I. INTRODUCTION

Verification of embedded software (SW) binaries is very important. Mainly, simulation-based methods are employed that execute (randomly) generated test-cases on Virtual Prototypes (VPs). However, to enable a comprehensive VP-based verification, sophisticated test-case generation techniques need to be integrated. Our demonstrator (called VP-CGF) combines state-of-the-art fuzzing techniques with SystemC-based VPs to enable a fast and accurate verification of embedded SW binaries. The fuzzing process is guided by the coverage of the embedded SW as well as the SystemC-based peripherals of the VP. The effectiveness of our approach is demonstrated by our experiments, using RISC-V SW binaries as an example.

II. RISC-V VP OVERVIEW

As a foundation for VP-CGF we use our open-source RISC-V VP (MIT license, visit [GitHub link](http://www.systemc-verification.org/riscv-vp) for the GitHub link as well as most recent updates and related information). We provide a 32 and 64 bit RISC-V core supporting the RV32IMACF+SUN and RV64GC+SUN instruction set, respectively, with different privilege levels, the RISC-V CLINT and PLIC interrupt controllers and an essential set of peripherals. Furthermore, we support simulation of (mixed 32 and 64 bit) multi-core platforms, provide SW debug (using Eclipse IDE) and coverage measurement capabilities and support the Linux, FreeRTOS and Zephyr operating systems. The VP is designed as configurable and extensible platform. As an example we provide a configuration matching the HiFivel board from SiFive.

III. FUZZING EMBEDDED SOFTWARE BINARYSN

Fig. 1 shows an overview of our approach VP-CGF for verification of embedded SW binaries. Our approach leverages two components, (LLVM) libFuzzer (shown on the left side of Fig. 1) and a SystemC-based VP (shown on the right side of Fig. 1) that interoperate in a loop. Essentially, the fuzzer provides new inputs and the VP executes these inputs and returns coverage information to the fuzzer. VP and fuzzer are separate processes and communicate through sockets and files. Please note, we use the fork system call to spawn a new VP instance for each fuzzer input, which has been very important to obtain good performance results. We observed speed-ups of more than 20x compared to starting a new VP instance (i.e. new process) for each fuzzer input. The main reason for this performance impact is the SystemC simulation engine which requires significant time for an initial startup.

We obtain branch coverage information for the SystemC peripherals by compiling them with Clang using the `--fsanitize-coverage=trace-pc-guard` option. With this option Clang instruments the peripherals to emit coverage information for branch instructions at runtime by calling special interface functions, that we intercept in the VP. We obtain (branch) coverage information for the embedded SW binary by observing the executed instructions in the ISS (Instruction Set Simulator) of the VP. In particular, we have installed a hook in the ISS that intercepts and processes every executed branch instruction. We collect all observed branch edges in the Clang instrumentation format to keep it compatible with libFuzzer.

IV. EXPERIMENTAL RESULTS

First, we used VP-CGF to test two bare-metal embedded RISC-V applications. VP-CGF was much more efficient (at least factor 20x faster in finding the bug) than pure random test generation. We also obtained very high coverage values for both the SW (98%) and the SystemC-based peripherals (94%).

Second, we used VP-CGF to test the Zephyr OS network IP-stack to demonstrate it’s applicable to analyze large real-world embedded SW. We have obtained 82.44% line coverage in the IPv4 and IPv6 processing stack of Zephyr by running VP-CGF for one hour. The resulting testset contains 575 testcases. VP-CGF executed 276,395 testcases in total with an average and maximum of 76 and 165 testcases per second, respectively. The analyzed RISC-V binary has 46,105 lines of ASM code.

REFERENCES