TaPaFuzz - An FPGA-Accelerated Framework for RISC-V IoT Graybox Fuzzing

Florian Meisel[1 [0000−0002−4671−9526]], David Volz[1 [0000−0002−1031−2695]], Christoph Spang[1 [0000−0003−1606−4474]], Dat Tran[1 [0000−0001−7483−3687]], and Andreas Koch[1 [0000−0002−1164−3082]]

Embedded Systems and Applications Group (ESA), Technical University of Darmstadt,
Karolinenplatz 5, 64289 Darmstadt, Germany,
[meisel, volz, spang, koch]@esa.tu-darmstadt.de

Abstract. Fuzz testing, which repeatedly executes a given program with auto-generated random inputs, and records its dynamic control flow, aims to discover sources of unexpected program behavior impacting security, which can then be fixed easier by directed developer effort. When targeting IoT devices, fuzzing faces the problem that the small IoT processors often lack the observability required for fuzzing, e.g., a high-performance trace port, while software-emulation on a faster host CPU is often slow, and compilation of the IoT application to a different ISA for faster native execution on the host introduces inaccuracies in the fuzzing process. To overcome all three of these drawbacks for RISC-V-based IoT processors, which are expected to dominate future IoT applications with their lack of ISA licensing costs, we modify an open-source RISC-V core for use in an FPGA-based hardware-accelerated fuzzing system. Our fuzzer has demonstrated up to *four times* the performance of the state-of-the-art QEMU-based fuzzing tool AFL++, even when running on very fast x86 host processors clocked at 4.95 GHz.

Keywords: Security · Fuzzing · LibAFL · TaPaSCo · RISC-V · Coverage

1 Introduction

With the global number of IoT devices continually rising, a single security vulnerability may result in thousands of affected devices at once [2]. To prevent attackers from quickly accumulating large nets of devices under their control, software security is key. Testing the device firmware for issues can help in detecting many potential weaknesses, but also increases development costs and is too often deemed infeasible. Automatically generating test cases by using a fuzzer framework is one way to effectively search for vulnerabilities in the firmware of such devices. A fuzzer can find vulnerabilities in a *target* program by repeatedly executing it using inputs from a generated *corpus* of inputs. The fuzzer traces the Control Flow (CF) of the running program, detecting invalid program states (such as crashes, timeouts or memory leaks) in the process.

Fuzzing the firmware of a RISC-V IoT device on x86 hardware, however, comes with drawbacks, which are addressed in this work by modifying an existing RISC-V core to support fuzzing in hardware on FPGA:

Precision: Due to ISA differences, cross-compiled RISC-V program binaries differ from their x86 pendants. These differences originate for example from different instruction mappings, potentially changing addressing, word width, and compiler backend optimizations. Fuzzing a program in the host computer's native ISA yields a chance of finding program bugs that would not actually apply to the RISC-V version. On the other hand, it might miss bugs that would occur only in the RISC-V ISA. Fuzzing the original RISC-V program binary thus has a better chance of precisely finding the relevant bugs.

Emulation Overhead: Emulating an ISA results in a huge runtime overhead. The AFL++ documentation estimates x2 to x5. We measured x20 overheads, and assume the difference to be due to the lack of advanced AFL++ features, such as *persistent mode*, which is not available yet on the RISC-V platform [3]. In contrast, native execution carries *no* emulation overhead.

Monitoring Overhead: Graybox software fuzzing frameworks implement their CF monitoring by patching additional function calls into the program to be tested (target), causing interrupts and overheads at runtime. As an alternative, monitoring can be implemented in hardware running in *parallel* to the actual software execution, ideally with no additional runtime overhead.

Our main contributions are:

- Seamless integration of the FPGA accelerator into an existing software fuzzer framework. Its usage becomes as easy as using a plain ISA-emulating fuzzer.
- Compared to prior work, we rely on hardware extensions instead of software patches, allowing to fuzz the original program in real time. We contribute a new hardware unit, which is being attached to a RISC-V processor core for monitoring and compressing the target program's CF events.
- An AXI wrapper for legalizing aborted AXI transfers, which would otherwise hang due to random partial design resets occurring between fuzzer job runs. As the wrapper operates solely on the AXI and reset interfaces, it is portable across RISC-V core microarchitectures and different AXI components.
- We contribute microarchitecture fixes to the CVA5 RISC-V core, allowing it to fully reset its caches and branch predictors between fuzzer runs.
- We reach up to 4.5x wall clock speedups over the traditional emulation-based methods in job launch rate, and an improvement of five additionally detected CF edges over one hour.

Sections 2 and 3 give background information and related work. Section 4 contains the implementation. Sections 5 and 6 evaluate and conclude.

2 Fundamentals

Fuzz-Testing is a well-established research area, this section can thus only cover fundamentals. For an overview of the current research, we recommend [11].

Definition - Fuzzing: A fuzzer is an application, which iteratively executes a test program with varying inputs. In literature, the program that is being tested is called a *target*. A fuzzer's generation of new target inputs can be either generation- or mutation-based. The set of known input test cases is called *corpus*. For mutation based fuzzers, new inputs mutate from initial *seeds*, which define the starting configuration. Both, generation and mutation based fuzzers, can be aware of the input's legal structure, e.g., a JSON file, which is provided to the target. In addition to program inputs, awareness of the target's internal structure and state helps in increasing coverage. Beyond job execution rate, the rate of coverage growth is also influenced by the search strategy. Finally, fuzzers are categorized into black-, white-, and graybox fuzzers:

Black-, White-, and Graybox Fuzzers: Lacking a feedback loop for program-internal state, blackbox fuzzers are unaware of the target's internal structure. They monitor external behavior, such as crashes, to evaluate the target's state.

Whitebox fuzzers use static code analysis to direct a target's CF towards higher coverage, or to focus on user-defined critical program regions.

In contrast to whitebox fuzzers, graybox fuzzers collect CF information via a feedback loop at target runtime. Typically, this is implemented by instrumenting (patching) the target, which causes significant memory and runtime overheads and may also alter the program's behavior.

The **Graybox Fuzzer Result Aggregation (FRA)** may include different coverage information. First, basic block coverage provides information about which basic block (BB) has been executed. Additionally, the number of BB executions can be counted and visualized in a BB histogram. As an alternative, CF edge coverage tracks information on the actually taken CF edges.

Other coverage approaches are possible, but their benefit depends on the individual fuzzing target. For example, hash digests identifying entire CF paths guide towards high path coverage, which can be reached by mutating just one loop limit. It will find a new path per run, but miss other relevant CF edges.

3 Related Work

Quick EMUlator (QEMU) is a generic machine emulator and virtualizer [4]. QEMU executes non-native Instruction Set Architecture (ISA) programs by software emulation, exploiting dynamic translation to improve performance. For fuzzing, this enables us to fuzz-test software targeting IoT devices in their native ISA, rather than fuzz-testing in x86. All frameworks discussed in this section rely on QEMU to provide the capability of non-native ISA fuzzing.

American fuzzy lop (AFL) is an open-source fuzzing framework from Google that was originally developed by Michal Zalewski [16]. It contains tools and fuzzing operation modes, and supports genetic algorithms for input mutation. AFL uses LLVM and GCC for target instrumentation and alternatively allows binary-only instrumentation. To fuzz non-instrumented targets, AFL will fall back onto a blackbox mode, and rely on crash and hang detection (timeout) for feedback. Additionally, AFL provides a QEMU mode to fuzz non-native ISA targets. The authors generally approximate QEMU mode's runtime overhead between factor 2x and 5x [3]. Beyond its powerful mutation algorithms, AFL is easy to use. To fuzz a target, a user provides the target and a dataset of one or more sample legal inputs. To optimize the re-spawning processes for fuzzer runs, AFL uses faster or a reduced number of fork system calls.

AFL++ is a community fork of AFL with a newer codebase and features [8]. AFL++ exposes a custom mutator API to enable researchers to implement plugins to combine new ideas with existing fuzzing technologies.

LibFuzzer is a fuzzing framework related to the LLVM project [14]. It is integrated into the target binary. The user provides an entry point to the target, from which libFuzzer spawns parallel threads to run with varying inputs. As limitations, the target may not modify global state or provide its own reset.

Real-time: Some fuzzing techniques, e.g., used by AFL++ and libFuzzer, employ compiler transformations to make the application easier to fuzz. This ranges from instrumenting special tracing instructions to CF altering transformations. E.g., CF edges with complex conditions are hard to fuzz, because a specific edge is taken only when all partial conditions are met simultaneously. By splitting the condition over multiple basic blocks, the fuzzer receives more runtime feedback to find inputs that meet all partial conditions. As this transformation affects the runtime of the application, it may be inappropriate for real-time IoT targets.

4 Hardware/Software Co-Designed Fuzzer

This section discusses our hardware/software co-designed fuzzer for RISC-V IoT firmware, and details hard- (Section 4.1) and software (Section 4.5) components.

Hardware: Our hardware component executes the IoT firmware program, captures the execution's edge coverage map, and finally, together with the target's return value, returns it back to the host software.

Software: In an iterative search, the host's *fuzzer software* creates target inputs, launches the actual fuzzer runs, which traditionally would be executed in a RISC-V emulator, on the FPGA accelerator hardware instead, and finally evaluates the program execution to create the next iteration's inputs.

4.1 Hardware - Interconnects (PE Ports)

For a seamless hardware/software integration, we employ the freely available Task Parallel System Composer (TaPaSCo) FPGA abstraction framework [9]. TaPaSCo allows composing SoCs consisting of heterogeneous processing elements onto a wide spectrum of FPGA platforms. The tool also provides the drivers and middleware to communicate between soft- and hardware components.

As can be seen in Figure 1, the TaPaFuzz Fuzzer PE, which contains the actual RISC-V core and FRA logic, has two AXI slave ports. One AXI port enables PE control, e.g., for reset and restart, and upon request, also provides

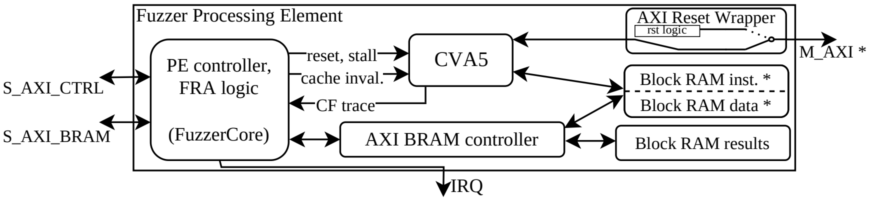


Fig. 1: Simplified layout of the Fuzzer TaPaSCo PE (based on [9,10]). The * marks different memory configurations.

status information. The second AXI slave port allows writing into the PE's memories, which are instruction, data, and the fuzzing result memory.

The PE has an optional AXI master port, which, when being connected to an AXI-DRAM controller, replaces or extends the BRAM-based instruction and data memories that are needed for larger, possibly non-IoT, targets.

Finally, a single-bit interrupt (IRQ) signals a finished (or broken) target execution to the host software. Next, we discuss the PE's internals.

4.2 Hardware - Processor Core

A fuzzer Processing Element (PE) consists of the CVA5 *RISC-V* processor [12,13], a job controller, and FRA hardware.

The PEs rely on CVA5 soft-core processors, which achieve a better performance than many other RISC-V cores when used on FPGA. Instead of a single execute stage, the CVA5 single-issue processor core has multiple independent functional units. This allows it to perform higher latency operations, such as memory loads/stores and divisions, without stalling, assuming that the immediately following instruction does not depend on their writeback results and can be handled by other currently idle functional units [13]. Furthermore, as an alternative to memory bus designs, CVA5's optional BRAM instruction and data scratchpad interfaces considerably reduce memory access latency.

4.3 Hardware - Fuzzer Result Aggregation

General Mechanism For reporting the fuzzing coverage, we implemented an edge coverage FRA (Section 2), which outputs a histogram of taken CF edge transitions (the *fuzzer result map* or *coverage map*). In hardware, it is stored in a configurable size BRAM memory, containing an 8-bit counter per-edge.

While the RISC-V core runs the target software to be fuzzed (here: IoT firmware), the FRA hardware block receives CF events. Branch and jump instructions in flight are detected by their instruction encoding. A hash from the instruction's source and target Program Counters (PCs) is then generated and trimmed to an index into the coverage map, where the 8-bit counter corresponding to that control edge is then incremented. On end of execution, the hardware signals an interrupt to the host, which in turn fetches the coverage map, exception status, and time (in cycles) to generate the next fuzzer run's inputs.

Hashed Indexing of Control Flow Edge Counters The FRA hardware is attached to the CVA5 core via the core's built-in tracing interface, which provides dedicated PC and instruction word outputs. Based on each CF edge's start and target addresses, the hash algorithm creates pseudo-random address-indexes within the coverage map, whose values count each edge's occurrences. The hash algorithm implementation needs to enable high throughput, low risk of collisions, and low hardware overhead, while cryptographic security is not a requirement.

As SHA and similar hash algorithms do have massive hardware overheads and potentially limited throughput, we decided on a suitable lightweight algorithm from the hash prospector repository [15]. It is not cryptographically secure, but runs with only 8 cycles latency and guarantees a throughput of 1 item per cycle, thus does not limit the CF throughput. We feed the 32-bit hash algorithm with the edge's source PC, add the destination PC to an intermediate value of the algorithm to avoid collisions with nearby CF, and trim the result's bit-length depending on the chosen coverage map size to form a valid index.

Due to arbitration between PE-external and internal access, and due to additional latency from the AXI BRAM controller, the overall jump and branch throughput is limited by the read-and-write round trip time to the result memory. A direct stall signal into the processor is triggered if required to not miss CFs.

4.4 Hardware Modifications for more effective Fuzzing

The fuzzing use-case examined here has somewhat unusual requirements on the acceleration hardware due to the many resets that may occur when fuzzing discovers anomalous behavior, which is the goal of the entire fuzzing process. Thus, we need to enable the hardware to efficiently and reliably deal with these many resets. This requires extensions to the internal bus interfaces and, for the CVA5 core, to the cache and branch predictor.

Legalizing AXI Bursts in the Context of Partial Design Resets When a RISC-V core is reset to prepare the next program execution, its internal bus component drops any ongoing transactions, while the external memory bus must remain active and is generally not able to deal with the abruptly aborted transfers.

Between individual executions, while the results are downloaded by the host and the program data memory is refreshed to its original state for the next fuzzing iteration, the fuzzing controller asserts the processor's reset line to return it to a consistent state. As an alternative, it would be possible to keep the core active after a successful execution, which would be similar to the persistent mode of *AFL++*. However, exceptions and especially timeouts would require an external control sequence to actually restart the processor for the next iteration. That sequence would depend on the current processor state and the concrete application. This can be avoided by resetting the softcore processor subsystem.

With the BRAM resources being handled inside the PE, the core can be reset regardless of timing, as the BRAM interface does not have any handshakes or request sequences that need to be finished. However, the AXI4 specification [5] does not include mechanisms to safely abort its handshakes and transactions. As a consequence, for the fuzzer variant allowing access to external memories via an AXI bus, an arbitrary reset could lock up the entire design.

As a solution to this problem, we devise a *wrapper* module placed in-between the core and downstream AXI4 memory bus to *complete* the remaining transactions even when the core resets, as shown in Figure 1. While the core is operating normally, the wrapper combinationally passes through the AXI signals, but keeps a registered copy of each handshake from the core. In addition, it also maintains counters for in-flight requests and the remaining number of beats in a write transaction. The wrapper would only stall write handshakes in case the burst length FIFO or an in-flight access counter would overflow. Note that for the CVA5, no such stalling will occur in practice, as that core sends all write beats *before* starting the next burst.

On a synchronous local core reset, the wrapper module takes over the bus lines from the resetting core. First, pending handshake requests from the core are stabilized using the registered copies until accepted by the bus, if required, to match the AXI specification. Second, for each outstanding write burst, write beats with all bytes disabled (wstrb ← 0) are sent matching the burst length. Finally, the shim waits until all response handshakes (final read beats and write responses) arrive from the bus before notifying the fuzzing controller that the bus is now stable and the core reset can be released.

The shim module is intended to be portable across different cores or other AXI components by making only few assumptions beyond the AXI standard. These assumptions are that the core never issues handshakes for write beats (data) before the write burst request (address, length), and that no more than a configurable number of in-flight read requests (default 15) are sent by the core and accepted by the downstream bus.

Clearing CVA5 Cache Tags If a different program is to be uploaded, or data memory is to be restored to its original state, consistency with caches inside the core needs to be maintained. For the CVA5, the tag memories for instruction and data caches, as well as the branch predictor, persist through a reset. Since restoring post-reset consistency after program upload involves invalidating a significant portion of the set addresses in the processor's caches, we implemented hardware support for accelerated invalidations. This allows to invalidate tags at the rate of one set per clock cycle, regardless of cache associativity.

4.5 Fuzzer Software Architecture

The software portion of our work uses LibAFL [6], an existing library for fuzzer development. Its main authors, Fioraldi and Maier, have also worked on the AFL++ [8] fuzzing engine. While being an independent project, LibAFL uses similar concepts and techniques to AFL++ and AFL, such as a *forkserver* for target execution and a variety of mutators [6,8].

As a key difference, LibAFL provides abstract components for fuzzers but leaves open their concrete composition into an application. Aside from selecting

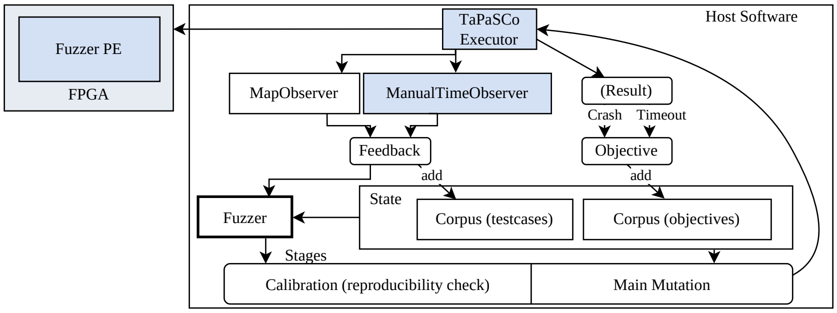


Fig. 2: Simplified layout of the fuzzer software, based on LibAFL's StdFuzzer implementation [7]. Blue color signifies custom or customised components.

fuzzer stages or mutators, the modular and abstract design of LibAFL also enables fuzzer developers to implement new components and, for instance, use alternative input data structures for target programs with existing modules. It is designed to minimize the need for library code forks in custom fuzzer development.

Fuzzer Components A typical LibAFL fuzzer is constructed by instantiating interdependent modules (see Figure 2), loading the initial corpus from disk and calling the *fuzzer* loop. The fuzzer loop, in turn, runs a fuzzer *strategy* consisting of one or several *stages* that, invoked with a *testcase* (program input previously deemed interesting) chosen by a scheduler, implement strategies to *mutate* inputs and running evaluations through an *executor* module.

This executor, which is the key contribution in our software, runs the program with given input data, captures runtime information such as the coverage *map*, and differentiates between normal runs, crashes and timeouts. Captured information is passed on via *observers* to a *feedback* function to determine whether the program inputs should be stored for future iterations. For instance, inputs that uncover a previously unseen CF edge, or other notable coverage and program run time results, would ideally be detected as such and added to the *testcase corpus*.

The *objective* function is defined to detect erroneous behavior, including program crashes and excessive run time (timeouts), and determines whether the corresponding program input should be stored in a separate directory.

Execution Offloading Mechanism For job execution on the FPGA PE, the *libtapasco* runtime library serves as a means to interact with the hardware.

The first step is to retrieve the PE object for the required Fuzzer PE. To do this, *libtapasco* internally queries a status core added during design composition, describing all available PEs and composition details. The program and fuzzer result memory configuration for job dispatching is determined based on the PE type, as it differs between BRAM- and DRAM-based fuzzer configurations.

Our hardware-accelerated executor is, as are the existing executor modules in LibAFL, interfaced to by a single method call. Instead of locally running the program with inputs provided by the caller, it selects the PE, prepares and launches a job on it, waits for job completion, and retrieves the results.

Job preparation involves uploading the initial program data memory with inputs from the fuzzer engine, and also the program instruction memory on the first launch of the PE. For a PE using caches to speed up DRAM access, an explicit invalidation is requested for the data cache and, on the initial instruction memory upload, additional invalidations for the instruction and branch predictor caches. The PE controller is passed arguments via its Memory Mapped Input Output (MMIO) space, including the program input address range, the size of the fuzzer result map to create, the timeout cycle count, and the PE's program memory DRAM address section.

On completion, the result and execution cycle count (i.e. execution time) fields are retrieved. The fuzzer result map is downloaded from the device. The executor returns a success, timeout, or crash status depending on the PE result, and passes the map and elapsed time on to the *Observer* objects in software.

5 Evaluation

In this section, we evaluate hardware overheads and full system wall clock execution time performance. Performance is compared to AFL++. Our FPGA designs are composed with TaPaSCo 2022.1 and Xilinx Vivado 2021.2 [9].

5.1 FPGA Design Resources

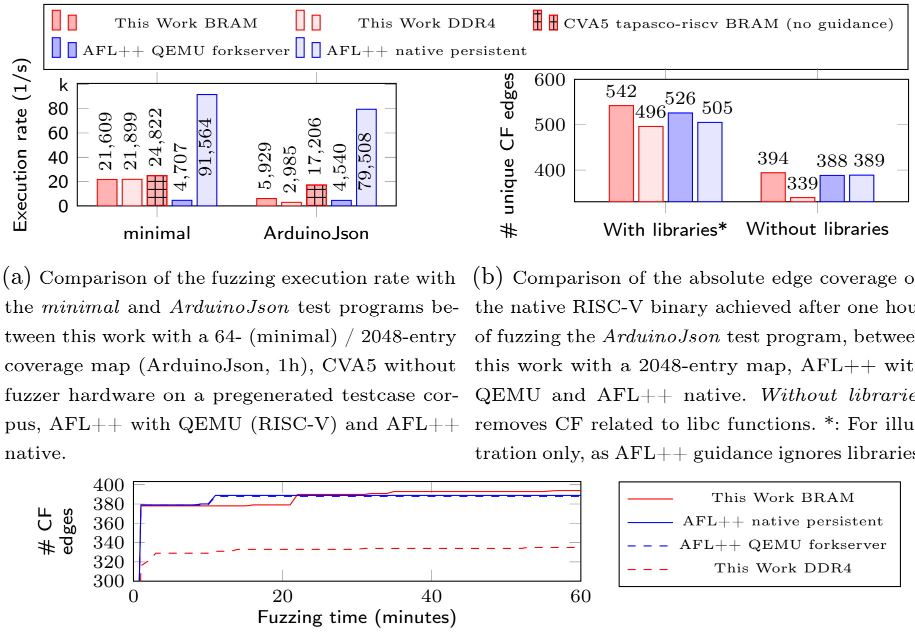
As shown in Table 1, compared to a plain CVA5 RISC-V core PE, the BRAM variant of this work uses 54% − 66% additional LUTs and registers, respectively at a 10 MHz lower frequency. The alternative DRAM version of the fuzzer backed by DDR4 memory requires 2% − 5% more registers and LUTs over the BRAM variant, but reduces the device BRAM resource footprint by 80 KiB, reflecting its tradeoff with additional cache and branch prediction logic enabled in the RISC-V processor, but elimination of scratchpad BRAM.

5.2 Fuzzing Performance

We evaluated single-thread execution performance on the following systems:
FPGA system: Xilinx Alveo U280, PCIe 3.0 x16, AMD EPYC 7443P (4.0 GHz)
AFL++ evaluation system: AMD Ryzen 5900X (4.95 GHz)

Table 1: FPGA utilisation (compared to total available) for the different PEs excluding auxiliary components, and clock frequency on the Alveo U280 device

Variant	BRAM, No Fuzzer HW	BRAM	DRAM via AXI4
LUT	4521 (0.35%)	6978 (0.54%)	7323 (0.56%)
Register	3467 (0.13%)	5745 (0.22%)	5877 (0.23%)
DSP	4 (0.04%)	10 (0.11%)	10 (0.11%)
BRAM	128 KiB (1.59%)	136 KiB (1.69%)	56 KiB (0.69%)
$f_{\max}$	400 MHz	390 MHz	360 MHz



(a) Comparison of the fuzzing execution rate with the *minimal* and *ArduinoJson* test programs between this work with a 64- (minimal) / 2048-entry coverage map (ArduinoJson, 1h), CVA5 without fuzzer hardware on a pregenerated testcase corpus, AFL++ with QEMU (RISC-V) and AFL++ native.

(b) Comparison of the absolute edge coverage on the native RISC-V binary achieved after one hour of fuzzing the *ArduinoJson* test program, between this work with a 2048-entry map, AFL++ with QEMU and AFL++ native. *Without libraries* removes CF related to libc functions. *: For illustration only, as AFL++ guidance ignores libraries.

(c) Coverage over time without libraries, corresponding to Figure 3b. All lines start with 19 edges.

Fig. 3: Results: Fuzzer job execution rate and number of detected CF edges

We examine two target programs for this work. The first program, *minimal*, consists solely of the fuzzer-specific entry point calling an empty function; this serves as a peak execution rate benchmark. The second program, which we refer to as *ArduinoJson*, is a typical part of IoT applications that deserializes the input data using the *ArduinoJson* C++ library [1] into a dynamic memory buffer. It also makes limited use of floating-point arithmetic for number parsing.

The programs are compiled in *-Os* mode, 1) for our RISC-V hardware fuzzer environment (*gcc*), 2) for a RISC-V Linux environment for AFL++ QEMU evaluation (*gcc*), and 3) with AFL++ native x86 instrumentation (*afl-clang-fast*). Note that the native x86 variant employs an AFL++ *persistent mode* harness to significantly reduce the number of *fork()* system calls, whereas the RISC-V Linux build uses a *forkserver* harness for the same purpose, since the faster AFL++ QEMU *persistent mode* is not supported for RISC-V targets [3].

Comparing the raw program execution rates in Figure 3a, AFL++ in native persistent mode is significantly faster than both AFL++ QEMU forkserver mode and our work. But this approach carries the accuracy penalties discussed in Section 1. When performing the more accurate fuzzing on the actual RISC-V code, our work is 4.6*x* / 31% faster compared to AFL++ QEMU when fuzzing the *minimal* and *ArduinoJson* programs, respectively. As we expected, runtime is increased when fuzzing the more complex *ArduinoJson* target. This is also

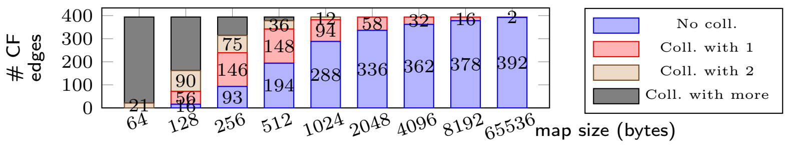


Fig. 4: Number of collisions for different fuzzer result map sizes, obtained via our hashing algorithm (Section 4.3) on the *ArduinoJson* corpus after 1h of fuzzing.

because we chose larger coverage maps (2 KiB) that have to be transferred back to the host via PCIe, which impacts our speedup relative to AFL++ QEMU. The third result in ArduinoJson without fuzzer hardware shows the impact.

Figures 3b and 3c show the absolute edge coverage attained after one hour of fuzzing. Notably, most edges are found within the first minute, indicating that the last edges are harder to find; the total number of reachable edges cannot be predicted. Since the FRA in our work includes all observed CFs, fuzzer guidance also optimizes coverage over the library code in contrast to AFL++. Overall, our BRAM fuzzer variant achieves the highest coverage both including and excluding library address ranges for *libc* and software floating point. The DRAM variant has the lowest result, which we attribute to higher execution rates from slower memory connectivity, due to which the fuzzer prefers inputs with simpler CF.

5.3 Hash Collisions

Since the coverage edges are hashed and then reduced to lower bit widths, collisions appear, such that two or more edges are assigned the same index in the map. Figure 4 quantifies this for the *ArduinoJson* example. If the overall coverage maps for inputs that reach certain colliding edges are similar, the fuzzer may store only one of the inputs in its corpus, leading to suboptimal coverage. The probability of collisions drops with higher map sizes, which on the other hand increase data transfer and processing overhead.

6 Conclusion and Future Work

In this work, we developed multiple enhancements for the CVA5 RISC-V core to make it more suitable for hardware accelerated fuzzing. The resulting solution is competitive when compared to an existing SoA fuzzer with software emulation, even when the latter employs a very fast desktop CPU as base for emulation.

For future work, we intend to scale the number of fuzzer units on the FPGA and optimize the DRAM fuzzer unit in order to compete with multithreaded fuzzing in software. Since the transfers of large coverage maps over PCIe currently limit performance, we will also explore approaches to reduce their required sizes.

Acknowledgements The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the projects "Open6GHub" and "MANNHEIM-FlexKI" (grant numbers: 16KISK014, BMBF F&E 01IS22086A-L). Part of this research work has been funded by the German Federal Ministry of Education and Research and the Hessian Ministry of Higher Education, Research, Science and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

References

1. ArduinoJson: Efficient JSON serialization for embedded C++ (2022), https://arduinojson.org/, [as of 21-October-2022]
2. clickguard - the most recent botnet attacks: The 2022 edition (2022), https://www.clickguard.com/blog/recent-botnet-attacks-2022/ [as of 21-October-2022]
3. High-performance binary-only instrumentation for afl-fuzz (2022), https://github.com/AFLplusplus/AFLplusplus/blob/stable/qemu_mode/README.md [as of 2022-10-21]
4. QEMU - A generic and open source machine emulator and virtualizer (2022), https://www.qemu.org [as of 2022-01-05]
5. Arm Limited: AMBA AXI and ACE Protocol Specification Version E, part 1. AMBA AXI3 and AXI4 Protocol Specification
6. Fioraldi, A., Maier, D.: The LibAFL fuzzing library, https://aflplus.plus/libafl-book/libafl.html [as of 2022-07-17]
7. Fioraldi, A., Maier, D.: LibAFL StdFuzzer, https://github.com/AFLplusplus/StdFuzzer [as of 2022-07-17, commit c9f1bed]
8. Fioraldi, A., Maier, D., Eißfeldt, H., Heuse, M.: AFL++ : Combining incremental steps of fuzzing research. In: 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association (Aug 2020), https://www.usenix.org/conference/woot20/presentation/fioraldi
9. Heinz, C., Hofmann, J., Korinth, J., Sommer, L., Weber, L., Koch, A.: The TaPaSCo Open-Source Toolflow. Journal of Signal Processing Systems **93**, 545–563 (2021). https://doi.org/https://doi.org/10.1007/s11265-021-01640-8
10. Heinz, C., Lavan, Y., Hofmann, J., Koch, A.: A catalog and in-hardware evaluation of open-source drop-in compatible risc-v softcore processors. In: 2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig). pp. 1–8 (2019). https://doi.org/10.1109/ReConFig48160.2019.8994796
11. Liang, H., Pei, X., Jia, X., Shen, W., Zhang, J.: Fuzzing: State of the art. IEEE Transactions on Reliability **67**(3), 1199–1218 (2018). https://doi.org/10.1109/TR.2018.2834476
12. Matthews, E., Shannon, L.: TAIGA: A new RISC-V soft-processor framework enabling high performance CPU architectural features. In: 2017 27th International Conference on Field Programmable Logic and Applications (FPL). pp. 1–4 (2017). https://doi.org/10.23919/FPL.2017.8056766
13. OpenHW Group: CORE-V CVA5 (repository), https://github.com/openhwgroup/cva5 [as of 2022-07-21, commit 3239e20]
14. The LLVM Project: libFuzzer – a library for coverage-guided fuzz testing, https://llvm.org/docs/LibFuzzer.html [as of 2022-07-17]
15. Wellons, C.: Hash function prospector, https://github.com/skeeto/hash-prospector [as of 2022-10-21, commit 2051e59]
16. Zalewski, M.: american fuzzy lop, https://lcamtuf.coredump.cx/afl/ [as of 2022-07-25]