Verifying Instruction Set Simulators using Coverage-guided Fuzzing*

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

***Abstract*—Verification of Instruction Set Simulators (ISSs) is crucial. Predominantly simulation-based approaches are used. They require a comprehensive testset to ensure a thorough verification.**

We propose a novel coverage-guided fuzzing (CGF) approach to improve the testcase generation process. In addition to code coverage we integrate functional coverage and a custom mutation procedure tailored for ISS verification. As a case-study we apply our approach on a set of three publicly available RISC-V ISSs. We found several new errors, including one error in the official RISC-V reference simulator Spike.

I. Introduction

In todays design flow *Instruction Set Simulators* (ISSs) play an important role in serving as executable specification for subsequent development steps. Thus, ensuring the correctness of the ISS is crucial as undetected errors will propagate and become very costly. The ISS is an abstract SW model of a processor, typically implemented in C++ to enable a high simulation performance. Formal methods do not scale to complete verification and thus simulation-based methods are employed. They require a comprehensive testset (i.e. stimuli). Manually writing the testcases is not practical due to the significant effort required and pure random generation offers only very limited coverage.

Various approaches have been proposed to improve random generation of processor-level stimuli. Model-based test generators use an input format specification to guide the generation process and can integrate constraints processed by CSP/SMT solver [1]–[3]. In [4] an optimized test generation framework, by propagating constraints among multiple instructions in an effective way, has been presented. [5] proposed to mine processor manuals to obtain an input model automatically. Other notable approaches include coverage-guided test generation based on bayesian networks [6] and other machine learning techniques [7].

Since the ISS is a SW model, semi-formal methods based on dynamic program analysis and constraint solving are applicable. They provide automatic ways to increase code coverage but are still susceptible to scalability issues [8], [9] or impose limitations (wrt. modeling memory access and loops) on the ISS [10].

Recently, in the SW domain the automated SW testing technique *fuzzing* [11] has become a standard in the SW development flow [12], [13]. Traditional fuzzing methods are *generational*, which in spirit are similar to the model-based generation techniques employed in processor-level verification and consequently have also been adopted for the verification of ISSs [14]. More recent fuzzing approaches in the SW domain employ the so called *mutation* based technique. It mutates randomly created data and is guided by code coverage, hence avoiding the effort to create an input model. Notable representatives in this *Coverage-guided Fuzzing* (CGF) category are the LLVM-based *libFuzzer* [15] and *AFL* [16], which both have been shown very effective and revealed various new bugs [15], [16].

In this paper we propose to leverage state-of-the-art CGF, as used recently in the SW domain, for ISS verification. In addition we propose a novel functional coverage metric (to complement code coverage) and mutation procedure tailored specifically for ISS verification to improve the efficiency of the fuzzer. As a case-study we have implemented our CGF approach with the proposed extensions on top of libFuzzer and evaluated it on a set of three publicly available RISC-V ISSs. We found new errors in every considered ISS, including one error in the official RISC-V reference simulator *Spike*.

II. Background

A. LibFuzzer: LLVM-based Coverage-guided Fuzzing

LibFuzzer is an LLVM-based coverage-guided fuzzing engine. It aims to create input data (binary bytestreams) in order to maximize the code coverage of the DUT. Therefore, the DUT is instrumented by the *clang* compiler to report coverage information that is recognized by libFuzzer. Please note, libFuzzer does not use functional coverage metrics. Input data is transformed by applying a set of pre-defined mutations (shuffle bytes, insert bit, etc.) randomly. The input size is periodically increased.

Technically libFuzzer is linked with the DUT, hence performs so called *in-process* fuzzing, and allows to pass inputs to the DUT as well as receive coverage information back through specific interface functions. Thus, libFuzzer will call the input interface function defined in the DUT (interface shown in

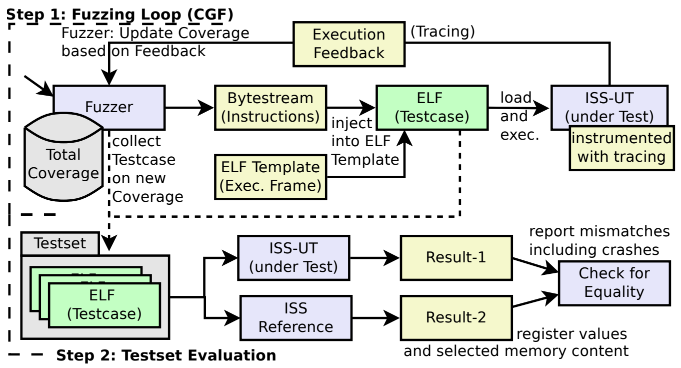
*This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project CONFIRM under contract no. 16ES0565 and within the project SATiSFy under contract no. 16KIS0821K, and by the University of Bremen's Central Research Development Fund, and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

```
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  // allows to run some initialization code once, first time this function is called
    static bool initialized = do_initialization();

    // ... process the input ...

  // tell libFuzzer that the input has been processed
    return 0;
}
```

Fig. 1. Interface function that the DUT provides and that is repeatedly called by libFuzzer (many) times during the fuzzing process.



Step 1: Fuzzing Loop (CGF)
Fuzzer: Update Coverage based on Feedback
Execution Feedback
(Tracing)
Fuzzer
Bytestream (Instructions)
ELF (Testcase)
load and exec.
ISS-UT (under Test)
instrumented with tracing
Total Coverage
collect Testcase on new Coverage
inject into ELF Template
ELF Template (Exec. Frame)
Testset
ELF (Testcase)
ISS-UT (under Test)
Result-1
report mismatches including crashes
Check for Equality
ISS Reference
Result-2
register values and selected memory content
Step 2: Testset Evaluation

Fig. 2. Overview on coverage-guided fuzzing (CGF) for ISS verification

Fig. 1) and the instrumentation performed by clang will call these coverage functions.

B. *RISC-V*

RISC-V is an open and free *Instruction Set Architecture* (ISA). The ISA consists of a mandatory base integer instruction, 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), etc. Thus, RV32IMA denotes a 32 bit core with M and A extensions. For the RV32IMA core all instructions are 32 bit width and have at most two source register RS1 and RS2, and one destination register RD. *Control and Status Registers* (CSRs) are registers serving a special purpose. For example the *mtvec* (Machine Trap-Vector Base-Address) CSR stores the address of the trap/interrupt handler. For a comprehensive description of the RISC-V ISA please refer to the official specifications [17], [18].

III. Coverage-guided Fuzzing for ISS Verification

This section presents our proposed CGF approach and our fuzzing extensions tailored for ISS verification: functional coverage metric (definition in Section III-B and instrumentation to measure it in Section III-C) and a specialized mutator (Section III-D). Functional coverage complements code coverage and improves the thoroughness of the testset especially in detecting computational errors (which depend on operand values and structure). The use-case for our mutator is to introduce specific instruction pattern into the fuzzing process, further guiding the fuzzer and allowing the fuzzer to re-use those pattern in its own mutations and cross-over operations. We start with an overview.

A. *Overview*

An overview of our CGF approach for ISS verification is shown in Fig. 2. Essentially, it consists of two subsequent steps: First a testset is generated by the fuzzer (upper half of Fig. 2), then the testset is used to verify the functionality of the ISS under test (ISS-UT, lower half of Fig. 2) by comparing the execution results with other reference ISSs (can be multiple). In the following we describe both steps in more detail.

1) Fuzzing Loop (CGF): The fuzzer starts with an empty coverage and empty testset. The fuzzer iterates until the coverage goal or the specified time limit is reached. In each step the fuzzer generates a (binary) bytestream. We interpret this bytestream as a sequence of instructions for the ISS under test (ISS-UT). Every such bytestream is transformed into an (ELF-)testcase, by embedding the bytestream into a pre-defined ELF-template[1].

The ELF-template contains prefix and suffix code (execution frame) that is supposed to be executed before and after the actual sequence of instructions. The prefix is responsible to initialize the ISS into a pre-defined initial state. This includes initializing all registers to pre-defined values to ensure that all ISS implementations start in the same state. The suffix is responsible to collect results and stop the simulation. It will write all register values into a pre-defined region in memory (can contain additional content beside the register values) to enable dumping the result of the execution to a file (an ISS typically provides an operation to dump specific memory regions), which can be compared.

The testcase is then executed on the ISS-UT. The ISS-UT generates execution feedback by tracing relevant information. The tracing functionality need to be instrumented into the ISS-UT. The fuzzer will analyze the execution feedback and update its coverage metrics accordingly. We consider structural and functional coverage. As already mentioned, we present our functional coverage metrics and the instrumentation to trace it in Section III-B and Section III-C, respectively. In case the coverage is increased by executing the testcase, the testcase is added to the fuzzer testset.

2) Testset Evaluation: After the testset has been generated, every testcase is executed one after another on the ISS-UT and other reference ISSs. The results are compared and mismatches reported. Please note, not all mismatches are necessarily bugs. The reason is that the fuzzer is not constrained to specific well-defined subsets of the instruction set but considers all possible instructions and sequences of instructions (including illegal instructions). This behavior is intended to check specifically for

[1] Technically, we use a linker script that generates an empty section in the ELF-template. Then we use *objcopy* utility with the *–update-section* argument to overwrite the empty section with the (binary) instruction bytestream.

uncommon (error-) cases. A common source for mismatches are differences in configuration. For example the memory size can be different or some peripheral mapped into the address space (which can not always be easily changed without intrusive modifications). A load/store instruction might then succeed for one ISS and fail for the other. We do not consider such mismatches to be bugs. Thus, mismatches need to be analyzed to check if they relate to bugs in the corresponding ISS.

B. Functional Coverage Metric

We define generic functional coverage metrics for registers and immediates that are applicable to a large set of ISAs. In the following we introduce functional metrics that 1) reason about the instruction structure, and 2) the operand values:

1) Structure Metrics: A testset satisfies the coverage metric **R2** iff for every instruction with one source (RS1) and destination register (RD) at least once the case $RD = RS1$ and at least once the case $RD \neq RS1$ is observed. The coverage metric **R3** extends R2 to instructions operating on three registers (another source register RS2 is used). For R3, each of the four following cases should be observed for every such instruction:

- RS1 = RD $\land$ RS2 = RD
- RS1 $\neq$ RD $\land$ RS2 $\neq$ RD $\land$ RS1 $\neq$ RS2
- RS1 = RD $\land$ RS2 $\neq$ RD
- RS1 $\neq$ RD $\land$ RS2 = RD

R3 can be further extended for ISAs involving more source and/or destination registers.

2) Value Metrics: The metrics **V(Rx)** and **V(Ix)** require that the Rx register and Ix immediate are assigned at least once to each special value in {MIN, -1, 0, 1, MAX}, where MIN and MAX are the smallest and largest possible value of Rx and Ix, respectively. For immediates, that are interpreted as unsigned values, e.g. shift amount, the negative values are omitted from the value set. Both metrics are applicable to every instruction having the Rx and Ix parameter, respectively. For example V(RD), V(RS1) and V(I_imm) are applicable for the RISC-V ADDI instruction (addition of register with immediate).

C. Instrumentation for Tracing Functional Coverage

An ISS typically consists of an execution loop that will fetch the next instruction, decode it, and then execute it. We instrument the execution step to call *begin-fcov-trace* and *end-fcov-trace* function (before/after the actual execution). Both trace functions take three arguments: 1) the instruction fetched from memory, 2) the decoded operation, and 3) a snapshot of the register values (read-only). Using two trace functions allows to capture the register state before (source register values) and after the instruction execution (destination register value). Please note, immediate values are also captured since they are encoded in the instruction itself.

D. Custom Mutations

A mutation is applied on a bytestream (i.e. instructions inside a testcase) and returns a modified bytestream. We propose an additional mutator tailored for ISS verification. Our mutator starts at the beginning of the bytestream and proceeds forward, applying local mutations, until the end of the bytestream is reached. In each step one of the following two mutations is performed (randomly selected):

1) Select a random instruction and inject its opcode bits at the current position into the bytestream, overwriting existing data. This ensures that a legal instruction is used, but the parameters (source registers, destination register, immediate values, etc.) are not modified (still randomized by the fuzzer). For example the ADDI instruction of the RISC-V ISA consists of 32 bit (4 bytes) where the bits [11,7], [19,15] and [31,20] encode the parameters of ADDI: the destination register (stores the addition result), source register (first operand) and immediate (second operand), respectively. All remaining ten bits [14,12] and [6,0] encode the opcode (a fixed value) of ADDI and hence would be injected in case ADDI is selected. Optionally, a randomly selected special immediate value can also be injected into the instruction.

2) Select a random constrained sequence of instructions and inject it into the bytestream, overwriting existing data. A sequence is a list of concrete instructions with some parameters constrained to fixed values and others randomized. A common sequence is to use two instructions to load a large constant value into a register by loading the lower and upper half separately (because typically only small immediates can be encoded in one instruction). In this case the destination register is constrained to be the same for both instructions, but the actual value is randomized (or selected from a set of special values). Sequences are defined by the verification engineer and can be specialized for each instruction set.

IV. Case-Study: RISC-V ISS Verification

As a case study we built our CGF approach on top of *libFuzzer* and verify the RV32IMA ISS extracted from the publicly available RISC-V Virtual Prototype (VP) [19][2]. We denote this ISS as *ISS-UT*. As reference we use the following two ISS:

1) *Spike*, the official RISC-V ISA reference simulator [23].
2) *Forvis*, an ISS implemented in Haskell aiming to be a formal specification of the RISC-V ISA [24].

We have found errors in ISS-UT as well as both reference ISSs. We first describe the evaluation setting and libFuzzer integration. Then we show our detailed evaluation results.

A. Evaluation Setting and LibFuzzer Integration

We have instrumented ISS-UT with the *clang* compiler to trace branch coverage information. We manually (can be automated by an LLVM pass) added a call to the *begin-fcov-trace* and *end-fcov-trace* function to trace functional

[2]The VP is implemented in standard-compliant SystemC and TLM-2.0 [20], [21] and is designed as extensible and configurable platform with a generic bus system [22]. It integrates a RISC-V RV32IMA core and a PLIC-based interrupt controller together with an essential set of peripherals. The VP is available under MIT licence. Visit www.systemc-verification.org for our most recent VP-based approaches.

TABLE I
EVALUATION RESULTS - ALL EXECUTION TIMES ARE REPORTED IN SECONDS - [V1..V7] MEANS ALL 7 ERRORS V1 TO V7 HAVE BEEN FOUND.

Tests / Generator	Time (sec.)	Coverage (measured by instrumenting ISS-UT)									Errors found in ISS		
		Branch Cov.	Functional Cov.								ISS-UT	Spike	Forvis
			R1	R2	R3	V(RS1)	V(RS2)	V(RD)	V(I_imm)	V(I_shmt)			
T1: RISC-V ISA Tests	2	90.24%	58.57%	61.70%	50.00%	14.29%	2.70%	7.55%	8.33%	100.00%	[V1..V3]	/	/
T2.1: RISC-V Torture 1000	5280	74.30%	2.17%	66.67%	69.23%	58.82%	91.43%	52.17%	9.09%	100.00%	V1,V2	/	H2
T2.2: 5000	26108	74.30%	2.17%	66.67%	69.23%	58.82%	91.43%	56.52%	18.18%	100.00%	V1,V2	/	H2
T2.3: 10000	52168	74.30%	2.17%	66.67%	69.23%	58.82%	91.43%	56.52%	63.64%	100.00%	V1,V2	/	H2
T3: Cov.-guided Fuzzing	32492	100.00%	100.00%	100.00%	100.00%	98.21%	100.00%	81.13%	100.00%	100.00%	[V1..V7]	S1	H1,H2

```
if ((op == ADDI) && (instr.RD() == instr.RS1()))
    features.add(1);
if ((op == ADDI) && (instr.RD() != instr.RS1()))
    features.add(2);
if ((op == ADDI) && (regs[instr.RS1()] == REG_MAX))
    features.add(3);
if ((op == ADDI) && (instr.I_imm() == 0))
    features.add(4);
// etc ...
```

Fig. 3. Concept of mapping functional coverage trace information to features.

coverage information before/after executing one instruction, respectively. The branch coverage information is already recognized and collected by libFuzzer, and is internally mapped to unique *features* (i.e. integer values identifying the coverage information). We have extended libFuzzer to also support the proposed functional coverage information, carefully making sure to generate unique features, i.e. not interfere with the branch coverage collection[3]. Conceptually, we implement it as a chain of consecutive if-blocks matching the cases of the functional coverage metrics and mapping each case to a unique feature for every instruction, as shown in Fig. 3.

We generate this code automatically from a *python* script based on the RISC-V ISA specification. We also use a slightly optimized representation based on a switch-case construct to distinguish different opcodes more efficiently.

We integrated our custom mutator into libFuzzer in a way to be randomly selected with the same probability as any of the existing mutators. As discussed in Section III-D we define instruction sequences to load a random or special value into a register and also allow to inject special immediate values.

In addition to the metrics defined in Section III-B, we propose the RISC-V specific metric **R1**. R1 requires that every instruction with a destination register (RD) is executed with the case RD=0 and with the case RD$\neq$0. This metric is useful to check that the hardwired zero register is not erroneously overwritten for some instruction. The metrics V(I_imm) and V(I_shmt) cover immediates used in computational operations.

We integrate the official RISC-V ISA tests [25] and the RISC-V Torture testcase generator [26] in the comparison to further evaluate the effectiveness of our fuzzing approach. All experiments are performed on a Linux system with an Intel Core i5 processor with 2.4 GHz and 32 GB RAM.

[3]Technically, we essentially added a new tracing collector class and extended the size of the feature representation data type. This step required adapting the fuzzer core to use sparse data structures, instead of fixed sized arrays, due to the increased feature state space.

B. Evaluation Results and Discussion

Table I shows the results. The table is separated into four main columns. The first column (Tests/Generator) reports which testset is used or how it has been obtained, respectively. The second column (Time) shows the time in seconds to generate (9h timeout for our fuzzer) and execute the respective testcases. Please note, the RISC-V ISA tests are directed tests that are hand-written and thus do not require a generation step. The third column shows the branch- and functional coverage obtained by running the respective testset. The coverage is measured based on the instrumented ISS-UT. The fourth and last column shows which (and how many) errors have been found by each approach. Table II lists and describes the errors we have found in some more detail.

Already the RISC-V ISA tests can be very effective in detecting errors (Table I, row T1). The testset revealed three errors (V1, V2 and V3) in ISS-UT. Furthermore, the testset is very compact and thus can be executed very fast. However, being hand-written, the testset is susceptible to miss relevant behavior, which is also reflected by the obtained coverage values. Furthermore, significant manual effort is required in order to create the testset.

Torture tests reveal one additional error in Forvis (Table I, rows T2.X). It can be observed that gradually increasing the testset from 1,000 (Table I, row T2.1) to 10,000 (Table I, row T2.3) randomly generated tests does only slightly increase the coverage. The reason is that Torture receives no execution feedback, hence every test is generated independent of the previous ones.

Our fuzzer is able to detect all previously shown errors and finds six additional errors (4 in ISS-UT, 1 in Spike and 1 in Forvis), see (Table I, column *"Errors found in ISS"*, row T3). Most of these errors relate to dealing with different forms of illegal instructions in different steps of the execution process. Besides being coverage-guided, a major benefit of our fuzzer is being not constrained to some specific instruction subset, as for example Torture (and hence could not detect the errors our fuzzer did, independent of the number of testcases generated). In particular the fuzzer operates on the binary level, thus it can be used to check for errors that might even be masked by a compiler/assembler (as they do not generate

TABLE II
DESCRIPTION OF ALL ERRORS WE HAVE FOUND IN EACH ISS.

ISS	Error description
Spike	**S1:** Decoder error, which allows illegal instructions to be interpreted as *FENCE* or *FENCE.I* instruction. The reason is that not all opcode bits are present in the decoding mask.
Forvis	**H1:** Erroneously allows to access CSRs, representing counters for hardware performance monitoring (e.g. *cycle* CSR), from a lower privileged execution mode without first enabling the access (by setting the *mcounteren* and *scounteren* CSR).
	H2: The *REMU* instruction, which computes the remainder of a division, fails because it performs a 64 bit operation even though the ISS is configured to run in 32 bit mode.
ISS-UT	**V1:** Multiplication erroneously dropping the upper 32 bit of the multiplication result for large operands due to working on 32 instead of 64 bit operands.
	V2: Overwriting the hardwired *zero* register with a non-zero value (i.e. instruction with destination register RD=0 and non-zero result). Such an instruction is normally not generated by a compiler and thus can be easily missed.
	V3: CSR access instructions fail to update the CSR in case the source (RS1, contains value to be loaded into CSR) and destination (RD, will receive current value of CSR) register is the same, because in this case RS1=RD is overwritten before it is read.
	V4: Undetected misaligned branch instruction due to not four byte aligned immediates. Again, the compiler will not generate such branches and thus this error can easily be missed.
	V5: Illegal CSR instruction has side-effects. A CSR access instruction reads the current CSR value into register RD and then writes the RS1 register value into the CSR. In case of a read-only CSR the write access fails and RD is not allowed to be modified.
	V6: Illegal jump instruction has side-effects. The jump instruction safes the current program counter into register RD before taking the jump. However, in case the jump target address is misaligned, RD is not allowed to be modified.
	V7: Various incomplete decoder checks similar to the error found in Spike. The reason is that ISS-UT only checks the instruction opcodes as far as necessary to uniquely identify each instruction (which is revealed by random mutations on the opcode bits).

illegal instructions). This also enables to thoroughly check the instruction decoder unit, which even revealed an error (S1 in Table II) in the RISC-V reference simulator Spike.

It can be observed that our fuzzer maximizes most coverage metrics to 100% (Table I, row T3). Besides V(RS1), which is close to 100%, only the V(RD) metric is below at 81%. The reason is that the value of the destination register depends on the operand values and the operation. Some result values might even be impossible for some operations. We believe that formal methods are required to fully maximize the V(RD) metric. In total our fuzzer generates a testset with 5,160 testcases. The smallest test consists of 1 instruction and the largest of 23 instructions with an average of 3 instructions.

V. DISCUSSION AND FUTURE WORK

In our experiments we observed that our CGF approach has been very effective in finding various errors in the considered ISSs and maximized most coverage metrics to 100%. One exception is the V(RD) metric, which is below at 81%. It depends on the input parameters and thus can be very difficult to reach with simulation-based methods. To close this remaining coverage gap, we plan to investigate formal (verification) techniques at the abstraction level of VPs, e.g. [27], [28], to automatically identify inputs that will cover the missing parts or infer that the missing parts are indeed unreachable (some result values may not be possible in combination with some specific operations).

Another promising direction to complement CGF is to employ constrained random (CR) techniques. CR techniques [29] have been very effective for various system-level use cases covering both functional as well as non-functional aspects, see for example [30], [31]. It is also possible to integrate CR techniques with the fuzzer, by using the CR generated testcases as input seeds for the fuzzer. This might help to guide the fuzzer towards generating longer test sequences (without illegal instructions) and at the same time reducing the number of *false-mismatches*, i.e. where two ISSs report a difference which is due to a configuration mismatch as briefly discussed in Section III-A2[4]. However, the fuzzer still retains the ability of generating completely random instructions (guided by the state-of-the-art fuzzing heuristics) hence the ability to cover rare corner- and error-cases (which has been very effective in our experiments) that might even be masked by a compiler/assembler. For example, the RISC-V Torture test generator only generates valid instruction sequences and thus would

[4]We currently use automated debug scripts in case of a result mismatch that splits the input instruction sequence and repeatedly generates and executes a new ELF file on both ISSs adding one instruction after another. This allows us to pin-point the precise location of the mismatch, which in turn greatly reduces the effort to debug/analyze mismatches (in particular for longer instruction sequences).

not be able to detect some of the errors that our fuzzer did, independent of the number of testcases generated by Torture.

Ideas from [32], to cover the values of output operands, might also be applicable in this context and help in maximizing our functional V(RD) metric. Also, recently there has been a lot of interest in integrating machine learning techniques into the fuzzing process, e.g. [33], [34], which seems very promising to investigate in our application area.

Another important direction for future work is to evaluate the effectiveness of stronger coverage metrics. Path coverage and cross-coverage of functional metrics can be very effective but at the same time very challenging metrics and often impractical due to the large feature state space. Therefore, we plan to consider *selective* path coverage and (functional) cross-coverage. These are only applied to selected code regions, e.g. consider all evaluation paths for each instruction separately in the ISS but not across instructions, and input operand values. This can further improve the verification result while still maintaining scalability.

Finally, we plan to broaden the scope of our evaluation to integrate further architectures and instruction sets, as well as apply our CGF approach to analyse the whole platform instead of limiting the analysis to the ISS component.

VI. Conclusion

In this paper we proposed to leverage state-of-the-art *Coverage Guided Fuzzing* (CGF) for ISS verification. We integrated a novel functional coverage metric (to complement code coverage) and mutation procedure tailored specifically for ISS verification. Our extensions complement the code coverage metric used by CGF and thus improve the efficiency of the fuzzing process. We have implemented our CGF approach with the proposed extensions on top of the LLVM-based libFuzzer and in a case-study evaluated it on a set of three publicly available RISC-V ISSs. We observed that our fuzzer has been very effective in maximizing most coverage metrics and in finding various errors. Fuzzing is particularly useful to trigger and check for corner- as well as error-cases and can complement other testcase generation techniques. We found new errors in every considered ISS, including one error in the official RISC-V reference simulator *Spike*. In our discussion we sketched various directions for future work to further improve and complement our CGF.

References

[1] 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.

[2] 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.

[3] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin, "X-gen: a random test-case generator for systems and socs," in *HLDVT*, 2002, pp. 145–150.

[4] Y. Katz, M. Rimon, and A. Ziv, "Generating instruction streams using abstract CSP," in *DATE*, 2012, pp. 15–20.

[5] W. Ma, A. Forin, and J. Liu, "Rapid prototyping and compact testing of CPU emulators," in *RSP*, 2010, pp. 1–7.

[6] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *DAC*, 2003, pp. 286–291.

[7] 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, pp. 112–128.

[8] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," *SIGPLAN Not.*, pp. 213–223, 2005.

[9] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *SIGSOFT Softw. Eng. Notes*, pp. 263–272, 2005.

[10] H. Wagstaff, T. Spink, and B. Franke, "Automated ISA branch coverage analysis and test case generation for retargetable instruction set simulators," in *CASES*, 2014, pp. 1–10.

[11] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, pp. 32–44, 1990.

[12] "Oss-fuzz - continuous fuzzing for open source software," https://github.com/google/oss-fuzz, 2018.

[13] "Microsoft security development lifecycle," https://www.microsoft.com/en-us/sdl/process/verification.aspx, 2018.

[14] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi, "Testing CPU emulators," in *ISSTA*, 2009, pp. 261–272.

[15] "libFuzzer - a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, 2018.

[16] "american fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2018.

[17] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: User-Level ISA*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2017.

[18] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2017.

[19] "RISC-V virtual prototype," https://github.com/agra-uni-bremen/riscv-vp.

[20] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.

[21] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.

[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] "Forvis: A formal RISC-V ISA specification," https://github.com/rsnikhil/RISCV-ISA-Spec.

[25] "RISC-V ISA tests," https://github.com/riscv/riscv-tests.

[26] "RISC-V torture test generator," https://github.com/ucb-bar/riscv-torture.

[27] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Verifying SystemC using intermediate verification language and stateful symbolic simulation," *TCAD*, 2018, (early access).

[28] ——, "Compiled symbolic simulation for SystemC," in *ICCAD*, 2016, pp. 52:1–52:8.

[29] J. Yuan, C. Pixley, and A. Aziz, *Constraint-based Verification*. Springer, 2006.

[30] F. Haedicke, H. M. Le, D. Große, and R. Drechsler, "CRAVE: An advanced constrained random verification environment for SystemC," in *SoC*, 2012, pp. 1–7.

[31] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Towards early validation of firmware-based power management using virtual prototypes: A constrained random approach," in *FDL*, 2017, pp. 1–8.

[32] M. Aharoni, S. Asaf, L. Fournier, A. Koifman, and R. Nagel, "Fpgen - a test generation framework for datapath floating-point verification," in *HLDVT*, 2003, pp. 17–22.

[33] "Neural fuzzing: applying DNN to software security testing," https://www.microsoft.com/en-us/research/blog/neural-fuzzing/.

[34] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," 2017, pp. 50–59.