Verification of Embedded Binaries using Coverage-guided Fuzzing with SystemC-based Virtual Prototypes

Vladimir Herdt
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
Vladimir.Herdt@dfki.de

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

Jonas Wloka
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
Jonas.Wloka@dfki.de

Tim Güneysu
Chair for Security Engineering, Ruhr-University Bochum
Cyber-Physical Systems, DFKI GmbH
Bremen, Germany
tim.gueneysu@rub.de

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

ABSTRACT

Extensive verification of embedded SW is very important to avoid errors and security vulnerabilities. Therefore, mainly simulation-based methods are employed that leverage *Virtual Prototypes* (VPs) for SW execution early in the design flow. VPs are essentially abstract models of the entire HW platform including peripherals. They are predominantly created in SystemC. However, a comprehensive simulation-based verification requires integration of sophisticated test generation techniques.

In this paper we propose to leverage state-of-the-art *Coverage-guided Fuzzing* (CGF) methods in combination with SystemC-based VPs for verification of embedded SW binaries. Using VPs, our approach allows a fast and accurate binary-level SW analysis and enables checking of complex HW/SW interactions. To guide the fuzzing process we combine the coverage from the embedded SW with the coverage of the SystemC-based peripherals. Our experiments, using RISC-V embedded SW binaries as examples, demonstrate the effectiveness of our approach.

CCS CONCEPTS

• **General and reference → Verification**; • **Computer systems organization →** *Embedded software.*

KEYWORDS

Verification; Fuzzing; Virtual Prototype; RISC-V; SystemC

ACM Reference Format:
Vladimir Herdt, Daniel Große, Jonas Wloka, Tim Güneysu, and Rolf Drechsler. 2020. Verification of Embedded Binaries using Coverage-guided Fuzzing with SystemC-based Virtual Prototypes. In *Proceedings of the Great Lakes Symposium on VLSI 2020 (GLSVLSI '20), September 7–9, 2020, Virtual Event, China.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3386263.3406899

1 INTRODUCTION

Embedded systems are prevalent nowadays in many different application areas. They integrate several peripherals alongside the CPU core and extensively rely on embedded *Software* (SW) for configuration and to implement complex functionality. Verification of the embedded SW is very important to avoid errors and security vulnerabilities.

Therefore, mainly simulation-based methods are employed that leverage *Virtual Prototypes* (VPs) for SW execution early in the design flow [11]. VPs are essentially abstract models of the entire *Hardware* (HW) platform and predominantly created in SystemC TLM (*Transaction-Level Modeling*) [1]. VPs offer much better simulation performance compared to an RTL (*Register-Transfer Level*) simulation and at the same time offer more accuracy than high speed *Instruction Set Simulators* (ISSs), like *QEMU*, and are designed from the ground up to represent the whole HW platform not just the CPU core. Thus, SystemC-based VPs provide an industry-proven approach for analysis of complex HW/SW interactions (and other system-level use cases such as design space exploration or power/timing/performance validation). However, a comprehensive simulation-based verification requires integration of sophisticated test generation techniques.

Recently, in the SW domain the automated SW testing technique *fuzzing* [24] has been shown very effective in testcase generation and became a standard in the SW development flow [5]. These modern fuzzing techniques typically employ the so called *mutation* based technique. It mutates randomly created data and is guided by code coverage. Notable representatives in this *Coverage-guided Fuzzing* (CGF) category are the LLVM-based *libFuzzer* [4] and *AFL* [3], which both have been shown very effective and revealed various new bugs that can lead to errors and security vulnerabilities [3, 4]. However, using CGF for checking embedded SW

binaries is challenging, because it requires to analyze architecture specific SW binaries that extensively interact with HW peripherals.
Contribution: In this paper we propose to leverage state-of-the-art CGF methods in combination with SystemC-based VPs for verification of embedded SW binaries. We call this combination *VP-CGF*. VP-CGF brings together the benefits of both worlds: CGF is a sophisticated and very effective method for testcase generation and VPs allow for a fast and accurate binary-level SW analysis and enable checking of complex HW/SW interactions during testcase execution. To guide the fuzzing process we combine the coverage from the embedded SW additionally with the coverage of the SystemC-based peripherals. Both coverage information are tracked in the VP alongside the testcase execution. Our experiments demonstrate the effectiveness of our approach in analyzing real-world RISC-V embedded SW binaries.

2 RELATED WORK

Several formal verification methods, e.g. [8, 10, 14, 17, 18], primarily based on symbolic/concolic execution techniques, have been proposed for verification of embedded SW binaries. However, formal methods are still highly susceptible to state explosion.

Different random testing / fuzzing approaches targeting embedded systems have been reported: [21, 22] use random fuzzing of automobile ECUs with CAN packets. [9, 19] use generational (model-based) and evolutionary (based on genetic algorithms) fuzzing techniques to analyze smart cards. [26, 27] analyze the GSM implementation in mobile phones by sending random SMS messages. However, neither of these approaches leverages state-of-the-art CGF or VPs. An overview on challenges in fuzzing embedded systems is provided in [25].

In [20, 28] methods are presented to integrate real HW peripherals with the SW simulation to enable accurate analysis of embedded SW even when no peripheral models are available. We consider these approaches to be complementary to our approach.

Recently, QEMU has been combined with AFL and even been used for fuzzing the Linux kernel [6, 7]. This combination (QEMU-AFL) enables CGF-based analysis of embedded binaries by emulating them in QEMU and propagating the observed SW coverage at runtime from QEMU back to AFL. Another similar project combines AFL with the Unicorn CPU emulator [2]. However, these approaches are primarily focused on emulating the CPU core and either do not consider peripherals at all (Unicorn-AFL) or only to a limited extent (QEMU-AFL), without using the peripheral coverage in the fuzzing process. In addition, QEMU does not support SystemC-based VPs, which is an industry-proven modeling standard (IEEE-1666) and hence deserves dedicated verification solutions.

Finally, fuzzing has also been leveraged for generation of processor-level stimuli to verify (instruction set) simulators / CPU cores, e.g. [12, 15, 23]. However, this is a completely different focus to verifying embedded SW binaries.

3 OUR APPROACH: VP-CGF

In this section, we present our proposed approach *VP-CGF* for verification of embedded SW binaries using CGF with SystemC-based VPs. We start with an overview (Section 3.1). Then, in Section 3.2 we describe how we track code coverage information for the SystemC-based peripherals (in the VP) and the embedded SW binary (that is tested). Finally (Section 3.3), we present an example embedded application that is verified with VP-CGF.

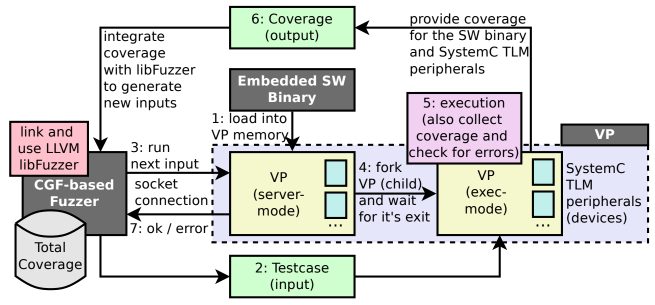
6: Coverage (output)
integrate coverage with libFuzzer to generate new inputs
provide coverage for the SW binary and SystemC TLM peripherals
Embedded SW Binary
1: load into VP memory
5: execution (also collect coverage and check for errors)
VP
link and use LLVM libFuzzer
3: run next input
CGF-based Fuzzer
socket connection
7: ok / error
VP (server-mode)
4: fork VP (child) and wait for it's exit
VP (exec-mode)
SystemC TLM peripherals (devices)
Total Coverage
2: Testcase (input)

Figure 1: Overview on VP-CGF: CGF with SystemC-based VPs for verification of embedded SW binaries

3.1 Overview

Fig. 1 shows an overview of our approach VP-CGF for verification of embedded SW binaries. Our approach leverages two components, a CGF-based fuzzer (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. In the following we present more details on our approach.

Starting point is an embedded SW binary that represents the embedded SW application. In the first step the VP is started in *server-mode* and the embedded SW binary is loaded into the VPs memory. When started in *server-mode*, the VP will fully initiate (i.e. setup the SystemC simulation engine and instantiate all components) until the VP is ready for execution of the embedded SW binary. At this point the VP will stop and wait for commands from the fuzzer.

Next, we start the fuzzer that connects with the VP through a socket connection and starts the *fuzz-loop* (repeating Steps 2 to 7 in Fig. 1), which works as follows: The fuzzer begins by writing a new testcase into a (binary) file (Step 2, bottom of Fig. 1) and notifies the *server-mode* VP by sending a run command (Step 3). The VP will then use the *fork* system call to create a new identical VP child process in *exec-mode* (Step 4) to execute the embedded SW binary with the fuzzer-provided input (Step 5, right side of Fig. 1). Alongside the execution, the *exec-mode* VP collects coverage information for the embedded SW binary as well as the SystemC peripherals, and writes them into a coverage output file (Step 6, top of Fig. 1). The *server-mode* VP waits until the *exec-mode* VP finishes (i.e. the forked process exits) and notifies the fuzzer that the execution has finished and whether an error was detected during this execution (Step 7), by checking the return code of the *exec-mode* VP process. The fuzzer reads and integrates the coverage output file. In case new coverage has been obtained, the fuzzer collects the testcase. The *fuzz-loop* continues (repeating Steps 2 to 7) until an error is detected or a user-defined timeout is reached. Based on the collected testcases a coverage report can be generated by re-running the testcases on the VP (in *exec-mode*).

```
 1 // memory mapped register/inputs
 2 volatile char*
      SENSOR_INPUT_ADDR=(char*)0x50000000;
 3 volatile uint32_t*
      SENSOR_SCALER_ADDR=(uint32_t*)0x50000080;
 4 volatile uint32_t*
      SENSOR_FILTER_ADDR=(uint32_t*)0x50000084;
 5 volatile char*
      FUZZER_INPUT_ADDR=(char*)0xC0000000;
 6 // consume fuzzer-provided input
 7 uint8_t fuzz_u8() {//one byte
 8   return *FUZZER_INPUT_ADDR;
 9 }
10 uint32_t fuzz_u32() {//four bytes
11  return fuzz_u8() | fuzz_u8()<<8 | \
12    fuzz_u8()<<16 | fuzz_u8()<<24;
13 }
14 // access/process sensor data
15 _Bool has_sensor_data = 0;
16 void sensor_irq_handler() {
17   has_sensor_data = 1;
18 }
19 void process_sensor_data() {
20   // wait for sensor interrupt
21   while (!has_sensor_data)
22     asm volatile ("WFI"); // Wait
         For (any) Interrupt
23   has_sensor_data = 0;
24   // read and validate sensor data
25   unsigned sum = 0;
26   for (int i=0; i<8; ++i) {
27    uint8_t n=*(SENSOR_INPUT_ADDR+i);
28    sum += n;
29   }
30   // validate sensor data
31   assert (sum < 8*250);
32 }
33 void main() {
34  register_interrupt_handler(IRQ_NUMBER,
       sensor_irq_handler);
35  // config. sensor using fuzzer inp.
36  *SENSOR_SCALER_ADDR=fuzz_u32();
37  *SENSOR_FILTER_ADDR=fuzz_u32();
38  process_sensor_data();
39 }
```

Figure 2: Example embedded SW accessing a sensor peripheral

Please note, using the *fork* system call to spawn a new VP instance for each fuzzer input 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.

3.2 Coverage Collection in the VP

During execution the VP collects coverage from the embedded SW binary and from the SystemC peripherals. Currently, we focus on branch coverage information.

3.2.1 Peripheral Coverage. 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. We provide these interface functions in the VP to collect the coverage information and forward that information, through the coverage output file, to the fuzzer accordingly. The fuzzer integrates the coverage by forwarding it to libFuzzer, since libFuzzer also understands Clang instrumentation and hence provides the same interface functions. Please note, we only selectively instrument the SystemC peripherals with Clang and not the whole VP to avoid the (potentially) significant runtime overhead of instrumenting (in particular) the ISS of the VP and also avoid the communication overhead in transferring this additional coverage information (which is irrelevant for testing the embedded SW) to the fuzzer.

3.2.2 Embedded SW Coverage. We obtain (branch) coverage information for the embedded SW binary by observing the executed instructions in the ISS of the VP. In particular, we have installed a hook in the ISS that intercepts every instruction execution and checks if it is a branch instruction. In case of a branch instruction we collect the address of the branch instruction itself (pre-pc) and the address of the instruction after the branch (post-pc, which depends on whether the branch was taken or not). The pair (pre-pc, post-pc) represents a control flow edge. We collect all observed edges in the Clang instrumentation format to keep it compatible with libFuzzer. This also allows us to store all coverage information (for the embedded SW binary and SystemC peripherals) in a single unified list. Essentially, Clang assigns each branch instruction a unique *guard* index. We mimic this behavior by mapping each different edge to an ascending index starting after the last *guard* index used by Clang for instrumenting peripheral branches (Clang provides an interface function to obtain the number of instrumented branches).

3.3 Concept: Embedded SW Fuzzing with VPs

To further illustrate the fuzzer integration concept, we present an example embedded application that consists of an embedded SW accessing a sensor. The sensor (SystemC TLM) model has two 32 bit configuration registers *scaler* and *filter* and a data frame of 8 bytes that represents the current sensor measurement and is periodically updated in a SystemC process with new data. The scaler register setting defines the speed at which the sensor data frame is updated. The filter register setting selects the sensor internal post-processing that is applied on the data frame. Registers and data frame are accessed by the SW through memory mapped I/O.

Fig. 2 shows the example embedded SW that accesses the sensor. The SW starts by installing an interrupt handler (Line 34) and configures the sensor *filter* (Line 37) and *scaler* setting (Line 36) by writing to designated memory mapped registers. Then, the SW waits for a sensor interrupt (Line 21-22), which denotes that new sensor data is available. Finally, the SW fetches the sensor data (Line 25-29) and validates it (Line 31).

Both sensor configuration registers are initialized with fuzzer-provided input (via the *fuzz_u32* function defined in Line 10). The SW can access the fuzzer-provided input through a special peripheral (denoted *FP-peripheral*) that maps the fuzzer input into the memory. Reading a byte from this memory location (Line 8) will consume and return a byte from the fuzzer-provided input. The sensor (and all other SystemC peripherals as well) also access and consume bytes from the *FP-peripheral*. The (SystemC) simulation is stopped by the (SystemC) *FP-peripheral* in case all fuzzer-provided input has been consumed (and the next byte is requested).

The assertion in the SW in Line 31, which checks that the sensor data stays within a valid maximum range, can be violated for specific sensor inputs. Please note, it is possible to maximize the SW code coverage but still miss the assertion violation because the occurence of the violation depends on the filter setting of the sensor (i.e. the value that the SW writes in Line 37 to configure the sensor post-processing behavior). Thus, it is important to also consider code coverage from peripherals during testing of embedded SW to detect bugs that depend on HW/SW interactions.

Table 1: Experiment results on applying our VP-CGF approach for testing embedded applications

Metric	Application	
	1) data-transfer	2) fan-control
1: LoC SW C	142	116
2: LoC SW RISC-V ASM	435	371
3: LoC Peripherals SystemC	331	262
4: VP-CGF Time to Bug (sec.)	180	20
5: VP-CGF Exec Total	22189	6400
6: VP-CGF Exec/Sec Average	123	320
7: VP-CGF Exec/Sec Maximum	651	640
8: VP-CGF Number Final Tests	98	46
9: VP-CGF Coverage SW	98.18%	97.5%
10: VP-CGF Coverage Peripherals	94.79%	96.35%
11: VP-Random Time to Bug (sec.)	T.O. (>3600)	3212

4 EXPERIMENTS

We have implemented our proposed VP-CGF approach for verification of embedded binaries using LLVM libFuzzer and the open-source RISC-V based VP [13, 16] as a case-study. We evaluate VP-CGF in two steps: First, we present results on testing two bare-metal embedded RISC-V applications with VP-CGF. Then, we show results on applying VP-CGF to test the Zephyr OS network IP-stack. All experiments are evaluated on a Linux machine with an Intel i5-7200U processor.

4.1 Testing Embedded Applications

In the following, we start with an overview of the results (Section 4.1.1) and then present more details for each of the two embedded applications (Section 4.1.2 and Section 4.1.3) and end with a discussion of the results (Section 4.1.4).

4.1.1 Results Overview. Table 1 gives the results. It shows 11 different metrics (one in each row) for both embedded applications. In particular, Table 1 shows the LoC (*Lines of Code*) of the embedded SW in C (Row 1) and RISC-V ASM (Row 2) as well as the LoC of the relevant SystemC peripherals (Row 3) in the VP. Next, Table 1 shows the time in second until VP-CGF finds the bug (Row 4), how many testcases have been executed in total (Row 5) as well as the average (Row 6) and maximum (Row 7) number of testcase executions per second. Row 8 shows the final number of collected testcases (i.e. testcases that increase the coverage, until the bug is found) and Rows 9 and 10 show the line coverage of the SW and SystemC peripherals that we obtain by re-running the testcases of Row 8. Finally, Row 11 shows the time in seconds to find the bug by generating testcases purely randomly (without any coverage feedback) as comparison to VP-CGF.

4.1.2 Application 1: Data-Transfer. The first application that we consider reads a data stream from an input device, processes the data stream and writes the processed stream back to an output device. We use an UART in this application to act as input (RX) and output (TX) device, respectively. The data is stored in an internal ring buffer during the copy process in the SW. The application starts initializing the system, i.e. setting up peripherals by writing to their configuration registers and registering interrupt handlers, and then enters a WFI (*Wait For Interrupt*) loop. On each UART interrupt (i.e. the UART has new incoming data to receive or is able to transmit new outgoing data) the application logic is triggered.

The application logic consists of two parts: 1) First, it transfers data in a loop from the input UART to the ring buffer. The loop stops when the UART receive queue is empty or the ring buffer has not sufficient free space to store new data. Two operation modes are available: *raw* and *escape*. In *raw* mode the data is simply passed on unprocessed. In *escape* mode the special character `0x7f` is escaped. Whenever the `0xff` byte is observed in the input stream the operation mode is changed from *raw* to *escape* mode and vice versa. 2) In the second part of the application logic, the ring buffer content is transferred to the output UART in a similar way. It stops when the transmit queue of the output UART is full or the ring buffer is empty.

Test Setup. On the SW side we initialize the UART RX and TX interrupt *watermark-level* with fuzzer-provided input. The *watermark-level* configures when interrupts are triggered by the UART depending on how many elements are currently stored in the RX and TX queue, respectively. We added coverage points (on the VP side by means of artificial branches, when the configuration is received in the SystemC peripheral) to distinguish between the different possible *watermark-levels* (0..7 for RX and 0..7 for TX). On the VP side we fill the RX queue of the UART with fuzzer-provided input and check that the data arrives again in the same order in the TX queue of the UART and is correctly escaped (after being processed by the SW). Therefore, we use SystemC threads with periodic delays (RX-delay and TX-delay, respectively) to 1) add new (fuzzer-provided) data to the RX queue of the input UART (RX-thread), and 2) consume data from the TX queue of the output UART (TX-thread). Both cases can trigger an interrupt, depending on the interrupt *watermark-level* configuration, that will in turn notify the SW application.

Please note, that the SW application is non-terminating, because it waits for new interrupts indefinitely. Therefore, we simply stop the SW execution in the VP when the whole fuzzer input has been consumed. We use fuzzer-provided input to configure the RX- and TX-delays, respectively. This allows to check for errors related to the execution order of the RX-/TX-threads. We added coverage points to distinguish the three cases: RX-delay < TX-delay, RX-delay = TX-delay, RX-delay > TX-delay. They describe settings where data can be faster/equally fast/slower received than transmitted.

4.1.3 Application 2: Fan-Control. The second application that we consider controls the system fan speed based on temperature values obtained from a temperature sensor. Therefore, the sensor triggers periodic interrupts after each time interval to notify the application that a new sensor data frame is available. Each data frame contains the temperature values measured during the last time frame. The sensor provides a filter register that controls the range of measured temperature values. On each sensor interrupt, the application first copies the sensor data frame into an internal buffer and then computes the average temperature value for the last time frame (i.e. for the elements of the buffer). Based on the current and the last observed average temperature value the fan speed setting is controlled (level=0 off to level=5 high speed). The fan speed is immediately set to high speed in case of a high temperature value but requires low temperature values for two time frames to reduce its speed.

Technically, the mapping from temperature value to fan speed is implemented by normalizing the temperature value to an array index and accessing a pre-configured buffer.

Test Setup. On the SW side we initialize the sensor filter register value with fuzzer-provided input. On the VP side we fill the sensor data frame with fuzzer-provided input. Similar to the first application, this application is also non-terminating, and hence we also stop the execution once the whole fuzzer input has been consumed. We added assertions in the fan control peripheral to check that a high fan speed is selected in case a high temperature value is measured. In addition, we have added assertions to check for buffer overflows in the SW, i.e by adding bounds checks before every array access.

4.1.4 Results Discussion. We found a bug with VP-CGF in 180 seconds in the first application (Section 4.1.2). The bug results in data being overwritten in the ring buffer before it is written back to the (TX) UART. The reason for the bug is an incorrect length calculation in the ring buffer. In particular, the free space calculation returns one more element than available if the buffer *read* pointer is before the buffer *write* pointer (*read* and *write* denote the next element to be read/written, respectively. They are updated after each access accordingly and wrap around the ring buffer when reaching the end). The bug is triggered if *read* points to the first element and *write* to the second last element of the ring buffer while the operation is set to *escape* mode and at the same time the data byte `0x7f` is received from the UART. This combination will cause the SW to overflow the ring buffer.

In the second application (Section 4.1.3) we found a buffer overflow with VP-CGF in 20 seconds. The reason for this error is a bug in the temperature normalization procedure that restricts all temperature values above a certain threshold to stay below that threshold. However, the case where the (average) temperature value equals the threshold is missed and hence this particular temperature value is not below the threshold, which in turn results in a computation of an out-of-bounds buffer index. To trigger the buffer overflow the temperature sensor (hence the fuzzer in this setting) has to provide a data frame such that the average temperature value of this frame is equal to the threshold. Please note, that the data frame also depends on the filter register value in the sensor which is also provided by the fuzzer.

It can be observed that VP-CGF is much more efficient than pure random test generation on both applications (factor 20x to 160x faster in finding the bug). We also obtained very high coverage values for both the SW (97% to 98%) and the SystemC-based peripherals (94% to 96%). Careful examination of the coverage results reveals that the remaining peripheral coverage is indeed unreachable in combination with this SW. It corresponds to access of additional configuration registers and interrupt priorities, which are not used by this SW. The remaining missing SW coverage (2% to 3%) can be closed by re-running VP-CGF on the fixed SW.

4.2 Testing the Zephyr IP Stack

In the second experiment we apply our approach VP-CGF to test the network IP stack of the Zephyr OS in combination with the RISC-V port of the Zephyr kernel (Zephyr version 1.14.99). This experiment demonstrates that VP-CGF is applicable to analyze large real-world embedded SW. We test for generic execution errors (e.g. SW assertion violation and access of unmapped memory) and buffer overflows in particular. We have integrated a custom mutator into the fuzzing process to further guide it. We start by explaining our test setup (Section 4.2.1), then present our custom mutator (Section 4.2.2) and method for buffer overflow detection at runtime (Section 4.2.3). Finally, we present our results (Section 4.2.4).

4.2.1 Test Setup. IP packets are processed by the platform independent IP-thread in Zephyr (threads and synchronization mechanisms are provided by the Zephyr kernel). Packets are delivered to the IP-thread through a shared queue by a lower-level network driver.

We use the SLIP (*Serial Line Internet Protocol*) network driver of Zephyr, which allows to send and receive IP packets (including higher-level protocols like UDP and TCP) by (de-)serializing the data through a UART device (which is provided by the VP).

In this experiment, we treat the whole fuzzer-provided input as a single IP packet. We wrap the IP packet as SLIP packet and pass it to the UART (in the VP). This in turn triggers a UART interrupt which notifies the serial driver and finally the SLIP driver. The SLIP driver extracts the IP packet and delivers it to the IP-thread of Zephyr.

In the SW main function we create and bind an UDP and TCP socket using the IPv4 as well as IPv6 protocol. This ensures that the Zephyr IP-thread does not drop IP packets prematurely because no matching socket is bound. Please note, that the Zephyr IP-thread is non-terminating because it waits for new IP packets indefinitely. Therefore, we have added a switch in the IP-thread to stop execution after one IP packet has been processed.

4.2.2 Custom IP Packet Mutation. LLVM libFuzzer provides a designated interface function to easily integrate custom mutators into the fuzzing process. We added such a custom mutator and configured it to be called with the same probability as the existing libFuzzer mutators. Our custom mutator simply overwrites the fuzzer-provided data with a small pre-defined UDP packet. Our mutator starts overwriting the data from the beginning and stops when either the whole packet has written or the end of the fuzzer-provided data is reached, hence only injecting a prefix of the packet in this case. The injected UDP packet is completely valid and will fully propagate through the IP stack to the UDP socket and reach the Zephyr application when send independently. Our mutator essentially introduces valid IP and UDP header options into the fuzzing process to enable a more comprehensive evaluation.

4.2.3 Heap Buffer Overflow Detection. Zephyr provides several different functions to manage memory. These include the functions *k_malloc* and *k_free*, which correspond to the well-known *malloc* and *free* functions of the C library, as well as specialized functions optimized for allocation of fixed size memory blocks. We have instrumented the memory management functions of Zephyr, by wrapping them, to track allocated memory blocks and their bounds.

As an example, Fig. 3 shows wrappers for *k_malloc* and *k_free*. The *k_malloc* wrapper calls the real *k_malloc* function of Zephyr to allocate a larger than requested block (Line 3-5), by adding extra space (*ALLOC_MARGIN*) before and after the requested block. The extra space is registered in the VP (Line 11), alongside the allocated block, and treated as protected memory. The VP monitors all

```
1  #define MARGIN 128 // in bytes
2  void *k_malloc(size_t size) {
3    size_t alloc_size=size+2*MARGIN;
4    // call real k_malloc of Zephyr
5    void *p=real_k_malloc(alloc_size);
6    if (p == NULL)
7      return NULL;
8    // obtain start of usable memory
9    void *block=((uint8_t*)p)+MARGIN;
10   // notify VP about allocation
11   VP_k_malloc(block, size, MARGIN);
12   return block;
13 }
14 void k_free(void *block) {
15   // notify the VP about the free
16   VP_k_free( block );
17   // obtain the real block start
18   void *p=((uint8_t*)block)-MARGIN;
19   // call real k_free of Zephyr
20   real_k_free( p );
21 }
```

Figure 3: Wrapper functions to keep track of dynamically allocated blocks and check for buffer overflows at runtime

memory access operations (read and write) during execution and triggers an error in case the address falls in a protected memory region. The *k_free* wrapper notifies the VP about the *k_free* call (Line 16), to unregister the protected memory margins and check for double free as well as non-allocated block errors, and finally calls the real *k_free* function of Zephyr (Line 20).

4.2.4 Results. 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.

To also evaluate the effectiveness of VP-CGF in detecting (buffer overflow) related bugs, we manually added a bug into the Zephyr IP stack. The bug corresponds to a typical overflow caused by incomplete buffer length checks which may result in severe security vulnerabilities. Similar bugs had been present in recent versions of FreeRTOS. In order to trigger that bug, the fuzzer has to generate an almost valid IPv4 header (otherwise the packet will be dropped in the IP stack before having a chance to trigger the bug) but with a specific malformed header length. VP-CGF detected this bug in 229 seconds and executed a total of 20,535 testcases.

5 CONCLUSION AND FUTURE WORK

In this paper we proposed VP-CGF, an approach that leverages state-of-the-art CGF methods in combination with SystemC-based VPs for verification of embedded SW binaries. In addition to the coverage of the embedded SW, we also integrate the coverage of the (SystemC-based) peripherals to further guide the fuzzing process. Our experiments showed very promising results and demonstrate the effectiveness of VP-CGF in analyzing real-world embedded SW binaries. To further improve VP-CGF, for future work we plan to:

- Support stronger coverage metrics. In particular, add support for Clangs data flow related coverage sanitizers and investigate cross-coverage metrics between SW and peripherals.
- Evaluate the impact of parallelization on the fuzzing process. In general our current architecture allows to run multiple VP-CGF sessions (each session has a separate VP and fuzzer instance) in parallel, since libFuzzer supports (safe multi-instance) coverage synchronization using the OS filesystem.
- Integrate further dynamic error checking mechanisms into the VP to detect additional error classes at runtime. Error detection is complementary to testcase generation.

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 SATiSFy under contract no. 16KIS0821K and within the project Scale4Edge under contract no. 16ME0127.

REFERENCES

[1] 2011. *IEEE Standard SystemC Language Reference Manual.* IEEE Std. 1666.
[2] Date accessed: 2019. afl-unicorn. https://github.com/Battelle/afl-unicorn.
[3] Date accessed: 2019. american fuzzy lop. http://lcamtuf.coredump.cx/afl/.
[4] Date accessed: 2019. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.
[5] Date accessed: 2019. OSS-Fuzz - Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz.
[6] Date accessed: 2019. Project Triforce: Run AFL on Everything! https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything.
[7] Date accessed: 2019. TriforceAFL. https://github.com/nccgroup/TriforceAFL.
[8] Sunha Ahn and Sharad Malik. 2014. Automated firmware testing using firmware-hardware interaction patterns. In *CODES+ISSS*. 25:1–25:10.
[9] V. Alimi, S. Vernois, and C. Rosenberger. 2014. Analysis of embedded applications by evolutionary fuzzing. In *HPCS*. 551–557.
[10] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *USENIX Security*.
[11] Tom De Schutter. 2014. *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press.
[12] Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. Closing the RISC-V Compliance Gap: Looking from the Negative Testing Side. In *DAC*.
[13] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. 2018. Extensible and Configurable RISC-V based Virtual Prototype. In *FDL*. 5–16.
[14] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. 2019. Early Concolic Testing of Embedded Binaries with Virtual Prototypes: A RISC-V Case Study. In *DAC*.
[15] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. 2019. Verifying Instruction Set Simulators using Coverage-guided Fuzzing. In *DATE*. 360–365.
[16] 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).
[17] Alex Horn, Michael Tautschnig, Celina G. Val, Lihao Liang, Tom Melham, Jim Grundy, and Daniel Kroening. 2013. Formal co-validation of low-level hardware/-software interfaces. In *FMCAD*. 121–128.
[18] Bo-Yuan Huang, Sayak Ray, Aarti Gupta, Jason M. Fung, and Sharad Malik. 2018. Formal security verification of concurrent firmware in SoCs using instruction-level abstraction for hardware. In *DAC*. 91:1–91:6.
[19] Nassima Kamel and Jean-Louis Lanet. 2013. Analysis of HTTP Protocol Implementation in Smart Card Embedded Web Server. In *IJINS*. 417–428.
[20] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. 2014. Prospect: Peripheral Proxying Supported Embedded Code Testing. In *ASIA CCS*. 329–340.
[21] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. 2010. Experimental Security Analysis of a Modern Automobile. In *IEEE S & P*. 447–462.
[22] H. Lee, K. Choi, K. Chung, J. Kim, and K. Yim. 2015. Fuzzing CAN Packets into Automobiles. In *AINA*. 817–821.
[23] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. In *ISSTA*. 261–272.
[24] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* (1990), 32–44.
[25] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. 2018. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *NDSS*.
[26] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. 2011. SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale. In *USENIX Security*. 24–24.
[27] Fabian van den Broek, Brinio Hond, and Arturo Cedillo Torres. 2014. Security Testing of GSM Implementations. In *ESSoS*. 179–195.
[28] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. 2014. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*.