, test generation approaches specifically targeting RISC-V have emerged [13]–[15]. The Scala-based *Torture Test* generator [13] generates tests by integrating pre-defined randomized test sequences and supports several RISC-V ISA extensions. However, it has two major drawbacks: it does not build upon the official compliance testing format and only performs positive testing, i.e. illegal instructions are not considered. Another approach is *RISCV-DV* [14]. It leverages SystemVerilog in combination with UVM (*Universal Verification Methodology*) to generate RISC-V instruction streams based on constrained-random descriptions. However,

*This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project VerSys under contract no. 01IW19001 and within the project SATiSFy under contract no. 16KIS0821K.

[1]Such an implementation is referred to as micro architecture and the most famous ones for the x86 ISA are the processors from AMD and Intel.

[2]Visit http://www.systemc-verification.org/risc-v for our most recent RISC-V related approaches.

RISCV-DV offers only very limited support for generation of illegal instructions and thus is not suitable for comprehensive negative testing. In addition, the approach does not support the compliance testing format and requires a commercial RTL simulator providing SystemVerilog (constrained-random features) as well as UVM support. [15] proposed coverage-guided fuzzing for verification of instruction set simulators. However, the approach is not compatible with the compliance testing format, since it generates platform dependent tests in ELF format instead of providing platform independent tests written in assembler (ASM), which also significantly reduces its applicability to different platforms. Furthermore, the approach does not support automated testing, as it requires manual inspection to avoid false negatives due to platform specific details.

In addition, there are also formal verification approaches for RISC-V based on model checking. Notable are riscv-formal [16] and the OneSpin 360 DV RISC-V verification app [17]. However, both approaches clearly target the verification of an implementation.

Finally, [18] specifically considers compliance testing of RISC-V. It defines a test-suite specification mechanism and leverages constraint solving techniques to generate a comprehensive compliance test-suite as counterpart to the hand-written official compliance test-suite. However, it also only focuses on positive testing and does not consider negative testing aspects, such as illegal instructions.

III. Preliminaries

A. RISC-V

The RISC-V ISA consists of a mandatory base integer instruction set, denoted RV32I, RV64I or RV128I with corresponding register widths, and various optional extensions denoted as single letters, e.g. M (integer multiplication and division), A (atomic instructions), C (compressed, i.e. 2 byte instructions), F and D (single and double precision floating point) etc. Thus, RV32IMC denotes a 32 bit core with M and C extensions. G denotes the IMAFD instruction set, hence RV32GC=RV32IMAFDC. Each core has 32 general purpose registers x0 to x31 (with x0 being hardwired to zero) and the floating point (FP) extensions add additional 32 FP registers. Instructions access registers (source: RS1 and RS2, destination: RD) and immediates to do their operation. Format and semantics (for the base ISA and extensions) are defined in the unprivileged ISA specification [1].

In addition, the privileged (architecture) specification [19] covers further important functionality that is required for environment interaction and operating system execution. It includes different execution modes, in particular the mandatory Machine mode as well as the Supervisor and User mode extensions with corresponding *Control and Status Registers* (CSRs) descriptions. CSRs are registers serving a special purpose, that form the backbone of the privileged architecture description, such as MTVEC (stores the trap/interrupt handler address), MHARTID (read-only core id) and MSTATUS (main control and status register for the core).

B. LLVM libFuzzer

libFuzzer is an LLVM-based state-of-the-art coverage-guided fuzzing engine that proved very effective in finding several SW bugs [20]. It aims to create input data (binary bytestreams) in order to maximize the code coverage of the SUT (SW Under Test). Therefore, the SUT is instrumented by *Clang* compiler to report code coverage to *libFuzzer*. Input data is transformed by applying a set of predefined mutations (shuffle bytes, insert bit, etc.) randomly. Input size is gradually increased (when coverage starts to saturate).

Technically, *libFuzzer* is linked with the SUT, hence performs so called *in-process* fuzzing, and allows to pass inputs to the SUT as well as receive coverage information back through specific interface

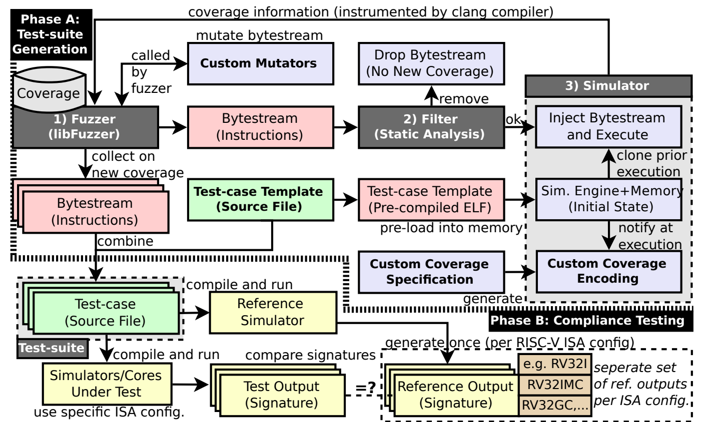

Fig. 1. Overview: fuzzer-based approach for RISC-V compliance testing

functions. For example, the SUT receives inputs through the *LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size)* function.

IV. Fuzzing-based RISC-V Compliance Testing

This section presents our fuzzer-based approach for RISC-V compliance testing. We start with an overview.

A. Approach Overview

Fig. 1 shows an overview on our approach. Essentially, it consists of two subsequent phases: first a fuzzer-based test-suite is generated (Phase A, shown on top of Fig. 1), then the test-suite is leveraged for compliance testing (Phase B, shown on bottom of Fig. 1). Our generated test-suite follows the same format as the official compliance test-suite and thus also generates signatures for compliance testing. However, in contrast to the official suite, which has a dedicated sub suite for each RISC-V ISA extension, we generate a single suite that can be compiled and executed with any supported RISC-V ISA (currently we support any configuration of RV32GC), since unsupported instructions should be considered illegal and result in an exception. Furthermore, due to the randomness of the fuzzing process, both phases can be continuously repeated, to achieve an even more comprehensive testing.

Test-suite generation involves three main steps: 1) fuzzer, 2) filter and 3) simulator, that are repeated until the specified time (or memory) limit is reached. The fuzzer generates (random) bytestreams, which are interpreted as RISC-V instruction sequences, and passes them to the filter that decides whether the bytestream is further processed or dropped. Essentially, the filter conservatively drops bytestreams with infinite loops and platform specific details (test-cases are available as source files and are compiled separately for each target platform with custom definitions), to avoid spurious signature mismatches. This is very important to enable a continuous and automated testing process, because the potential presence of spurious mismatches would require manual analysis to confirm that they are indeed spurious (to avoid missing bugs). In case the bytestream is dropped, no coverage information is returned to the fuzzer and hence the fuzzer considers that bytestream uninteresting and does not collect it. Otherwise, the bytestream is executed on the simulator and coverage information is returned to the fuzzer. This happens automatically by compiling the simulator with *Clang* and using the *-fuzzer* sanitizer (because we use LLVM *libFuzzer*, which is compatible with *Clang*). For simulation, we provide a test-case template, as RISC-V assembler (ASM) source file. As optimization, the test-case template is pre-compiled into an ELF and pre-loaded into the simulator memory. Before each bytestream execution the simulator is cloned to preserve the initial state.

To improve the fuzzing process we use a custom mutator and coverage specification. The coverage specification is automatically transformed into a source file that is embedded into the simulator and updated on every instruction execution.

Next, we present more details on the test-case format (Section IV-B) and filter (Section IV-C) as well as the custom mutator (Section IV-D) and coverage encoding (Section IV-E).

B. Test-case Template

Our test-case template builds on the RISC-V compliance testing format [5] to ensure that the generated test-suite is directly applicable to all platforms that support this standard format. It performs a generic system initialization sequence (initialize core CSRs and register a trap handler) and then enters the actual test-case body. Macros are used to mark the begin/end of code and data as well as halt execution. The macros as well as compilation flags are platform specific, thus we cannot rely on hardcoded absolute addresses to access memory or use as jump target (because code and data may be stored at different addresses per platform).

The test-case body starts by initializing all registers: x0 to x29 are loaded from hardcoded memory values, x30 and x31 (chosen arbitrarily) are set to point into the middle of the data memory by using a label. Thus, x0 to x29 have equal values among all platforms and hence can be used for *comparable* computations while x30 and x31 are platform specific but can be used as address for memory accesses. The data memory is large enough to support any additional immediate offset, i.e. [-2048, +2047].

The test-case body ends by first incrementing x26 (an arbitrary register to distinguish between cases where the test code executes with/without exceptions) and then initiates the shutdown sequence that will write back all register values (except x30 and x31 since they have platform specific values) to the data memory and halts execution (causing a signature dump). In case of an illegal instruction in the bytestream, control is transferred to the trap handler, which initiates the shutdown sequence (but bypasses the x26 increment).

In-between start/end of the test-case body, the fuzzer generated bytestream is injected. The template provides a list of jump instructions (to the body end) at this point that will simply be overwritten with raw memory declarations, e.g. *.word 0x12345678*, for each word in the bytestream. The number of jump instructions in the template is large enough for the bytestream to not exceed it.

Please note, we also load and store the content of floating point (FP) registers alongside the normal registers. However, we conditionally guard it with the definition of *__riscv_fdiv*, which is set by GCC when selecting a RISC-V ISA (*-march* flag) with FP support.

C. Filter

The filter works by performing an abstract local execution of the bytestream that traverses the local control flow and checks the reachable instructions alongside. The abstract execution state consists of a program counter (PC), a mark (clean/dirty) for each register that indicates whether the register can be used as address for a memory access, and data structures to keep track of the control flow to avoid loops. At the beginning PC is set to zero (i.e. pointing to the beginning of the bytestream) and all registers are marked dirty except for x30 and x31 (since they are initialized with a label to the data memory by the test-case, recall Section IV-B).

The filter then repeats a fetch, decode and execute loop. Thus, it checks whether the next instruction (based on PC) is compressed (the two least significant bits are not 11). Then, it decodes the instruction, increments PC by 4 (normal) or 2 (compressed) accordingly, and (abstractly) executes the decoded instruction. To avoid loops, the filter essentially checks that the same PC is not revisited. Furthermore, PC is not allowed to leave the local bounds of the bytestream (due to a jump/branch). A branch instruction forks the execution path, by cloning the abstract execution state S into S_T and S_F. The PC of S_T is updated with the branch offset, which is relative to the current PC and hence platform independent, accordingly (the PC of S_F is already set correctly to *fallthrough* to the next instruction). S_T and S_F are processed independently.

The instructions JALR, [M,S,U]RET, WFI, EBREAK and SFENCE.VMA are forbidden (the bytestream is dropped if they are reachable on any path). The reason is that JALR and [M,S,U]RET perform a register/CSR based jump. WFI (*Wait For Interrupt*) might halt a processor causing non-termination (since no interrupt is coming). EBREAK can have a special semantic and SFENCE.VMA is a privileged instruction that is often not implemented (which is not a bug by itself but a deliberate decision). All (six) CSR instructions are forbidden too, due to highly platform dependent behavior of CSRs (we provide more details on the problem and potential solutions in Section VI).

Any instruction writing to a register RD, marks RD dirty. A load/store instruction is forbidden if its address register is dirty. In addition, we also require that the immediate (which will be added to the register address to obtain the final access address) is properly aligned, because the RISC-V ISA allows both aligned and unaligned load/store instructions (which would lead to spurious signature mismatches).

A path passes when reaching an illegal instruction (since the next instructions will not be reached due to the exception) or the end of the bytestream.

For illustration Fig. 2 shows an example. The left side shows an ASM program, that represents the bytestream. Each instruction is prepended by it's (local) address (for simplicity we assume all instructions are non-compressed, i.e. are 4 byte long). The right side shows the three possible control flow paths through this program, starting from the initial state. Each instruction execution (annotated above a state transition) results in a new state. The current PC and set of registers marked *clear* are shown below each state. The ASM program (bytestream) is accepted by the filter because all paths are accepted. Please note, that the program contains a WFI instruction which is in the forbidden category. However, the WFI has no influence, since it is never reached on any path. Similarly, the ADD instruction at address 12 that marks x30 dirty is not reached and hence the LW at address 28 succeeds. BLT and BEQ fork the active path to continue at PC_T=28, PC_F=20 and PC_T=16, PC_F=28, respectively.

Our filter currently supports the RV32GC ISA. Hence, the generated test-suite can be executed on any sub-ISA of RV32GC, such as RV32I, RV32IMC etc. To add a new instruction extension, the filter needs to be extended as well. Otherwise, the filter will consider them as illegal instructions and let them pass unconditionally.

D. Custom Mutator

We integrate a custom mutator to provide the fuzzer with valid instruction (opcode) patterns to increase the number and length of valid instructions. Our mutator is attached through the *libFuzzer* provided interface and is called with equal probability to the existing mutators. Basically, the mutator moves through the bytestream instruction by instruction (we use a 4 byte format) and injects valid opcodes, while keeping all other parameters randomized by the fuzzer. Please note, we only inject instructions that pass our filter (since the bytestream will be dropped otherwise). Besides avoiding instructions from the forbidden category, we only use small offsets for branch and jump instructions (might still be rejected by the filter but the probability is much smaller) and only inject load/store instructions that use

```
1 0:  ADD x31, x2, x3  //mark x31 dirty
2 4:  JAL x2, 20       //mark x2 dirty, to PC=24
3 8:  WFI              //fobidden, drop bytestream
4 12: ADD x30, x2, x3  //mark x30 dirty
5 16: BLT x30, x31, 12 //fork, to PC=28:20
6 20: illegal          //accept path
7 24: BEQ x1, x2, -8   //fork, to PC=16:28
8 28: LW x5, -16(x30)  //require x30 clean
9 //accept path
```

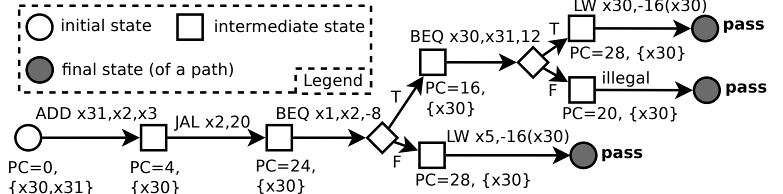

Fig. 2. Left side shows an example RISC-V ASM program (for a bytestream with 32 byte) and right side shows the corresponding control flow paths

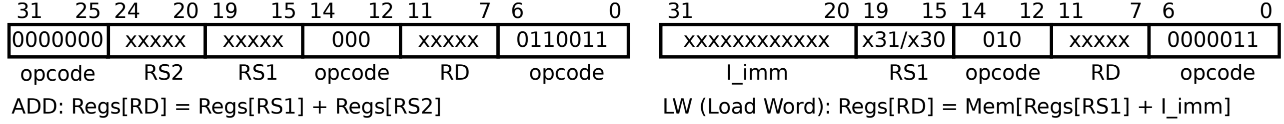

Fig. 3. Format and semantic for the ADD and LW instruction. The opcodes are injected by our mutator, the other fields remain randomized (each random bit is denoted with an *x*). Special constraints are used (such as setting RS1 to x30 or x31 for LW) to pass our filter.

x30 or x31 as address register. For illustration Fig. 3 shows the instruction format and semantic for ADD and LW. When injecting the ADD instruction, the RS1, RS2 and RD fields remain random, but all opcode fields are overwritten by the mutator, thus making the instruction a valid (though randomized) ADD. Similarly, for the LW instruction, though here rs1 is always set to either x30 or x31 register to pass the filter.

E. Custom Coverage

By default the fuzzing process is guided by code coverage emitted by the simulator that executes the bytestreams. We consider two additional coverage metrics.

The first is a hash-based coverage that is simple, generic and scalable. Basically it computes a small hash value of the instruction word and considers every different hash value as new coverage. This adds a significant amount of variance and randomness to the generated test-suite. Technically, we use a C++ *std::hash<uint32_t> fn* hash function. Then, every fetched instruction is passed through a (large) switch statement: *switch (fn(fetched_word) % N)*. N is the configurable number of hashes to use. Inside the switch statement we generate N cases, for $i \in \{0, ..., N\}$, as *case i: __asm__ __volatile__(""); break;*. The __asm__ __volatile__ statement ensure that the cases are not removed by the compiler.

The second coverage reasons about structure and values of RISC-V instructions. It is provided through an external specification file. It can strengthen the fuzzer in the field of positive testing by collecting further test-cases with valid instructions. Basically, we use a small set of rules such as: 1) RD=x0, 2) RD≠x0, 3) RD=RS1, and 4) RD≠RS1. Each rule is applicable to instructions that have the corresponding fields and defines a coverage point with the rules condition, for example *if (decoded_instruction.opcode == ADD && decoded_instruction.RD == x0) __asm__ __volatile__;* for rule 1 and opcode ADD (all matching opcodes are enumerated)[3]. The first and second rule are due to the RISC-V hardwired x0 register. The third and fourth are useful to check for effects where the update order is not correct. Similarly, we have a rule for three registers (all equal, all not equal, etc). Finally, we use value rules Reg[RS1] OP Reg[RS2] with $OP \in \{=, \neq, <, >\}$ and Reg[RS*] $\in \{MIN, MAX, -1, 0, 1\}$, and similar rules for immediates.

V. Experiments

We have implemented our fuzzer-based approach for RISC-V compliance testing and evaluated its effectiveness on a set of RISC-V simulators. As foundation for the fuzzing process we use the 32 bit (instruction set) simulator of the open source RISC-V VP [21], [22]. Next, we first provide more information on the fuzzer-based test-suite generation process (Section V-A) and then present results on the compliance testing evaluation (Section V-B). All experiments have been performed on a Linux system with an Intel Core i5-7200U processor.

A. Test-suite Generation

Fig. 4 shows execution information for four different fuzzing configurations (v0 to v3) that use different coverage metrics: v0 uses only code coverage of the ISS; v1 adds the custom coverage rules (structural and value metrics) to v0 (additional 2281 coverage points); v2 and v3 add hash coverage with 4096 and 16384 coverage points to v1, respectively. Fig. 4 shows how the number of test-cases grows compared to the number of fuzzer executions (i.e. over time). The runtime is fixed to 30 minutes for each configuration. We use a 64 byte input length limit for the fuzzer and configured it to increase the input length more slowly (*-len_control=10000*). It can be observed that the number of test-cases grows very rapidly in the first quarter and then gradually saturates (please note the logarithmic scale on the X axis). The average executions per second are at 45,873 with the minimum at 12,302 and maximum at 68,873. To achieve this high performance, it has been particularly important to pre-compile and pre-load the test-case template and use a small simulator memory size (32 KB). The highest measured memory consumption on our evaluation system has been 1063 MB for configuration v3. The coverage metric is very important since it has immediate impact on the fuzzing process. First, on the performance, since the coverage needs to be tracked (which costs time) and it influences how fast the fuzzer increases the input size (every time the coverage starts to saturate), which in turn increases the probability that our filter drops more inputs. Second, on the number of generated test-cases, since the fuzzer only collects test-cases that increase coverage. For the following compliance testing evaluation we use the v3 configuration.

B. Compliance Testing

We consider five different RISC-V simulators, which all support the RISC-V compliance testing format, in this evaluation[4]: *riscvOVPsim*, *Spike*, *VP*, *GRIFT* and *sail-riscv*. *riscvOVPsim* [5] (see the *riscv-ovpsim* folder) is the reference simulator for compliance

[3]We use a slightly optimized implementation by using switch case statement for the opcode and grouping all rules below the opcode.

[4]We also briefly evaluated the *rocket* and *Ibex* cores, since they are listed as targets in the compliance testing repository. Both (RTL) cores can be compiled into a (C++) simulator using *verilator*. However, the rocket simulator had problems with the compliance testing format (it failed every basic RV32I test) and the Ibex simulator stopped on the first exception (e.g. illegal instruction) without dumping a signature, which makes it not applicable in combination with negative testing. Thus, we omitted these cores from the evaluation.

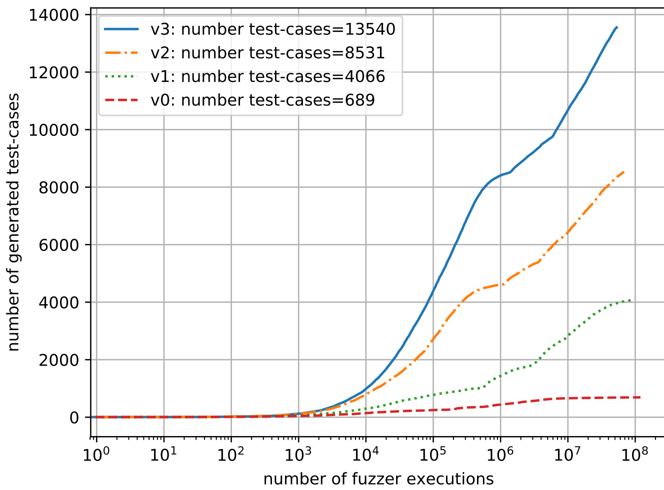

Fig. 4. Fuzzer execution information for different settings (30 min. runtime)

testing, i.e. it is used to generate the reference signatures by the compliance task group. *Spike* [23] is the official reference simulator for RISC-V from UC Berkeley that aims to be an executable golden model for the RISC-V ISA specification. *VP* [21] is a RISC-V based Virtual Prototype implemented in SystemC TLM. *GRIFT* [24] is a Haskell-based RISC-V formalization that aims to provide the foundation for several analysis techniques for RISC-V. *sail-riscv* [25] is implemented in Sail, a special language for describing ISAs with support for generation of simulator backends as well as theorem-prover definitions, and aims to become another (executable) formalization of the RISC-V ISA.

Except for *GRIFT* which currently fails one test-case of the compliance test-suite (in the A extension), all simulators pass all applicable compliance test-suites (some test-suites are not applicable, because the simulator simply does not implement the respective RISC-V ISA extension support).

Following the compliance testing convention, we also use *riscvOVPsim* to generate reference signatures. In addition, we consider three different RISC-V RV32 ISA configurations, namely: RV32I, RV32IMC and RV32GC. With our approach we observed several errors and inconsistencies on every simulator (across all ISA configurations) that we used in this experiment. Most errors are related to incorrect decoding of instructions, which can be particularly well detected with fuzzing based approaches, but we also found logical execution errors in valid instructions and some other issues in the (internal) simulator implementation. Table I shows a summary of the results for an exemplary fuzzer generated test-suite (the v3 configuration test-suite as discussed in Section V-A). It shows the number of signature mismatches that we observed between *riscvOVPsim* and the respective simulator (shown in Column 2 to 5) for each of the three ISA configurations (Rows 2 to 4) by running the test-suite (/ means not supported by simulator). It takes 10 to 20 minutes, depending on the simulator, to run (and also compare the generated signatures) the whole test-suite for a single ISA configuration. Please note, that the results can vary slightly, due to the randomness in the fuzzer. However, we consistently observed several mismatches for each fuzzer run. We consider this randomness actually a strength of our approach, since it can also be used for continuous negative testing to obtain more comprehensive results. Next, we present our findings in more detail, grouped by the respective simulator:

- **Spike** dumps an incorrect test signature in case of an ECALL instruction inside of the test body.

TABLE I
NUMBER OF SIGNATURE MISMATCHES AGAINST *riscvOVPsim*

RISC-V ISA	*Spike*	*VP*	*sail-riscv*	*GRIFT*
RV32I	7	5	crash	124
RV32IMC	9	32	crash	1047
RV32GC	9	/	/	141

- **VP** uses a wrong mask for the ECALL instruction in the decoder which allows an invalid instruction to be decoded and executed as an ECALL. In addition, reserved non-hint compressed instructions, e.g. *"c.lwsp x0, 0(sp)"*, are erroneously normally expanded and executed without causing an illegal instruction exception[5].
- **GRIFT** updates the RA register on an invalid jump (target address not 32 bit aligned on RV32I) before triggering an illegal instruction exception (which is incorrect, since illegal instruction should have no side effects). Furthermore, the RV32IMC compliance testing target has been incorrectly configured to RV32GC, thus floating point and atomic instructions are erroneously accepted as well. In addition, similar to VP, reserved non-hint compressed instructions are also erroneously accepted as legal instructions. Finally, we also found the bug that SC.W instruction performs memory access even without pending LR.W reservation (which was the only bug found by the official compliance test-suite).
- **sail-riscv** has several incomplete decoder checks that cause invalid instructions to be accepted as valid ones. Some inputs crashed *sail-riscv*, others led to non-termination (which indicates that an invalid instructions has been interpreted as a backward branch/jump).
- **riscvOVPsim** accepts opcodes reserved for custom (non-compressed) instruction extensions which should cause an illegal instruction exception on the base ISA configuration (to trigger this error on *riscvOVPsim* an additional special bit pattern must be set as well in the instruction besides the opcode). All other simulators that we tested (including *Spike*), correctly performed a jump to the trap handler due to an illegal instruction exception in this case.

Our evaluation clearly shows that a fuzzing-based approach is an enrichment for compliance testing. In contrast to the existing compliance test-suite, which focuses on positive testing (i.e. use hand-written well-defined tests) to check that the required instructions are correctly implemented, our fuzzing-based approach is complementary by focusing on negative testing (i.e. to check that no additional functionality is accidentally added). It is very well suited for decoder checking, in particular that no illegal instruction passes as a legal one and that no additional instructions have been accidentally implemented, and testing against unexpected cases. Such error cases are very hard to detect, because the compiler does not generate illegal instructions and the existing RISC-V testing frameworks provide only very limited support. In addition, the inherent randomness of fuzzing enables it to be re-run continuously. In combination with our carefully designed and fully compliance testing compatible test-case format and static analysis based filter, a fully automated and comprehensive negative testing is enabled. Finally, our approach is not fixed to a specific RISC-V ISA but applicable to different ISA configurations (currently any combination of RV32GC) and is easily extendable to support additional instruction set extensions.

[5]RISC-V distinguishes different cases of reserved compressed instructions, some are merely hints which execute as NOP, others are reserved opcodes for internal and custom extension and should trigger an illegal instruction exception when attempted to be executed on an ISA without the respective extension.

VI. Discussion and Future Work: The CSR Challenge

Our experiments demonstrated the effectiveness of our fuzzer-based approach for finding errors and inconsistencies through negative testing. It is complementary to the positive testing approach of the official compliance testing test-suite and in combination provides a strong compliance testing framework for the unprivileged RISC-V ISA specification. However, one large and important open challenge is compliance testing of the privileged ISA, in particular the CSRs. In contrast to the instruction set specifications, the CSR behavior is much less clearly and unambiguously defined. Next, we exemplarily discuss the most relevant Machine mode CSRs:

MTVAL provides exception specific information (for example, the instruction data in case of an illegal instruction exception). However, it is also legal behavior to simply set MTVAL to zero in case the feature is not supported (for some exception). Conditional behavior like this cannot be handled with the current compliance testing approach, since the reference signatures are generated in advance and compared unconditionally to the output signatures. Bits in **MIP** and **MIE** can be hardcoded to zero if the respective interrupt source is not available, thus in one case a write succeeds in the other case it may be ignored and both are legal behaviors. **MSTATUS** has flags which are optional and exhibit different behavior in Machine and Supervisor mode. **MSCRATCH** can be used by the implementation at will, and hence can arbitrarily change its value in an architecture specific way. Timing related CSRs such as **MCYCLE** and **MTIME**, as well as the performance counter (which are also optional and can be hardwired to zero), yield architecture specific results and thus should not be compared. Even **MINSTRET** that simply counts the number of executed instructions cannot be used as signature, because some platforms use a built-in hardcoded initialization sequence and the compliance testing framework uses customized initialization and shutdown sequences (hence executes different number of instructions). **MCAUSE** and **MEPC** are only guaranteed to hold supported exception codes and valid virtual addresses, respectively. Invalid addresses may be freely converted by the implementation before writing them to MEPC. **MTVEC** can also contain a hardwired read-only value, which again obviously can be architecture specific and hence need to be considered to be arbitrary. Similarly, PMP registers (memory protection) are optional.

Furthermore, Supervisor and User modes are often hardcoded and cannot be deactivated (which is even the case for Spike). Hence, access to supervisor and user mode CSRs is possible as well, even though the test is runs in Machine mode and is supposed to test Machine mode CSRs. In addition, this has impact on Machine mode CSRs such as MISA, which has the Supervisor and User bits set too.

In summary the privileged architecture, in particular the CSRs, expose architecture specific information and provide a large degree of freedom for implementations. The consequence is that basically all simulators/cores implement a (more or less slightly) different subset/configuration of the privileged architecture, yet all of them can still be compliant to the RISC-V specification. This significantly complicates compliance testing, since it is very difficult to automatically test for correct behavior. We envision three directions for future work: **1)** It seems viable to tackle the CSR problem by writing very fine grained tests[6] for each CSR and then select them dynamically for each tested platform. This requires the RISC-V privileged architecture specification (which admittedly still officially is a draft) to list capabilities and interdependencies between CSRs more accurately. **2)** Step 1 is a good foundation to develop suitable coverage metrics for CSR testing to quantify the testing effort. **3)** Extend the compliance testing signature with *don't care* values to deal with conditional behavior, i.e. store a second file alongside the signature that describes which parts can be ignored in the comparison on which condition (e.g. ignore the reference output of MTVAL if the test output assigned a zero to MTVAL).

[6]Including specialized tests that not just compare the whole (CSR) state, e.g. testing of a performance counter could check that the counter increments when enabled but not care about the exact architecture specific counter value.

VII. Conclusion

In this paper we proposed a fuzzing-based approach to provide strong negative testing capabilities for RISC-V compliance testing, which complements the existing official test-suite that focuses on positive testing (of the unprivileged RISC-V ISA). We found new bugs in several RISC-V simulators including *riscvOVPsim* from Imperas (the official reference simulator for compliance testing). Finally, we reviewed the still open challenge in compliance testing of the privileged RISC-V ISA (CSRs in particular) and sketched possible solutions.

References

[1] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2017.
[2] "RISC-V foundation," https://riscv.org/risc-v-foundation/.
[3] L. Moore, S. Davidmann, and L. Lapides, "Compliance methodology and initial results for RISC-V ISA implementations," in *Embedded World Conference*, 2019.
[4] B. Bailey, *The Challenge Of RISC-V Compliance*, https://semiengineering.com/toward-risc-v-compliance/.
[5] "RISC-V compliance task group," https://github.com/riscv/riscv-compliance.
[6] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, "Genesys-pro: innovations in test program generation for functional processor verification," *D&T*, pp. 84–93, 2004.
[7] B. Campbell and I. Stark, "Randomised testing of a microprocessor model using SMT-solver state generation," in *Formal Methods for Industrial Critical Systems*, F. Lang and F. Flammini, Eds., 2014, pp. 185–199.
[8] Y. Katz, M. Rimon, and A. Ziv, "Generating instruction streams using abstract CSP," in *DATE*, 2012, pp. 15–20.
[9] M. Chupilko, A. Kamkin, A. Kotsynyak, and A. Tatarnikov, "MicroTESK: specification-based tool for constructing test program generators," in *HVC*, 2017.
[10] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *DAC*, 2003, pp. 286–291.
[11] C. Ioannides, G. Barrett, and K. Eder, "Feedback-based coverage directed test generation: An industrial evaluation," in *Hardware and Software: Verification and Testing*, S. Barner, I. Harris, D. Kroening, and O. Raz, Eds., 2011.
[12] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi, "Testing CPU emulators," in *ISSTA*, 2009, pp. 261–272.
[13] "RISC-V torture test generator," https://github.com/ucb-bar/riscv-torture.
[14] "RISCV-DV," https://github.com/google/riscv-dv.
[15] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in *DATE*, 2019, pp. 360–365.
[16] "RISC-V formal verification framework," https://github.com/SymbioticEDA/riscv-formal.
[17] "OneSpin 360 DV RISC-V Verification App," https://www.onespin.com/solutions/risc-v.
[18] V. Herdt, D. Große, and R. Drechsler, "Towards specification and testing of RISC-V ISA compliance," in *DATE*, 2020.
[19] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2017.
[20] "libFuzzer - a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, 2018.
[21] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "RISC-V based virtual prototype: An extensible and configurable platform for the system-level," *JSA*, 2020.
[22] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *FDL*, 2018, pp. 5–16.
[23] "Spike RISC-V ISA simulator," https://github.com/riscv/riscv-isa-sim.
[24] "GRIFT - galois RISC-V ISA formal tools," https://github.com/GaloisInc/grift.
[25] "Riscv sail model," https://github.com/rems-project/sail-riscv.