Mutation-based Compliance Testing for RISC-V

Vladimir Herdt
Institute of Computer Science, University of Bremen
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
vherdt@uni-bremen.de

Sören Tempel
Institute of Computer Science, University of Bremen
Bremen, Germany
tempel@uni-bremen.de

Daniel Große
Institute for Complex Systems, Johannes Kepler University
Linz, Austria
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
daniel.grosse@jku.at

Rolf Drechsler
Institute of Computer Science, University of Bremen
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
drechsler@uni-bremen.de

ABSTRACT

Compliance testing for RISC-V is very important. Essentially, it ensures that compatibility is maintained between RISC-V implementations and the ever growing RISC-V ecosystem. Therefore, an official *Compliance Test-suite* (CT) is being actively developed. However, it is very difficult to achieve that all relevant functional behavior is comprehensively tested.

In this paper, we propose a mutation-based approach to boost RISC-V compliance testing by providing more comprehensive testing results. Therefore, we define mutation classes tailored for RISC-V to access the quality of the CT and provide a symbolic execution framework to generate new test-cases that kill the undetected mutants. Our experimental results demonstrate the effectiveness of our approach. We identified several serious gaps in the CT and generated new tests to close these gaps.

KEYWORDS

RISC-V, Compliance Testing, Mutation, Instruction Set Simulation, Symbolic Execution

ACM Reference Format:
Vladimir Herdt, Sören Tempel, Daniel Große, and Rolf Drechsler. 2021. Mutation-based Compliance Testing for RISC-V. In *26th Asia and South Pacific Design Automation Conference (ASPDAC '21), January 18–21, 2021, Tokyo, Japan.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3394885.3431584

1 INTRODUCTION

RISC-V [32, 33] is an open and free *Instruction Set Architecture* (ISA) and as such has evolved from academic research into mainstream adoption. RISC-V is very modular by defining 32, 64 and 128 bit integer ISAs in the base specification. Moreover, to the base ISA fixed standard extensions can be added. This includes for instance integer multiply/divide, single- and double precision floating point, atomic memory operations among many others. Finally, custom instructions can be added to create application specific solutions. However, there is a strong risk when allowing such an enormous flexibility: fragmentation of the RISC-V ecosystem. Simply speaking, if designers customize their RISC-V core too much, the benefits of the common ecosystem are lost as (a) significant adoptions in the tools become necessary and (b) sharing of implementations, SDKs, etc. is hindered. This very important problem is addressed with *compliance testing*. In contrast to verification, which attempts to prove that an implementation is correct, compliance testing attempts to show that an implementation meets the standard and thus ensures compatibility with the RISC-V ecosystem. More precisely, compliance testing checks whether registers are missing, modes are not there, instructions are absent, corner-case scenarios are working as specified, and is performing basic functional sanity-checks for each instruction.

The importance of compliance testing has been recognized very early by the RISC-V foundation and as a consequence the compliance task group has been formed [7]. Intensive ongoing discussions are taking place on how to proceed in order to develop suitable tools, models, and methodologies to solve the RISC-V compliance testing problem [2]. The official approach pursued by the task group is a *Compliance Test-suite* (CT) [7]. The quality of the test-suite is monitored by leveraging functional coverage metrics (and they guide further development of the test-suite) and recent reports indicate that a very high quality of the base RV32I test-suite has been reached [3]. However, it is important to cross-validate these results using different methods because it is very challenging to ensure that all relevant functional behavior is comprehensively tested.

In this paper, we propose a mutation-based approach to boost RISC-V compliance testing by providing more comprehensive testing results[1]. Therefore, we define mutation classes tailored for RISC-V to access the quality of the CT. In particular, we consider mutations in the execution unit that mask register and immediate values, modify constants, replace the accessed registers, immediates and operators as well as check for exception side-effects. Based on

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPDAC '21, January 18–21, 2021, Tokyo, Japan
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-7999-1/21/01…$15.00
https://doi.org/10.1145/3394885.3431584

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

these mutation classes we show that several serious gaps in the official CT do exist, and the most recent specification-based compliance testing approach [20] is unable to detect the critical mutants. Therefore, we provide a symbolic execution framework to generate new test-cases that kill the remaining mutants. Our experimental results demonstrate the effectiveness of our approach. Our final extended CT closes all gaps and detects bugs in RISC-V simulators.

2 RELATED WORK

Compliance testing for RISC-V is an emerging research area. Even the first steps towards the official CT started only in 2018, and hence only recently gained significant momentum. Therefore the number of research papers covering this topic is still limited. We are only aware of [18, 20] that specifically target the compliance testing problem. [18] leverages coverage-guided fuzzing to generate the CT. It primarily focuses on negative testing, i.e. illegal instructions and exceptions, and thus complements our mutation-based approach. [20] defines a test-suite specification mechanism and leverages constraint solving techniques to generate a CT according to the specification rules. It focuses on positive testing aspects and the specification mechanism is well suited to reason about different register and immediate values. In contrast to our mutation-based approach, which considers the problem from the angle of inserting bugs, [20] provides a value-driven specification, i.e. for instance what value ranges are expected for a specific instruction, and thus test-cases are generated with another objective.

For the purpose of verification, a set of test generation approaches specifically targeting RISC-V have also emerged recently [1, 10, 22]. The Scala-based *Torture Test* generator [1] generates tests by integrating pre-defined randomized test-sequences. *RISCV-DV* [10] by Google leverages SystemVerilog in combination with UVM (*Universal Verification Methodology*) to generate RISC-V instruction streams based on constrained-random descriptions. It requires a commercial RTL simulator providing SystemVerilog and UVM support. Finally, [22] utilizes fuzzing techniques to generate randomized instruction streams as platform dependent binary files (ELFs). These approaches are designed to continuously generate (randomized) test-cases for verification purposes. Furthermore, they do not support the compliance testing format.

Beside test-generation methods, there are also a few formal verification approaches for RISC-V. Notable approaches that leverage model checking are *riscv-formal* [8] and the *OneSpin 360 DV* RISC-V verification app [6]. However, both approaches clearly target the verification of an implementation.

Looking beyond RISC-V, several approaches for test-program generation have been proposed for the purpose of verification. For example, they integrate model-based techniques with constraint solving [13–15, 27] or leverage coverage-guided test generation based on Bayesian networks [16] and other machine learning techniques [25] as well as fuzzing [29].

Mutation testing has been intensively investigated over several decades and has its roots in the software domain as a fault-based software testing technique (a survey can be found in [26]). In the context of hardware, approaches based on injecting faults into RTL designs to determine the quality of the test-cases have been proposed, for instance [30]. Mutation-based testing and analysis also found their way into commercial tools, like Certitude from Synopsys. This goes back to [17] and is referred as functional qualification. The principles have been further advanced in [28]. The method generates high coverage input vectors for RTL designs by recording branch coverage controlled via mutated guards during symbolic simulation. Enhancements for guiding the stimuli generation process with mutation analysis have been presented in [34]. However, while all these works identify the weaknesses of the test-stimuli or even improve them, they cannot directly be used to generate compliance tests.

3 BACKGROUND ON RISC-V

In this work we consider the RISC-V base RV32I ISA. It defines a 32 bit core without any extensions. It has 32 general purpose registers x0 to x31 (with x0 being hardwired to zero) each 32 bit width. Instructions are grouped into different classes (i.e. computational, load/store, branch/jump). They access registers (source: RS1 and RS2, destination: RD) and immediates to perform their operation. Immediates are available in different sizes and signed/unsigned interpretation. For example, I_imm is a signed 12 bit immediate, thus has a value range of [-2048,...,2047]. This allows to define instructions such as *ADDI x1, x2, 128* which adds the value of x2 (RS1) with 128 (I_imm) and stores the result in x1 (RD). Format and semantics (for the base ISA and extensions) are defined in the unprivileged ISA specification [32].

In addition, the privileged (architecture) specification [33] covers further important functionality required for environment interaction and operating system execution (for example virtual memory support and interrupt handling). In particular, it defines CSRs (*Control and Status Register*) and special instructions to access them.

4 MUTATION-BASED COMPLIANCE TESTING

This section presents our mutation-based approach to boost RISC-V compliance testing. We start with an overview (Section 4.1) and then present our mutation classes tailored for RISC-V (Section 4.2). Next, we provide more details on how to kill mutants based on symbolic execution (Section 4.3) and generate compliance test-cases based on the solutions (Section 4.4). Our approach thus complements the existing CT infrastructure.

4.1 Overview

Fig. 1 shows an overview on our approach. Starting point is a set of mutation classes and a reference *Instruction Set Simulator* (ISS). Each mutation class describes how to generate a set of mutants. Each mutant represents a mutation in the reference ISS, i.e. a mutant is a mutated ISS. In this work we focus on mutations in the execution unit of the ISS (we provide more information on the mutation classes in Section 4.2).

In the first step, the set of mutation classes is processed in combination with the reference ISS to generate a set of mutants. Then, each mutant is checked against the CT. Killed mutants are considered uninteresting and thus filtered out. A mutant is killed, if it is detected by at least one test-case of CT (i.e. the mutant and the reference produce different results on this test-case).

The (still) alive mutants are passed to the mutation solver (Step 3). It leverages a symbolic execution framework to generate specific

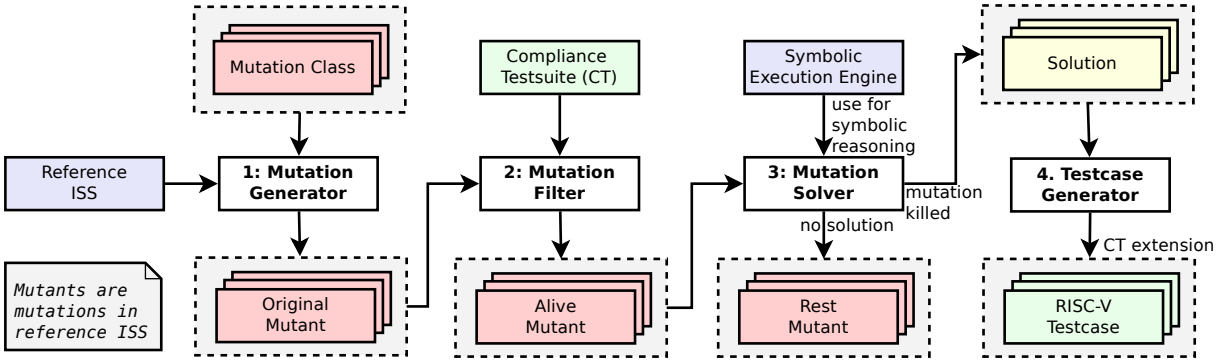

Figure 1: Overview: mutation-based approach for RISC-V compliance testing

inputs to kill the mutants. Such an input is called a *solution*. It is possible, that no solution exists, because the mutation has no influence on the result (e.g. replacing an integer X with X+0).

A solution is essentially a snapshot of the relevant execution state and a single instruction that will show a difference in the output behavior between the reference ISS and mutated ISS when executed from that state.

In the last step the solution is transformed into a RISC-V compliance test-case. The generated test-suite complements the CT in providing stronger test coverage. Furthermore, it allows to access the quality of the existing CT and reveal coverage holes.

4.2 Mutation Classes

We consider mutations in the execution unit of the reference ISS. In total, we define nine mutation classes with rules tailored for ISS mutations in the context of RISC-V:

- M1 Modify the value of a source register by applying a mask that sets a single bit to zero. This ensures that all register bits are comprehensively tested.
- M2 Similar to M1 but applied on immediate values. The range of the mask is set to cover the whole immediate range.
- M3 Replace a constant with another one by adding or removing a bit from the left or right side. This ensures that similar constants are used.
- M4 Replace a load instruction with another one (e.g. load word with load byte) or replace a store instruction, respectively.
- M5 Replace any of the RS1, RS2 or RD register with each other, e.g. ADD x1, x2, x3 could be mutated to ADD x2, x2, x3 by replacing RD=x1 with RS1=x2. Each register can also be replaced with the hardwired zero register.
- M6 Similar to M5 but applied to immediates, e.g. replace I_imm with S_imm, etc.
- M7 Replace unary operations {!,-,~} with each other or remove the operation. Another mutation is to replace the new and the current PC, i.e. which will cause being of by one instruction in jumps and return address computations.
- M8 Replace binary operations with each other. Distinguish between computational operations and relational operations.
- M9 Move a trap check to the end of the instruction execution, i.e. this will cause side effects (writing a register) to apply before taking the trap.

```
1  switch (op) { // execute instruction in ISS
2  //...
3      case BEQ: // Branch EQual instruction
4          if (mutation_begin()) { // mutated paths
5            if (regs[instr.rs1()] == regs[instr.rs2()])
6              pc = last_pc - instr.B_imm();
7          } else { // unmutated paths
8            if (regs[instr.rs1()] == regs[instr.rs2()])
9              pc = last_pc + instr.B_imm();
10         }
11         mutation_end();
12     break;
13 //...
14 }
```

Figure 2: Example mutation for the BEQ instruction.

Each mutation class defines a set of mutations. Only a single mutation is selected and applied at a time.

4.3 Killing Mutations via Symbolic Execution

Killing a mutation is considered on a per instruction basis. Fig. 2 illustrates the basic idea using the BEQ instruction as an example. The normal instruction execution code (Line 8-9) is duplicated and the mutation is applied (Line 5-6). In this example the binary operator plus (Line 9) is replaced by minus (Line 6). The resulting code block is wrapped with calls to the artificial *mutation_begin* and *mutation_end* functions (Line 4 and Line 11, respectively). These functions mark the mutation area and are recognized by the symbolic execution framework.

To find a solution that kills the mutation, we provide a single symbolic instruction to the ISS and make the ISS register file unconstrained symbolic. Additionally, we overwrite ISS functions which are used to access memory, thereby making load/store instruction operate on symbolic values. The program counter is set to a fixed concrete value. Based on the symbolic instruction, a path to mutation_begin is searched. At this point, we distinguish two sets of paths trough the marked code block: The set of mutated paths and the set of unmutated paths based on the return value of mutation_begin (which is controlled by the symbolic execution engine). Then, we calculate the Cartesian product of these two sets, thereby enumerating all possible combinations of unmutated and mutated paths. For each resulting combination, we add additional solver constraints and search for a solution that kills the mutation. Through these additional constraints we ensure that only aligned

```
1  /**[ BEQ template ]****/
2  //..init relevant regs..
3  J to_beq
4  ADDI x1, x1, 1
5  //..cover branch range..
6  ADDI x1, x1, 1
7  J halt
8  to_beq:
9  BEQ RS1, RS2, B_imm
10 ADDI x2, x2, 1
11 //..cover branch range..
12 ADDI x2, x2, 1
13 halt:
14 //..exit sequence..
```

```
15 /**[ LW template ]*****/
16 LA RS1, data_middle
17 LW RD, (I_imm)RS1
18 halt:
19 //..exit sequence..
20 data_begin:
21 .byte b0
22 //...
23 .byte b2047
24 data_middle:
25 .byte b2048
26 //...
27 .byte b4095
28 data_end:
```

Figure 3: Example template for the BEQ and LW instruction.

memory addresses are considered and non-terminating self loops are avoided. A mutation is killed, if a difference in the output behavior is observed. For this reason, we compare the register values, addresses used to access memory, values used to read (or write) memory and the PC. If a solution has been found, concrete values for the register file and memory addresses as well as used memory values are provided (if any).

4.4 Test-case Generation

Test-case generation transforms a solution into a test-case. We use a custom template for the RISC-V compliance test format. The template contains case distinctions for different instruction types.

For a computational instruction, we simply generated instructions to initialize all registers according to the solution and then put the generated instruction into the (RISC-V assembly) test-case. The difference between a mutated and unmutated ISS will be observed in the resulting register file.

For branches and jumps we generate code which ensures that a difference in the register outputs will be observed in case the resulting PC differs. Fig. 3 shows the test code template. First, the registers are initialized based on the solution. Then, a jump to the instruction (BEQ in this case) is performed. Before and after the generated instruction, we add enough enough ADDI instructions to cover the maximum possible range of the 12-Bit branch immediate. Different destination registers are used in the forward and backward ADDI to ensure that a different result is always produced (for negative/positive immediates). Since difference in output behaviour of branch instructions can only be observed in the PC, we know that the PC of a mutated/unmutated ISS will differ if a solution was found by our symbolic execution engine.

For load and store instructions the generated solution contains a concrete memory base address (as stored in the RS1 register), the relative immediate offset and used memory values. Fig. 3 illustrates the test generation for the LW (Load Word) instruction. First, the fixed RS1 base address is replaced by a label (Line 16, LA = Load Address), to keep the test platform independent (though it should resolve to the same address on the reference ISS). Then, the LW instruction is executed (Line 17). Before and after the label, enough data is placed to cover the whole immediate offset range. The memory is initialized with the concrete memory values provided by the solution. Thus, a difference will be observed in the RD register between the mutated and unmutated ISS (for a store instruction, the difference would be observed in the memory).

5 EXPERIMENTS

We have implemented our proposed approach for mutation-based compliance testing using the ISS of the open source RISC-V VP [19, 21, 23] as reference ISS. We focus on the base RV32I ISA, use the mutation classes described in Section 4.2 as foundation for the testing process and leverage *angr* [4, 31] as symbolic execution backend. Beside the official CT [7], we also consider the specification-based CT from [20] in this evaluation. All experiments have been performed on a Linux system with an Intel i5-7200U processor. We start with a results overview and then present results on the official, specification-based and our mutation-based CT.

Result Overview. Table 1 shows the results. The first two columns show the mutation class and the number of mutants in this class (column: #mutants). The remaining columns report results on: 1) the official CT, 2) the specification-based CT and 3) our symbolic execution framework, to kill the mutants. These three steps are applied one after another and only mutants that are still alive (i.e. not killed) are passed to the next step. The columns #killed and #alive report the number of killed and still alive mutants after each step. In addition, the runtime in seconds is reported for each step.

Official CT. It can be observed, that the official CT already provides strong results in killing the mutants. An average of around 92% of mutants across all nine mutation classes is killed. It takes around four hours to process all mutants with CT. This corresponds to around 6 seconds per mutation. Most of the time is spend in executing the 48 tests one after another, which involves loading the ELF test files as well as writing and comparing signature results files and glue code written in Python. Though, performance optimizations would be possible in this area (for example by pre-loading ELFs and signature files) when they become necessary.

While only around 8% of the mutants are not killed, careful analysis of these alive mutants revealed several interesting error classes that are not detected by CT. We discuss them in the following:

(1) RISC-V provides three register-based shift operations SLL (Shift Left Logical), SRL (Shift Right Logical) and SRA (Shift Right Arithmetic). They shift the value of register RS1 by the value of register RS2 and store the result in register RD. According to the RISC-V specification, only the lower 5 bit of the RS2 register should be used for shifting, i.e. the RS2 register is masked by 0b11111. However, changing the mask to also use upper bits of RS2 for shifting is not detected by CT. We observed similar problems on the immediate-based shift instructions.
(2) RISC-V provides six branch instructions that perform a conditional relative jump. The BEQ (Branch if registers are EQual) instruction is not sufficiently tested by CT. Most of the mutations that mask the branch immediate have not been found. Even using a mask of 0b1111 on the branch immediate is not detected. That means backward jumps are not tested (since the above mutation cuts away the sign bit) and only very small forward jumps (up to 2 instructions) are tested.
(3) Another interesting class of undetected mutations is moving the PC alignment check to the end of the jump instruction. The JAL (offset-based relative jump) and JALR (register-based absolute jump) instructions store the return address

Table 1: Experiment results

Mutation Class	#mutants	CT: Official			CT: Spec-based			Symbolic Execution	
		#killed	runtime	#alive	#killed	runtime	#alive	#killed	runtime
M1: Mask Register	1395	1249 [89.5%]	8700s	146	79 [54.1%]	70216s	67	67 [100%]	5954s
M2: Mask Immediate	374	343 [91.7%]	2375s	31	4 [12.9%]	12536s	27	27 [100%]	2080s
M3: Replace Constant	27	23 [85.2%]	141s	4	4 [100%]	1926s	0	/	/
M4: Replace Load / Store	26	26 [100%]	136s	0	/	/	0	/	/
M5: Replace Register	243	238 [97.9%]	1344s	5	3 [60.0%]	2609s	2	2 [100%]	149s
M6: Replace Immediate	96	95 [99.0%]	518s	1	0 [0%]	568s	1	1 [100%]	105s
M7: Replace Unary Operation	21	19 [90.5%]	169s	2	1 [50.0%]	1114s	1	1 [100%]	92s
M8: Replace Binary Operation	266	265 [99.6%]	1497s	1	0 [0%]	557s	1	1 [100%]	92s
M9: Move Trap Check	7	5 [71.4%]	36s	2	0 [0%]	1114s	2	2 [100%]	160s
Total	2455	2263 [92.2%]	14916s	192	91 [47.4%]	90640s	101	101 [100%]	8632s

into the RD register when performing the jump. However, in case the jump address is misaligned, a trap is triggered and no side effects should occur, i.e. RD should not be modified. This common class of errors is not detected by CT.

(4) One more interesting undetected error is storing any wrong return address for the JAL and JALR instruction.

The remaining alive mutants correspond to special computational cases such as masking specific bits from a register or immediate. It certainly makes sense to strengthen the CT to kill them as well, but they are less important compared to the above cases, which have a much higher potential to find implementation bugs.

Specification-based CT. The specification-based CT contains additional 8900 tests that cover a large set of different immediate and register values as well as register access combinations for each instruction. Due to the large number of tests (and complex execution infrastructure, which requires around 10 minutes to execute the test-suite once), the processing time is very high with around 23 hours. This CT kills around half of the remaining 192 mutants. It partly closes the gaps (1) and (2), but the gaps (3) and (4) as well as the immediate-based shift problem from gap (1) still remain.

Mutation-based CT. Using our symbolic execution framework, we are able to kill all remaining 101 mutants and close the gaps. These mutants are not easy to kill, as they have remained undetected by both CT test-suites. It takes around 2.5 hours in total (85 seconds on average per mutant). Thus, we obtained a mutation-based CT with 101 focused test-cases to further complement the existing CT infrastructure. We cross-validated it on our reference ISS to ensure that the 101 mutants are indeed killed.

Evaluation. Finally, we evaluated the complete CT infrastructure on five RISC-V simulators: riscvOVPsim [7], SPIKE [12], VP [9], GRIFT [5] and SAIL [11]:

- No mismatches were detected with the official CT.
- Using the specification-based CT, we detected a configuration error in SAIL which causes it to execute RV32I compliance tests with the C extension enabled.
- With our mutation-based CT we additionally found a bug in GRIFT, where the JAL instruction has a side effect in updating the RD register even though an unaligned instruction trap is taken (gap 4). Furthermore, we spotted a potential overshift condition in VP, which is based on undefined behavior in C++ (gap 1).

6 DISCUSSION AND FUTURE WORK

Our mutation-based approach has been very effective in finding several serious gaps in the official CT which can lead to common implementation bugs that remain undetected by CT. We evaluated our approach on a reference ISS and considered mutations in the ISS execution unit for the base RV32I ISA. We envision several directions for future work to extend, complement and further boost our approach. We discuss them in the following.

One of the first steps would be to consider CSRs and additional RISC-V extensions. By leveraging our existing mutation classes and framework, it should be straightforward to integrate additional RISC-V extensions with our approach. An interesting point in this direction would also be to produce a minimized test-suite such that all mutants are still killed. This would be particularly helpful with additional RISC-V extensions, because the number of test-cases can grow significantly with each extension (since all mutation classes are applied in combination with the new instructions).

Another direction is to devise and evaluate the impact of even stronger mutation classes, for example by considering multiple instead of single mutations, on the obtained coverage with respect to the CT and potential bugs found in RISC-V simulators. In this direction it would also be very helpful to optimize the symbolic encoding and integration with the symbolic execution engine (angr here) as well as the mutation generation and CT execution to facilitate fast exploratory experiments with different mutation classes.

Some mutants cannot be killed in a platform independent way because they for example rely on a very specific (hardcoded) memory address or PC value in order to trigger the mutation and the compliance testing setup allows each RISC-V simulator to define its own memory layout by providing a custom linker script (though the existing set of supported simulators mostly use the same memory layout). For example a mutation in the memory access unit might only trigger if the access address is below 0x1000. In case the data memory is placed above this address, the mutation cannot be

triggered and thus cannot be killed. One way to tackle this problem in a platform independent way would be to leverage virtual memory and thus setup (platform independent) virtual code and data memory sections (per test-case) that re-map the (platform dependent) physical memory sections as necessary. However, beside being more complex, this approach would require a simulator with support for virtual memory (which is an advanced feature in RISC-V and typically not available in base configurations).

In this work we focus on mutations in the execution unit. However, conceptually our approach can also supports different error categories. One very interesting part would be to test the virtual memory implementation (typically done by an MMU), since it requires a significant amount of complex initializations. Besides setting up the CSRs to activate virtual memory support, it is necessary to setup appropriate page tables in memory. In addition, the page tables need to be setup in a way to reach a very specific mutation which makes it much more complex. Hence, it would be very interesting to consider extensions in this direction to facilitate comprehensive and automated MMU testing.

An orthogonal direction would be to evaluate our mutation-based approach on different reference simulators to access the impact on the generated test-suite. Going further in this direction, the next step would be to evaluate the generated test-suite on RISC-V RTL cores. It would be interesting to see what kind of bugs are detected at RTL and measure the obtained coverage. In a final step, the mutation-based approach could be applied at RTL to generate test-cases specifically tailored for an RTL core and evaluate the results on different RTL cores.

Finally, it would also be very interesting to leverage a symbolic execution framework, e.g. [24], to perform a (more general) difference-based testing. The idea is to find test-cases that show differences in behavior between different simulators. Such test-cases are certainly interesting, since they pinpoint the RISC-V ISA specification parts that are complex or unclearly formulated. The reason is that two simulators implemented a feature differently and hence disagree on the understanding of the specification in that point (which should be highlighted by CT). Another argumentatively similar direction is to perform a difference-based testing on the same simulator but with different ISA configurations (since it can pinpoint changes between two configurations which may be a source for common bugs due to the high configurability of RISC-V).

7 CONCLUSION

We proposed a mutation-based approach to boost RISC-V compliance testing and demonstrated its effectiveness. Based on our mutation classes, we identified several serious gaps in the *Compliance Test-suite* (CT) and generated new tests to strengthen the CT by closing these gaps. Our approach has also been effective in finding bugs in RISC-V simulators. Finally, we provided an extensive discussion that sketched promising directions for future work.

ACKNOWLEDGMENTS

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 Scale4Edge under contract no. 16ME0127.

REFERENCES

[1] 2017. RISC-V Torture Test Generator. https://github.com/ucb-bar/riscv-torture.
[2] 2019. The Challenge Of RISC-V Compliance. https://semiengineering.com/toward-risc-v-compliance/.
[3] 2019. Imperas delivers highest quality RISC-V RV32I compliance test suites to implementers and adopters of RISC-V. https://riscv.org/2019/11/imperas-delivers-highest-quality-risc-v-rv32i-compliance-test-suites-to-implementers-and-adopters-of-risc-v/.
[4] 2020. angr. https://angr.io/.
[5] 2020. GRIFT - Galois RISC-V ISA Formal Tools. https://github.com/GaloisInc/grift.
[6] 2020. OneSpin 360 DV RISC-V Verification App. https://www.onespin.com/solutions/risc-v.
[7] 2020. RISC-V Compliance Task Group. https://github.com/riscv/riscv-compliance.
[8] 2020. RISC-V Formal Verification Framework. https://github.com/SymbioticEDA/riscv-formal.
[9] 2020. RISC-V Virtual Prototype. https://github.com/agra-uni-bremen/riscv-vp.
[10] 2020. RISCV-DV. https://github.com/google/riscv-dv.
[11] 2020. RISCV Sail Model. https://github.com/rems-project/sail-riscv.
[12] 2020. Spike RISC-V ISA Simulator. https://github.com/riscv/riscv-isa-sim.
[13] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv. 2004. Genesys-Pro: innovations in test program generation for functional processor verification. *D&T* (2004), 84–93.
[14] Brian Campbell and Ian Stark. 2014. Randomised Testing of a Microprocessor Model Using SMT-Solver State Generation. In *Formal Methods for Industrial Critical Systems*, Frédéric Lang and Francesco Flammini (Eds.). 185–199.
[15] Mikhail Chupilko, Alexander Kamkin, Artem Kotsynyak, and Andrei Tatarnikov. 2017. MicroTESK: Specification-Based Tool for Constructing Test Program Generators. In *HVC*.
[16] S. Fine and A. Ziv. 2003. Coverage directed test generation for functional verification using Bayesian networks. In *DAC*. 286–291.
[17] Mark Hampton and Stephane Petithomme. 2007. Leveraging a Commercial Mutation Analysis Tool For Research. In *MUTATION*. 203–209.
[18] Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. Closing the RISC-V Compliance Gap: Looking from the Negative Testing Side. In *DAC*.
[19] Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer.
[20] Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. Towards Specification and Testing of RISC-V ISA Compliance. In *DATE*. 995–998.
[21] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. 2018. Extensible and Configurable RISC-V based Virtual Prototype. In *FDL*. 5–16.
[22] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. 2019. Verifying Instruction Set Simulators using Coverage-guided Fuzzing. In *DATE*. 360–365.
[23] Vladimir Herdt, Daniel Große, Pascal Pieper, and Rolf Drechsler. 2020. RISC-V based Virtual Prototype: An Extensible and Configurable Platform for the System-level. *JSA* (2020).
[24] Vladimir Herdt, Hoang M. Le, Daniel Große, and Rolf Drechsler. 2019. Verifying SystemC using Intermediate Verification Language and Stateful Symbolic Simulation. *TCAD* 38, 7 (2019), 1359–1372.
[25] Charalambos Ioannides, Geoff Barrett, and Kerstin Eder. 2011. Feedback-Based Coverage Directed Test Generation: An Industrial Evaluation. In *Hardware and Software: Verification and Testing*, Sharon Barner, Ian Harris, Daniel Kroening, and Orna Raz (Eds.).
[26] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng.* 37, 5 (Sept. 2011), 649âĂŞ678.
[27] Y. Katz, M. Rimon, and A. Ziv. 2012. Generating instruction streams using abstract CSP. In *DATE*. 15–20.
[28] Lingyi Liu and Shobha Vasudevan. 2011. Efficient validation input generation in RTL by hybridized source code analysis. In *DATE*. 1596–1601.
[29] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. In *ISSTA*. 261–272.
[30] Youssef Serrestou, Vincent Beroulle, and Chantal Robach. 2007. Functional Verification of RTL Designs Driven by Mutation Testing Metrics. In *DSD*. 222–227.
[31] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. (2016).
[32] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley.
[33] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley.
[34] Tao Xie, Wolfgang Mueller, and Florian Letombe. 2012. Mutation-analysis driven functional verification of a soft microprocessor. In *SoC*. 283–288.