Early Concolic Testing of Embedded Binaries with Virtual Prototypes: A RISC-V Case Study*

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

Extensive testing of IoT SW is very important to prevent errors and security vulnerabilities. In the SW domain the automated concolic testing technique has been shown very effective.

In this paper we propose an approach for concolic testing of binaries targeting RISC-V systems with peripherals. Our approach works by integrating the *Concolic Testing Engine* (CTE) with the architecture specific *Instruction Set Simulator* (ISS) inside of a *Virtual Prototype* (VP). We provide a designated *CTE-interface* to integrate (SystemC-based) peripherals into the concolic testing by means of SW models. This combination enables a high simulation performance at binary level with comparatively little effort to integrate peripherals with concolic execution capabilities. Our approach has been effective in finding several buffer overflow related security vulnerabilities in the FreeRTOS TCP/IP stack.

1. INTRODUCTION

All predictions agree that the momentum already gained by the *Internet-of-Things* (IoTs) will not stop, quite the contrary, i.e. a paramount growth in both, the number of connected devices and the size of the potential business is expected. From the development perspective of IoT devices several challenges arise which often manifest in conflicting requirements: high performance, low power, correctness, remote accessibility, security and reliability, to just name a few. Among these, correctness and security are of fundamental significance. Hence, extensive testing of IoT SW is very important to prevent errors and security vulnerabilities.

In the SW domain the automated concolic testing technique has been shown very effective. Essentially, concolic testing successively explores new paths through the SW program by solving symbolic constraints, that are tracked alongside the concrete execution. This combination of concrete with symbolic execution enables an efficient exploration of a large set of different program paths.

* 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, 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.

DAC '19, June 2-6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06...$15.00
https://doi.org/10.1145/3316781.3317807

A full IoT solution consists of SW and HW. The traction coming from open source SW recently reached open source HW. In particular, *RISC-V* has started to become a game changer for IoT processors. RISC-V is an open and free *Instruction Set Architecture* (ISA) [23]. Since 2015 the RISC-V ISA standard is maintained by the non-profit RISC-V foundation [3] which has more than 200 members aiming innovation. However, to the best of the authors knowledge, concolic testing of binaries targeting RISC-V systems with peripherals has not been considered yet.

Contribution: In this paper we propose such a concolic testing approach for binaries targeting RISC-V systems with peripherals[1]. Our approach works by integrating the *Concolic Testing Engine* (CTE) with the architecture specific *Instruction Set Simulator* (ISS) inside of a *Virtual Prototype* (VP). We provide a designated *CTE-interface* to integrate additional peripherals through a SW-library that is linked with the RISC-V binary under test into a combined RISC-V binary. Our *CTE-interface* consists of a small set of interface functions, tailored for SystemC-based peripherals with TLM 2.0 communication [12, 18]. Our approach enables a high simulation performance, by tightly integrating the concolic execution engine with the VPs ISS for the specific target ISA, and at the same time significantly reduces the implementation effort in adding concolic execution capabilities to each peripheral (since peripherals are executed as SW on top of the VP and hence *inherit* the concolic execution capabilities of the VP). Our experiments demonstrate the efficiency and applicability of our approach in analyzing real-world embedded applications. We found several buffer overflow related security vulnerabilities in the FreeRTOS TCP/IP stack.

Related Work: Most of existing work on concolic testing and symbolic execution has been focusing on testing non-embedded SW that does not or only rarely interacts with HW. KLEE [6] and SAGE [11] are among the pioneering work that has made the techniques really work for real-world SW. While KLEE operates on the LLVM intermediate representation and requires the source code, SAGE works directly on the x86 binary. Operating on the binary level is important to achieve accurate verification results, as binaries are code that will actually be deployed. Thus, despite being more difficult due to the complexity of lower-level constructs, concolic testing of binaries (mostly x86 and ARM to some extent) is increasingly being considered by subsequent work, e.g. S2E [8], Mayhem [7] or Angr [22]. More recently, concolic/symbolic testing has gained attention from the HW verification community, see e.g. [4, 21] for RTL and [14, 15, 19] for SystemC VPs.

For embedded SW, these approaches are not sufficient due to the much stronger dependence of the SW on the underlying HW. This gave rise to a number of specialized HW/SW symbolic (concolic) co-validation approaches that are related to ours. They mainly differ on how the underlying HW is being integrated. [5, 16] use virtual peripheral models manually extracted from QEMU. [20] on the other hand integrates HW Verilog

[1]Please note, we use the RISC-V ISA as a case-study. Our proposed methodology is applicable to other ISAs as well. Also, visit `www.systemc-verification.org` for our most recent VP-based approaches.

models. Recently, [17] proposed a new formalism called instruction level abstraction to formally model SW-visible behavior of HW. This enables more scalable symbolic exploration but it is unclear whether the abstraction can be fully automated. These approaches operate at the source-code level. FIE [10] is more similar to ours in the sense that it targets SW running on a specific HW platform. FIE brings a set of modifications to KLEE to resemble an MSP430-based execution environment and optimizations for more scalable concolic testing (e.g. memory smudging and state pruning). However, it still requires the SW source code and cannot handle inline assembly instructions that often can be found in low-level embedded SW. Therefore, Inception [9] introduces an assembly to LLVM-IR lifting approach. AVATAR [25] extends S2E to allow hybrid binary concolic testing with physical devices.

Note that while we describe our approach in some details, we do not claim the concolic testing technique to be our central contribution. Rather it is the combination on how the concolic execution engine is integrated with the VP with our method for integration of (SystemC-based) peripherals. By this means we provide a practical framework filling the gap of concolic testing for embedded RISC-V binaries that is not possible with any existing framework. Conceivably, Angr – operating on the VEX intermediate representation at binary level – can be extended to support RISC-V instructions. However, due to its focus on non-embedded SW, extending Angr to the same extent of capability of handling RISC-V binaries and peripherals would require a lot of effort. Furthermore, based on its promises, RISC-V deserves a tailored platform that is more lightweight and amenable to specific optimizations.

2. PRELIMINARIES

2.1 RISC-V

RISC-V is an open and free *Instruction Set Architecture* (ISA). The ISA consists of a mandatory base integer instruction set (denoted RV32I, RV64I or RV128I with corresponding register widths) and various optional extensions denoted as single letters, e.g. M (integer multiplication and division), C (compressed instructions), etc. Thus, RV32IMC denotes a 32 bit core with M and C extension. The instruction set is very compact, RV32I consists of 47 instructions and the M extension adds additional 8 instructions. All RV32IM instructions have a 32 bit width and use at most two source and one destination register. The C extensions adds 16 bit encodings for common operations.

The RISC-V ISA also defines *Control and Status Registers* (CSRs), which are registers serving a special purpose. For example the *mtvec* (Machine Trap-Vector Base-Address) CSR stores the address of the trap/interrupt handler. Furthermore, the ISA provides a small set of instructions for interrupt handling (*wfi*, *mret*) and interacting with the system environment (*ecall*). For a comprehensive description of the RISC-V ISA please refer to the official specifications [23, 24].

2.2 Concolic Testing

Overview: Concolic testing is a technique to successively explore new paths through the SW program. It uses concolic values in place of pure concrete ones. A concolic value is a pair with a concrete N and a symbolic part x written as (N,x). The concrete part is always available. We denote a concolic value to be *symbolic*, if the symbolic part x is also available. Otherwise ($x = /$) we call it *concrete*. For concolic testing, input variables (or memory regions in general) are marked to be symbolic. Furthermore, a (symbolic) *Execution Path Condition* (EPC) and a list of (symbolic) *Trace Conditions* (TCs) are tracked. After each execution, an SMT solver checks each emitted TC for satisfiability. Each satisfiable TC represents a new testcase (input). An input assigns concrete values to symbolic variables (i.e. memory regions), thus preventing them being assigned random values and guiding exploration towards a specific path.

Concolic testing starts by (concretely) executing a random path through the SW program (i.e. all variables marked symbolic are simply assigned random values). During SW execution the symbolic input constraints are propagated alongside the concrete execution. Let us consider an example operation combining two operands A and B. In case one operand $A = (1, x_0)$ has a symbolic value and the other $B = (2, /)$ has not

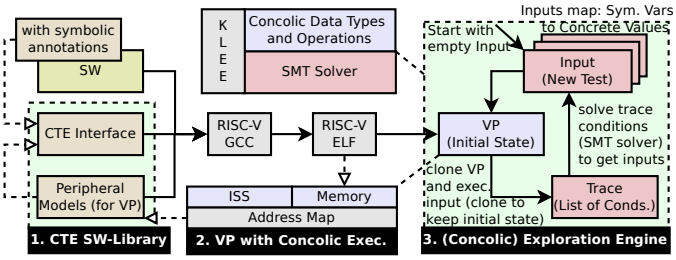

Figure 1: Concolic Testing Engine (CTE) architecture overview

B will be first converted to $(2, 2_S)$, where 2_S is an SMT expression representing the constant value 2. Thus, for example $A + B$ would then result in $(3, x_0 + 2_S)$. In the following we often omit the S suffix in case the context is clear and write 2_S simply as 2.

At each branch instruction with symbolic condition $s = (c, x)$, the path condition EPC is extended and a trace condition TC_i is emitted, either: 1) $TC_i = EPC \wedge \neg x$ and then $EPC := EPC \wedge x$ in case the branch is taken (c evaluates to *true* in the concrete execution), or 2) $TC_i = EPC \wedge x$ and then $EPC := EPC \wedge \neg x$, otherwise. For verification purposes, *assume(s)* adds x to the current EPC to prune irrelevant paths and emits a TC in case c is *false*. *Assert(s)* checks c and emits a $TC = EPC \wedge \neg x$ in case c is *true*. Then EPC is extended with x.

Concretization: Symbolic values (N,x) can be concretized, i.e. converted to concrete values $(N, /)$, by adding the constraint $N = x$ to the path condition EPC. This can be useful in case parts of the simulation only support concrete values. Optionally, trace conditions can be emitted to (try) generate different concrete values N (e.g. the minimum and maximum possible values would be good candidates).

3. CONCOLIC TESTING ENGINE (CTE) FOR RISC-V EMBEDDED BINARIES

CTE enables concolic testing of binaries targeting RISC-V systems with peripherals. We start with an overview of CTE. Then, we present our approach on integrating peripherals into CTE in more details and finally show an example that illustrates how CTE interoperates with SW in combination with peripherals.

3.1 Overview

Fig. 1 shows an overview on the architecture of CTE. Essentially, CTE consists of three parts:

1) A SW-library that contains *CTE-interface* functions and a set of peripheral models. The CTE functions allow to declare and reason about symbolic variables and enable CTE to interface with peripheral models.

2) A VP that is operating with concolic, instead of concrete, data types. The VP essentially consists of an ISS that executes instructions one after another and a memory that stores code and data. Both, ISS and memory use concolic data types and propagate symbolic constraints during program execution. The ISS supports the RISC-V RV32IMC ISA. Additional peripherals are integrated into the simulation through the CTE SW-library.

3) A concolic exploration engine that is successively exploring new paths through the SW program (binary) by leveraging the VP and solving constraints on symbolic variables using an SMT solver.

3.1.1 Initialization and Exploration

CTE of SW works as follows: First, the SW is compiled together with the CTE SW-library into a RISC-V ELF (binary file). Variables, representing input data, are marked to be symbolic (using the CTE-interface functions). Then, the (concolic execution) VP is instantiated and the (combined) RISC-V ELF is loaded into the VPs memory. The exploration engine then starts by assigning each symbolic variable a random value (empty input) and then successively generates new inputs (based on the observed constraints) to explore different paths through the SW program. In particular it will try to generate new inputs in case of a symbolic branch condition or *assume* or *assert* function. The VP is cloned

```
1  static uint32_t scaler = 25;
2  static uint32_t filter = 0;
3  static uint32_t data = 0;
4  #define SCALER_REG_ADDR 0x00
5  #define FILTER_REG_ADDR 0x04
6  #define DATA_REG_ADDR 0x08
7
8  void update() {
9    // overwrite data with new concolic bytes
10   CTE_make_symbolic(&data, sizeof(data), "d");
11   CTE_assume(data >= MIN_SENSOR_VALUE && data <=
         MAX_SENSOR_VALUE);
12   data -= filter;
13
14   // PLIC receives interrupts, prioritize them and
         calls a CTE-function to notify the VP
15   plic_process_interrupt(2 /*IRQ_NUMBER*/);
16
17   // corresponds to simple thread wait in SystemC
         (or just a method process)
18   CTE_notify(&update, scaler*CYCLES_PER_MS);
19   CTE_return();
20 }
21
22 // corresponds to a simple TLM transaction
23 void transport(uint32_t addr, uint8_t *data,
       uint32_t size, _Bool is_read) {
24   CTE_assert (size == 4);//only access whole reg.
25   uint32_t *vptr = (uint32_t *)data;
26   uint32_t *reg = 0;
27
28   // pre-process actions
29   if (addr == SCALER_REG_ADDR) {
30     if (!is_read)
31       CTE_notify(&update, scaler*CYCLES_PER_MS);
32     reg = &scaler;
33   } else if (addr == DATA_REG_ADDR) {
34     reg = &data;
35   } else if (addr == FILTER_REG_ADDR) {
36     reg = &filter;
37   } else { assert (0 && "invalid addr"); }
38
39   if (is_read) *vptr = *reg;
40   else *reg = *vptr;
41
42   // post-process actions
43   if (addr == FILTER_REG_ADDR && !is_read)
44     if (filter >= MIN_SENSOR_VALUE)
45       filter = MIN_SENSOR_VALUE+1;
46
47   CTE_return();
48 }
```

Figure 2: Simple sensor peripheral that illustrates the concept on peripheral modeling in combination with the CTE-interface.

each time before executing a new input to preserve the initial VP state and allow exploration of different paths. CTE continues until all inputs have been processed or a runtime check fails. CTE checks for SW assertion violations as well as some generic error sources: null pointer dereference, access (read or write) of an illegal memory address and invalid jump target (either not properly aligned or a jump to an illegal instruction). In our experimental evaluation we also check for overflows of heap allocated memory in FreeRTOS (more details will follow later).

3.1.2 Peripheral Integration

To implement CTE for embedded binaries (targeting a specific ISA) with peripherals, there are in general two fundamental choices with different trade-offs between execution performance and implementation effort: 1) Integrate concolic execution into every component of the VP, i.e. including every peripheral. This fully specialized solution requires significant effort (for each peripheral) but leads to the best performance. 2) Use a generic (x86) symbolic/concolic execution engine like S2E and run the VP, which in turn simulates the embedded binary, inside of S2E. This generic solution requires only little integration effort but has a significant performance overhead, in particular due to running the (VPs) ISS inside another simulator (S2E).

We aim to combine the benefits of both approaches and alleviate their respective major disadvantages. Our approach enables a high simulation performance, by tightly integrating the concolic execution engine with the VPs ISS for the specific target ISA (RISC-V in our case). This avoids the (very significant) additional interpretation layer in the ISS, which arguably is the component with the greatest performance impact. At the same time we integrate peripherals into the VP by providing SW models of the peripherals. Thus, peripherals are compiled to the RISC-V instruction set and normally executed on our VP alongside the actual binary that is tested. This has the major benefit, that the peripherals are executed with concolic data types and hence propagate symbolic constraints without further modification. While this involves a small (currently manual) transformation step to obtain an appropriate peripheral SW model (we use an existing SystemC model as starting point), it is much less effort than integrating concolic execution capabilities directly into the (SystemC-based) peripheral. Hence, our approach enables a fast simulation performance with comparatively little implementation effort.

In the following we start with an example that demonstrates our main modeling concepts for peripherals and then present how peripherals are accessed from SW and interoperate with the concolic exploration engine.

3.2 Peripheral Modeling Concepts

For illustration of our peripheral modeling concepts, Fig. 2 shows a simple sensor peripheral. It provides three (memory mapped) 32 bit registers: *data* holds the most recent generated sensor data, *scaler* controls how fast new sensor data is generated, and *filter* is applied on the generated data. Register access is provided through the *transport* function, which roughly corresponds to a TLM *b_transport* function[2]. New sensor data is periodically generated by the *update* function (Line 8). Therefore, the data register is overwritten with a new symbolic value (Line 10) and constrained to stay in the sensor range (Line 11). An interrupt is triggerd by calling the *PLIC* peripheral (a RISC-V specific *Platform Level Interrupt Controller*, implementation not shown) in Line 15. PLIC will prioritize incoming interrupts and eventually notify the ISS using the *CTE_trigger_irq* function. *CTE_notify* instructs the VP to call the function (address provided as first argument) after the specified number of execution cycles has passed (second argument). In case the function already has a pending notification, it will be reset. To handle notifications, the VP has a simple timing model that assigns each RISC-V instruction a fixed number of cycles, which are added up during simulation. *CTE_notify* allows to model simple processes. Currently, we only support concrete delay arguments to *CTE_notify*. In case a symbolic argument is used, it will be concretized. *CTE_return* returns execution control back to the SW program, which is interrupted in order to enter peripheral functions (*transport* and *update*). We present more details on this in the following (sub-)section. Finally, we provide the *CTE_get_cycles* function to obtain the current number of simulation cycles from the ISS to model the RISC-V specific *CLINT* (Core Local INTerruptor) peripheral. *CLINT* triggers periodic (configurable) timer interrupts and allows the SW to obtain the current simulation time.

3.2.1 Software Side

Fig. 3 shows an example SW that is accessing the sensor peripheral (shown in Fig. 2). The SW first installs an interrupt handler to listen for the sensor interrupt (Line 11). Then, it initializes the sensor (Lines 14-15) using memory mapped I/O and waits for the sensor interrupt (which indicates new data). The *filter* register is assigned a symbolic value, while *scaler* is assigned a concrete one. Finally, the SW retrieves the sensor data value and checks its validity (Lines 20-21).

To handle memory mapped accesses, the VP has an address map that contains specific non-overlapping address ranges for the memory and every peripheral. This address map information is obtained from a configuration file. Every memory access transaction, i.e. RISC-V *load*/*store*

[2]It is easily possible to write a *transport* wrapper that stores the arguments in a *tlm_generic_payload* and then calls the real *transport* function to avoid modifying the peripheral. An optional *delay* parameter could also be passed in and returned back to the VP through the CTE-interface.

```
1  uint32_t*SENSOR_SCALER_REG_ADDR=(uint32_t*)0x10000000;
2  uint32_t*SENSOR_FILTER_REG_ADDR=(uint32_t*)0x10000004;
3  uint32_t*SENSOR_DATA_REG_ADDR=(uint32_t*)0x10000008;
4
5  volatile _Bool sensor_has_data = 0;
6  void sensor_irq_handler() {
7    has_sensor_data = 1;
8  }
9
10 int main() {
11   register_interrupt_handler(2 /*IRQ_NUMBER*/,
         sensor_irq_handler);
12   uint32_t filter;
13   CTE_make_symbolic(&filter,sizeof(filter),"f");
14   *SENSOR_FILTER_REG_ADDR = filter;
15   *SENSOR_SCALER_REG_ADDR = 50;
16
17   while (!sensor_has_data) { // check for sensor
18     asm volatile ("wfi");  // wait for any irq
19   }
20   uint32_t n = *SENSOR_DATA_REG_ADDR;
21   CTE_assert (n <= MAX_SENSOR_VALUE);
22   return 0;
23 }
```

Figure 3: Example SW that is accessing the sensor peripheral (shown in Fig. 2).

$$I_0\colon \square \xrightarrow{\text{F}=(31,f_0)} \text{if}(\text{F} \geq 16) : T \xrightarrow{\text{F}=(17,/),\text{D}=(4,d_0)} \text{assume}(D \geq 16 \wedge D \leq 64) : F$$

$$I_2 : \square \xrightarrow{\text{F}=(23,f_0)} \text{if}(\text{F} \geq 16) : T \xrightarrow{\text{F}=(17,/),\text{D}=(45,d_0)} \text{assume}(D \geq 16 \wedge D \leq 64) : T \xrightarrow{\text{D}=(28,d_0-17)} \text{assert}(D \leq 64) : T$$

$$I_3 : \square \xrightarrow{\text{F}=(21,f_0)} \text{if}(\text{F} \geq 16) : T \xrightarrow{\text{F}=(17,/),\text{D}=(16,d_0)} \text{assume}(D \geq 16 \wedge D \leq 64) : T \xrightarrow{\text{D}=(\text{UINT\_MAX},d_0-17)} \text{assert}(D \leq 64) : F$$

Figure 4: Example concolic execution paths through the SW and peripheral shown in Fig. 3 and Fig. 2.

instruction, is matched against the address ranges and then routed accordingly. This step involves a *global-to-local* address translation. For example executing a RISC-V SW (store word) instruction with address 0x10000004 (Line 14 in Fig. 3), and assuming the sensor is mapped to e.g. the range (0x10000000, 0x1000ffff), then the access address is translated to 0x4 and routed to the sensors *transport* function. This step involves a context switch to the peripheral SW and then back to the main SW. In the following we provide more details on this step.

3.2.2 Context Switching: SW and Peripheral

To switch execution context from the SW to a peripheral function *FN*, the VP (actually the ISS) saves its current execution context (i.e. register values and program counter) to an internal stack. Then, the program counter is simply set to the address of *FN*. To return, the previously stored execution context is re-stored. Therefore, we introduced the *CTE_return* function[3]. Using a stack to save the execution context allows peripherals to access other peripherals memory through a memory mapped access.

Arguments between the VP and the SW peripherals are passed through registers and a (transaction) data array placed in memory (to hold larger arguments). The *transport* function takes four arguments: the (local) access address, the number of bytes to access, a data pointer (*uint8_t**), and a boolean flag indicating the access type (read/write). The data pointer is setup to point to the data array. These four arguments are stored in the VP registers (*a0* to *a3*, following the RISC-V function calling convention) prior to switching context to the transport function. The address of the *transport* function for each peripheral as well as the data array is obtained (during VP initialization) by parsing the ELF symbols (i.e. given a name of a symbol, it is possible to obtain its address).

3.3 Concolic Testing Example

This section illustrates how CTE interoperates with SW and peripherals. Fig. 4 shows the relevant execution paths for our sensor SW example. It shows the input, updates to the *filter* (F in Fig. 4) and *data* (D in Fig. 4) variables and instructions generating trace conditions.

CTE starts without input constraints ($I_0 = \{\}$), thus assigning each symbolic variable a random value and therefore exploring a random path through the SW program. We assume for this first run that *filter* and *data* are (randomly) assigned the concolic values $(31, f_0)$ and $(4, d_0)$ in Line 13 and Line 10, respectively. Furthermore, for this example we define the constants *MIN_SENSOR_VALUE=16* and *MAX_SENSOR_VALUE=64*. The first trace condition $TC_1 = \neg(f_0 \geq 16)$ is generated by the branch execution in Line 44, due to the memory mapped write of the symbolic filter value in the SW (Line 14). The second trace condition $TC_2 = (f_0 \geq 16) \wedge (d_0 \geq 16 \wedge d_0 \leq 64)$ is generated by the *assume* function in Line 11, due to the symbolic data value, while the main SW waits for an interrupt (Line 18). Thus, the *assume* function evaluates to *false* and this path ends.

Both trace conditions are satisfiable, resulting in e.g. the two new inputs $I_1 = \{f_0 = 1, d_0 = 25\}$ and $I_2 = \{f_0 = 23, d_0 = 45\}$ (obtained by using the SMT solver), respectively. It depends on the search strategy of the exploration engine, which input is considered next. For this example, we continue with I_2. Until the *assume* function (Line 11) it follows the same path as the initial execution. However, different concrete values are assigned at the *CTE_make_symbolic* functions based on I_2. Thus, this time the *assume* function evaluates to *true* and execution continues. Please note, no trace conditions have been generated until now to avoid re-exploration of the first execution path (and its descendants, i.e. I_1). Next, *filter* is applied to the generated *data* (Line 12) and an interrupt is triggered (Line 15). The main SW will leave the interrupt waiting loop, fetch the data register value (Line 20) and check the assertion in Line 21. The concrete part of the condition evaluates to *true*, the assert generates a trace condition $TC_3 = (f_0 \geq 16) \wedge \neg(d_0 - 17 \leq 64)$ and this path ends. TC_3 is satisfiable with e.g. $I_3 = \{f_0 = 23, d_0 = 16\}$. With I_3 the data register in Line 12 will underflow and thus the assert in Line 21 is violated. This is due to an incorrect filter value assigned in Line 45. It should use *minus one* instead of *plus one*. This example also shows that it is important to integrate the peripheral logic into the testing approach to avoid missing relevant behavior of the overall system.

4. EXPERIMENTS

We have implemented our proposed CTE for concolic testing of binaries targeting RISC-V systems with peripherals. As symbolic backend we use KLEE v1.4.0 with STP solver v2.3.1. We evaluate CTE in two steps: First, we evaluate the performance of our approach in Section 4.1. Then, we apply CTE to test the FreeRTOS TCP/IP stack. Our approach has been effective in finding various security vulnerabilities.

4.1 Performance Evaluation

We evaluate the performance of our CTE by comparing against: 1) a generic symbolic execution solution using S2E, and 2) a VP supporting RISC-V binaries [13]. The ISS in VP is similar to that in CTE but only supports concrete execution. Hence, VP uses DMI for memory access and native (C++ uint32_t) data types for instruction execution (instead of a custom concolic data type that propagates a concrete and symbolic value for each instruction). Furthermore, VP is using normal SystemC peripherals. For the S2E solution, we run VP simulating the RISC-V binary on top of S2E, thus allowing symbolic execution of RISC-V binaries with little implementation overhead (only a small interface layer is required to propagate the *make_symbolic*, *assume* and *assert* functions from the RISC-V binary through the VP into S2E).

Table 1 shows the results. All runtimes are provided in seconds. It is separated into 11 columns. The columns show, in order from left to right,

[3]Though, in general it is also possible to automatically infer the end of *FN* by e.g. monitoring function calls and returns.

Table 1: Experiment results – timeout (T.O.) set to 7200 seconds (2 hours)

Benchmark	#instr	LOC		Sim. Time (sec.)			CTE Statistics			
		C	ASM	VP	S2E	CTE	FoI S2E	stime	#paths	#queries
qsort	52,343,639	212	1126	1.83	122.33	3.87	31.6x	/	1	/
sha512	75,997,581	175	1132	2.71	136.90	4.62	29.6x	/	1	/
dhrystone	238,000,584	273	515	8.42	421.60	19.70	21.4x	/	1	/
freertos-sensor	21,348,342	217	18230	1.35	107.37	2.19	49.0x	/	1	/
counter/s	642,710	31	79	/	1264.21	41.74	30.3x	37.82	452	904
fibonacci/s	683,970	37	170	/	394.14	1.90	197.5x	0.15	22	41
qsort/s	58,648	219	1192	/	T.O	32.90	>218x	32.58	121	600
freertos-sensor/s	301,697,668	241	18329	/	422.78	38.66	10.9x	1.60	463	968

the benchmark name (column: Benchmark), number of instructions executed (column: #instr), number of lines (LOC) in C and RISC-V assembly (columns: C and ASM), the simulation time in seconds on the VP, S2E and our CTE (columns: VP, S2E and CTE), and relevant statistics for our CTE: factor of improvement (FoI) compared to S2E (column: FoI S2E), time in seconds spent with solver queries (column: stime), number of different paths explored (column: #paths) and number of solver queries (column: queries).

Table 1 is separated into two halves. The upper half shows benchmarks that only operate on concrete values, while the lower half shows benchmarks including symbolic values (hence exploring multiple execution paths). For symbolic benchmarks, the reported number of instructions is combined over all paths. As benchmarks we use *qsort* from the *newlib* C library, a standard *dhrystone* implementation, *sha512* performs a checksum computation, *counter* involves counting related constraints, *fibonacci* uses a recursive implementation (function call intensive) and *freertos-sensor* embeds a sensor application into FreeRTOS tasks. The */s* name suffix denotes that the benchmark uses symbolic values.

Comparing our CTE approach with VP, it can be observed that the overhead of using concolic execution is around a factor of 2.2x. The overhead of running the VP inside of S2E is between 21x and 49x compared to CTE on our benchmark set. The main reason is the additional interpretation overhead in the ISS component of the VP. With symbolic values, the overhead of S2E compared to CTE is even more pronounced. Speed-ups of multiple orders of magnitude can be observed. The (symbolic) state space representation is more heavy-weight and complex in S2E. Furthermore, the additional abstraction layer leads to increased number of forks in the symbolic execution engine. In one case (*freertos-sensor/s*) we observed only an improvement of 10.9x. The reason is that CTE explores all paths from the beginning instead of forking an active execution. This leads to significant overhead in re-initializing the FreeRTOS memory image. This inefficiency can be solved by cloning the CTE execution state after initialization is finished and then start the exploration engine from this point. Our evaluation shows that an ISA specific symbolic execution engine can provide significant speed-ups compared to a generic symbolic execution engine (in particular optimizations of the ISS are important). At the same time, our approach requires considerable less implementation effort compared to a fully specialized symbolic execution engine, by integrating SW models of peripherals instead of adding symbolic execution capabilities to every periperal.

4.2 Testing the FreeRTOS TCP/IP Stack

To evaluate the effectiveness of our tool we have analyzed the TCP/IP stack of FreeRTOS v10.0.0 in combination with the RISC-V port of the FreeRTOS kernel. Essentially, we inject a single (small) packet with symbolic size and content into the TCP stack and check for generic execution errors (including FreeRTOS assertions) and heap buffer overflows.

4.2.1 Test Setup

IP packets are processed in FreeRTOS inside the platform independent *IP-task*. The IP-task requires a driver to receive (send) packets from (to) the network card. The FreeRTOS documentation provides a generic porting guide [2] to create a new driver and connect it with the IP-task. Using the generic driver code, it is only necessary to implement three additional glue functions: 1) ReceiveSize, 2) ReceiveData, and 3) SendData.

Essentially, to receive a packet, the driver waits for a network card interrupt. Then, it asks the network card for the number of received bytes (*ReceiveSize* function), allocates a buffer on the heap large enough to hold *ReceiveSize* bytes, and asks the network card to store the data in the buffer (ReceiveData function). Finally, the allocated buffer is delivered to the (platform independent) IP-task for further processing. We model the network card as a peripheral (as described Section 3.2). Inside the peripheral we store a packet buffer of 512 bytes with symbolic content. Furthermore, we define a symbolic integer variable N and add the assumption $N \leq 512$ in the peripheral. The driver glue functions access our peripheral using memory mapped I/O. Our model simply ignores outgoing packets, returns N when asked for *ReceiveSize* and copies N bytes into the data pointer provided by *ReceiveData*.

In the (SW) main function, we create and initialize the FreeRTOS network (IP-task) and driver processing task and start the FreeRTOS scheduler (available from the FreeRTOS kernel). Then we create and bind a TCP socket in *listening* mode to ensure that the IP-task does not drop TCP packets (which is the case if no initialized TCP socket is available). Finally, we start the actual symbolic testing by calling the *init* function of our peripheral, which in turn will trigger an interrupt notifying the driver to receive and process a packet. Interrupts are processed using our *PLIC* peripheral. Timer interrupts for the FreeRTOS scheduler processing are generated using our *CLINT* peripheral.

Please note, the IP-task is non-terminating and will wait for new packets from the driver indefinitely. Thus, we added a switch to stop simulation after one packet has been processed, to allow CTE to explore other paths as well (alternatively we could bound the search depth in CTE).

4.2.2 Heap Buffer Overflow Detection

To check for heap buffer overflows, we provide wrappers for the FreeRTOS memory management functions *pvPortMalloc* (allocate memory) and *vPortFree* (free memory). We use the linker options *-Wl,–wrap=pvPortMalloc -Wl,–wrap=vPortFree* with GCC to automatically redirect all accesses of *pvPortMalloc* to our wrapper function *__wrap_pvPortMalloc*. The original function is available as *__real_pvPortMalloc* (*vPortFree* handled in same way).

Fig. 5 shows our wrapper functions. The *pvPortMalloc* wrapper allocates a larger than requested memory block, by adding additional bytes (protected zones) before and after the requested memory block. These protected zones are registered in CTE (Line 9). CTE will monitor all load/store operations and trigger a simulation error in case of a write (or read) access inside of the zones. CTE generates trace conditions in case of a symbolic memory address. The *vPortFree* wrapper unregisters the two corresponding protected zones from CTE (Line 15), which does also check for double free and non-allocated blocks, and then calls the real *vPortFree* function of FreeRTOS.

4.2.3 Test Results

We run CTE until we find the first error. We then repeatedly fix the error and re-run CTE, until no more error is found. We have set 4 hours (14400 seconds) as time limit to terminate the analysis. Table 2 presents the errors that we have found. All errors are related to buffer overflows of heap allocated memory and can therefore lead to serious security vulnerabilities. Besides a description, Table 2 shows additional relevant information similar to Table 1.

```
1 #define PROT_ZONE_SIZE 512 //in bytes
2
3 void *__wrap_pvPortMalloc( size_t xWantedSize ) {
4   size_t xSize = xWantedSize + 2*PROT_ZONE_SIZE;
5   // call the real FreeRTOS pvPortMalloc function
6   uint8_t *p=(uint8_t*)__real_pvPortMalloc(xSize);
7   if ( p == NULL ) return NULL;
8   void *addr = (void *)(p + PROT_ZONE_SIZE);
9   CTE_register_protected_memory( addr,
        xWantedSize, PROT_ZONE_SIZE );
10  return addr;
11 }
12
13 void __wrap_vPortFree( void *pv ) {
14   CTE_assert( pv != NULL );
15   CTE_free_protected_memory( pv );
16   void *pv_real = ((uint8_t *)pv) - PROT_ZONE_SIZE;
17   // call the real FreeRTOS vPortFree function
18   __real_vPortFree( pv_real );
19 }
```

Figure 5: Wrappers for the real FreeRTOS *pvPortMalloc* and *vPortFree* memory management functions.

Table 2: Errors found in testing the FreeRTOS TCP/IP stack implementation – time in seconds

Error description	time	stime	#paths	#queries	#instr
1: A malformed IP packet header length causes an integer overflow which leads to a *memmove* operation with a size close to UINT_MAX.	0.52	0.13	7	30	1,106,862
2: Multiple buffer overflows accessing (read) non-existing fields in the DNS and NBNS packet parser.	4.52	0.58	62	155	10,256,275
3: Buffer overflow (write) in the DNS reply generator, due to missing packet length checks, causing heap corruption.	4.94	0.50	90	207	14,946,642
4: Various buffer overflows (read), during TCP options checking, due to missing buffer length checks (in case of malformed packets).	10.90	2.55	108	346	20,148,331
5: Integer overflow in length calculation causes NBNS processing code to allocate a large reply buffer and fill it by reading beyond a much smaller input buffer.	380.92	33.00	6,135	15,495	1,025,290,638
6: NBNS processing code allocates not enough memory to hold the complete reply message for some malformed UDP packet sizes, leading to a buffer overflow.	456.88	36.23	8,055	20,171	1,343,648,874

It can be observed that our concolic analysis, despite currently being without sophisticated state-of-the-art search heuristics, is already very effective in finding errors. Some of the detected errors require very specific inputs that are hard to find without formal methods. Note that these errors/vulnerabilities were present within the FreeRTOS TCP/IP stack for a long time already. At the time of writing, it appears that other researchers might also have discovered these errors [1]. These have been fixed only very recently in the newest FreeRTOS Version (v10.1.0). Despite being assigned CVEs (detailed list in [1]), neither the errors nor how they have been found are disclosed.

5. CONCLUSION

In this paper we proposed an approach for concolic testing of binaries targeting RISC-V systems with peripherals. We embed the *Concolic Testing Engine* (CTE) into the core VP and integrate peripherals through a SW-library. This enables high execution performance by specializing the ISS for the target architecture and at the same time significantly reduces implementation effort in adding concolic execution capabilities to each peripheral. Our approach has been very effective in finding buffer overflow related errors in the FreeRTOS TCP/IP stack, which demonstrate the applicability and efficiency of our approach in analyzing real-world embedded applications. For future work we plan to: 1) Investigate using C++ peripheral models with a more comprehensive abstraction layer to avoid the current peripheral transformation step. 2) Integrate support for timers/interrupts with symbolic notification times to enable checking for timing/ordering related errors. 3) Evaluate different search heuristics to select inputs that increase code coverage more quickly.

6. REFERENCES

[1] FreeRTOS TCP/IP stack vulnerabilities put a wide range of devices at risk of compromise: From smart homes to critical infrastructure systems. `https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-put-wide-range-devices-risk-compromise-smart-homes-critical-infrastructure-systems`.

[2] Porting FreeRTOS+TCP to a different microcontroller. `https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_TCP/Embedded_Ethernet_Porting.html`.

[3] RISC-V Foundation. `https://riscv.org/`.

[4] A. Ahmed, F. Farahmandi, and P. Mishra. Directed test generation using concolic testing on RTL models. In *DATE*, pages 1538–1543, 2018.

[5] S. Ahn and S. Malik. Automated firmware testing using firmware-hardware interaction patterns. In *CODES+ISSS*, pages 25:1–25:10, 2014.

[6] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[7] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE S & P*, pages 380–394, 2012.

[8] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. In *ASPLOS*, pages 265–278, 2011.

[9] N. Corteggiani, G. Camurati, and A. Francillon. Inception: System-wide security testing of real-world embedded systems software. In *USENIX Security*, pages 309–326, 2018.

[10] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security*, pages 463–478, 2013.

[11] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

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

[13] V. Herdt, D. Große, H. M. Le, and R. Drechsler. Extensible and configurable RISC-V based virtual prototype. In *FDL*, pages 5–16, 2018.

[14] V. Herdt, H. M. Le, D. Große, and R. Drechsler. Compiled symbolic simulation for SystemC. In *ICCAD*, pages 52:1–52:8, 2016.

[15] V. Herdt, H. M. Le, D. Große, and R. Drechsler. Verifying SystemC using intermediate verification language and stateful symbolic simulation. *TCAD*, 2018.

[16] A. Horn, M. Tautschnig, C. G. Val, L. Liang, T. Melham, J. Grundy, and D. Kroening. Formal co-validation of low-level hardware/software interfaces. In *FMCAD*, pages 121–128, 2013.

[17] B. Huang, S. Ray, A. Gupta, J. M. Fung, and S. Malik. Formal security verification of concurrent firmware in SoCs using instruction-level abstraction for hardware. In *DAC*, pages 91:1–91:6, 2018.

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

[19] B. Lin, K. Cong, Z. Yang, Z. Liao, T. Zhan, C. Havlicek, and F. Xie. Concolic testing of SystemC designs. In *ISQED*, pages 1–7, 2018.

[20] R. Mukherjee, M. Purandare, R. Polig, and D. Kroening. Formal techniques for effective co-verification of hardware/software co-designs. In *DAC*, pages 35:1–35:6, 2017.

[21] S. Pinto and M. S. Hsiao. RTL functional test generation using factored concolic execution. In *ITC*, pages 1–10, 2017.

[22] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE S & P*, pages 138–157, 2016.

[23] 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.

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

[25] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, 2014.