RVX - A Tool for Concolic Testing of Embedded Binaries Targeting RISC-V Platforms[⋆]

Vladimir Herdt[1] [0000−0002−4481−057X], Daniel Große[1,2] [0000−0002−1490−6175], and Rolf Drechsler[1,3] [0000−0002−9872−1740]

[1] Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[2] Chair of Complex Systems, Johannes Kepler University Linz, Austria
[3] Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
{vherdt,grosse,drechsle}@informatik.uni-bremen.de

Abstract. We present RVX, a tool for concolic testing of embedded binaries targeting RISC-V platforms with peripherals. RVX integrates the *Concolic Testing Engine* (CTE) with an *Instruction Set Simulator* (ISS) supporting the RISC-V RV32IMC *Instruction Set Architecture* (ISA). Further, RVX provides a designated CTE-interface for additional extensions. It is an extensible command layer that provides support for verification functions and enables integration of peripherals into the concolic simulation. The experiments demonstrate the applicability and efficiency of RVX in analyzing real-world embedded applications. In addition, we found a new serious bug in the RISC-V port of the *newlib* C library.

Keywords: RISC-V · Concolic Testing · Verification · Embedded Binaries.

1 Introduction

Performing application *Software* (SW) verification on the binary level is very important to achieve accurate verification results. However, at the same time it is very challenging due to the detailed low level semantics. Concolic testing has been shown to be very effective for binary analysis [2,3,5,9]. Recently, we proposed a methodology for concolic testing of embedded binaries targeting platforms with peripherals, using the RISC-V *Instruction Set Architecture* (ISA)[4] as a case-study [7]. This initial prototype implementation has been extended, resulting in the tool RVX. To the best of our knowledge, RVX is the first available concolic testing tool targeting the RISC-V ISA[5]. In particular, RVX supports the RISC-V RV32IMC ISA, i.e. a 32 bit architecture with the mandatory

⋆ This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127 and within the project VerSys under contract no. 01IW19001 and within the project SATiSFy under contract no. 16KIS0821K, and by the German Research Foundation (DFG) as part of the Collaborative Research Center (Sonderforschungsbereich) 1320 EASE – Everyday Activity Science and Engineering, University of Bremen (`http://www.ease-crc.org/`) in subproject P04.

[4] Find the RISC-V ISA specification documents at `https://riscv.org/specifications/`.

[5] Visit `http://systemc-verification.org/risc-v` for the most recent updates on our RISC-V related approaches.

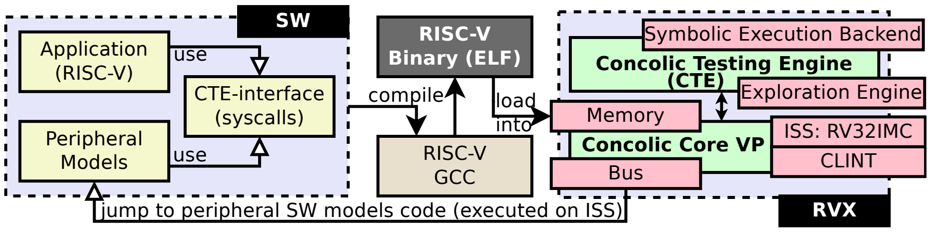

Fig. 1. RVX architecture overview. The Core VP is based on our RISC-V VP from [6].

base Integer instruction set together with the Multiplication extension and support for Compressed instructions, in combination with the RISC-V machine mode *Control and Status Registers* (CSRs) and interrupt handling instruction. In addition RVX provides a designated *Concolic Testing Engine* (CTE) interface to access verification functions from the SW and integrate additional peripherals into the concolic simulation by means of SW models. The CTE-interface peripheral integration is tailored for SystemC-based peripherals with TLM 2.0 communication [8]. Our experiments demonstrate the efficiency of RVX in analyzing real-world embedded binaries.

Compared to our initial paper [7], this tool paper focuses on additional implementation details and adds the following extensions and **contributions: 1)** several architectural improvements, including a search heuristic to speed-up bug hunting and an optimized memory system for more efficiency (lazy initialization and instruction fetch optimization); **2)** extended support for the RISC-V privileged ISA which enables to use the Zephyr *Operating System* (OS); and **3)** new set of experiments based on the Zephyr OS and we found a new serious bug in the RISC-V port of the *newlib* C library.

2 RVX Overview and Implementation

2.1 Architecture Overview

RVX is implemented in C++. Fig. 1 shows an overview of the architecture. RVX operates on the binary level. Starting point of the analysis is a RISC-V binary (ELF). The RISC-V binary is obtained by compiling and linking the SW application together with our CTE-interface and an optional set of peripheral SW models. We expect that calls to the verification functions (functions provided through our CTE-interface SW stub), to mark symbolic input variables and encode (safety) properties (i.e. *make_symbolic*, *assume* and *assert* with their usual semantic), have been embedded in the ELF already.

RVX performs concolic testing of the RISC-V ELF. Essentially, RVX consists of two parts: The concolic core *Virtual Prototype* (VP) and the CTE, as shown on the right side of Fig. 1. The CTE successively generates new inputs to explore new paths through the ELF. Based on the inputs the VP, and in particular the *Instruction Set Simulator* (ISS) component, performs the actual execution one after another and tracks symbolic constraints in order to generate new inputs. Therefore, the VP is operating with concolic data types in place of concrete values. Essentially, a concolic data type is a pair of a concrete value and a (optional) symbolic expression.

We implemented symbolic expressions as lightweight wrapper classes that provide a thin layer around KLEE [1] symbolic expressions. Beside, enabling to change the symbolic backend more easily, the wrapper provides expression simplification rules, based on term rewriting. We leverage KLEE constraint sets and use the solver API of KLEE (combining the counterexample and caching solvers) for constraint solving.

2.2 Exploration Engine and Memory Model

The **exploration engine** collects inputs in a priority queue to enable easy integration of different search algorithms. We prioritize inputs that lead to new program counter values (i.e. essentially increase branch coverage by selecting a branch direction that has not yet been executed). In case of multiple/none available candidates, we randomize the decision. By using the search depth as criteria a *Depth First Search* (DFS) or *Breadth First Search* (BFS) can be selected instead.

Memory is modeled as mapping from address to concolic byte. The mapping is constructed on-demand in a lazy fashion. A lazy implementation enables a significantly faster startup of the VP and reduces memory consumption, since the VP can have a large amount of memory and construction of symbolic data is resource intensive. The memory is initialized by loading the ELF file[6]. All other memory locations are uninitialized and will return a symbolic value on access. A memory access (read or write) with symbolic address will be concretized to a concrete address. Symbolic constraints are collected to enable generation of different concrete addresses. To speed-up instruction fetching we provide an option to load the *text* section of the ELF file into a native array and perform instruction fetching from that array.

2.3 CTE-Interface and Peripheral Integration

Verification Interface We provide the *make_symbolic*, *assume* and *assert* verification functions with their usual semantic, i.e. to make variables (memory locations in general) symbolic as well as constrain and check their values. Besides that, we provide two functions to set/unset memory regions to be access protected. These functions enable to e.g. implement heap buffer overflow protection by allocating a larger buffer and marking the beginning and end of the allocated buffer to be access protected. RVX reports an error in case an access to such a memory region is detected at runtime.

Peripheral Integration Both the actual application SW as well as the SW peripheral models are executed on the core VP. In case a memory access is routed to a SW peripheral the ISS performs a context switch to the peripheral handler. Therefore, the ISS sets the program counter to the handler address. Arguments between the ISS and the SW peripherals are passed through registers. Arguments are the access address, length, type (read or write) and a pointer to the data that is written or to be read (therefore a designated array is reserved). At the end of the handler, the CTE *return* function is called. It restores the previous execution context in the ISS.

[6] Essentially, this will copy code and data from the text and data sections, respectively, as well as zero initialize memory according to the *bss* section, as specified in the ELF program headers.

Besides the *return* function that transfers control back to the caller of a peripheral function, RVX provides four additional CTE-interface functions for peripheral integration: *notify*, *cancel*, *delay* and *trigger_irq*. Notify registers a callback function to be called after a specified delay by the core VP (based on the core VP timing model, i.e. execution cycles per instructions). Cancel removes a pending notification callback. Notify and cancel enable to implement a simple event-based synchronization targeting simple SystemC-based process (i.e. SC_THREAD and SC_METHOD) functionality. The *delay* function allows to annotate a processing delay that is added to the VPs internal timing model. The *trigger_irq* function triggers the given interrupt number. Please note, we provide an SW model of the RISC-V PLIC (*Platform Level Interrupt Controller*) that receives interrupts from other peripherals and prioritizes them. Finally, the PLIC is using the *trigger_irq* interface function to signal to the core VP that some interrupt is pending and requires processing.

Virtual Instructions Load instructions are split in the ISS into smaller virtual instructions. The reason is that they need to store the result of the memory access into a destination register (encoded in the instruction format). However, the result of a peripheral memory access is only available after context switching between the peripheral, which involves execution of several additional instructions (code from the peripheral) in-between. Splitting load instructions into two virtual instructions, where the first performs the memory access and the second stores the result in the destination register, enables the ISS to resume execution of the load instruction correctly.

2.4 ISS Main Loop

The ISS is the main component of the core VP. Algorithm 1 shows the instruction processing loop of the ISS. It executes instructions until the simulation terminates (Line 19, by issuing a special system call from the SW).

The ISS either executes application code (the default mode) or peripheral code (*in-peripheral* is *True*). In both cases the ISS might be executing virtual instructions (*in-virtual-mode* is True) to process load instruction correctly. Please note, *in-virtual-mode* is set to *False* when entering peripheral code and restored to its previous state on leaving (i.e. by storing *in-virtual-mode* on the context stack).

Pending notifications from peripherals (Lines 2-7) as well as external system calls (not CTE-interface, e.g. Zephyr OS context switches) and interrupts (Lines 8-10) are only processed if the ISS is currently executing normal application code. The *switch-to-trap-handler* function jumps to the trap/interrupt handler in SW, following the RISC-V trap/interrupt handling convention.

In each step either a virtual (Line 12) or normal (Lines 13-18) instruction is executed. In case of a normal instruction the ISS timing model is updated and the delay of the registered pending peripheral notifications is updated accordingly. The updates only happen for application code (Line 18), since the peripheral models emulate hardware devices and hence require a different timing model (we provide the *delay* system call to annotate the execution delay). The ISS uses a simple timing model that assigns each instruction a fixed (though configurable) execution time.

Algorithm 1: Main instruction processing loop inside the ISS

```
do
    if $\neg$ in-virtual-mode $\wedge$ $\neg$ in-peripheral then
        foreach $e \leftarrow$ pending-notifications do
            if delay(e) $\leq 0$ then                /* notification time elapsed */
                /* context switch to peripheral code                        */
                context-switch-to-event-handler(function(e))
                pending-notifications.remove(e)
                break
        if $\neg$ in-peripheral then
            if has-pending-system-call $\vee$ has-pending-enabled-interrupts then
                switch-to-trap-handler()  /* follow RISC-V convention */
    if in-virtual-mode then
        exec-virtual-step()        /* continue with instruction part */
    else
        /* exec-normal-step() might enter virtual mode and context
           switch to peripheral or set status to Terminated           */
        if in-peripheral then
            exec-normal-step()   /* peripherals have separate timing */
        else
            /* execute SW instruction and update core timing            */
            Instruction op $\leftarrow$ exec-normal-step()
            timing-and-pending-notifications-update(op)
while status != Terminated
```

3 Experiments and Conclusion

All experiments have been performed on an Ubuntu 16.04 Linux system with an Intel Core i5-7200U processor. As symbolic backend we use KLEE [1] v1.4.0 with STP [4] solver v2.3.1. Table 1 shows the results. The columns show: the application SW name, the number of executed instruction (*#instr*), lines of code in C and assembly, overall execution time (*time*), solver time (*stime*), number of concolic execution paths (*#paths*), and number of solver queries (*#queries*).

First, we consider two applications (each with and without a bug as indicated by the name) based on the Zephyr OS. Both applications use a consumer and producer thread and a sensor peripheral attached to an *Interrupt Service Routine* (ISR). The sensor generates symbolic data that is passed through the ISR to the producer (which applies post-processing) and finally the consumer (contains assertions) thread using message queues. The first application (*zephyr-filter-**) generates ten values, applies a filter and asserts that the sum and maximum value stays within a valid range. The second application (*zephyr-sort-**) generates six values, sorts the data (using the BSD *qsort* implementation) and then asserts that it is sorted. These applications demonstrate RVX's ability in analyzing complex embedded binaries.

Table 1. Experiment results (all times reported in seconds) - using RVX to analyze embedded SW targeting the RISC-V ISA and use the i) Zephyr OS, and ii) the RISC-V port of the *newlib* C library. In case of a bug (*-bug) RVX stops the analysis and reports a counterexample. Otherwise (*-ok), RVX performs an exhaustive concolic execution based on the symbolic inputs.

Application SW	#instr	C	ASM	time (S)	stime (S)	#paths	#queries
zephyr-filter-ok	421,206,516	265	4293	196.17	141.73	1024	2048
zephyr-filter-bug	24,628,768	265	4293	7.66	4.79	72	127
zephyr-sort-ok	180,274,083	408	4650	249.25	223.05	724	5043
zephyr-sort-bug	996,518	408	4648	1.43	1.27	4	34
memcpy-opt-bug	182,943	207	566	12.80	11.87	18	473

In addition, we found a new bug in the RISC-V port of the *newlib* C library. In particular, the bug is in the (speed) optimized *memcpy* function[7] and causes overwriting of nearly the entire address space due to an integer overflow in a length calculation. It is triggered by copying a small block to a destination (*dst*) address that is close to zero. We found the bug (last row in Table 1) by making the source (*src*) and *dst* address as well as the copy size symbolic. We added constraints that *src* and *dst* are not overlapping, and placed before the code segment. To catch buffer overflows we added a protected memory region (access is monitored by the ISS) around the buffer memory. Finally, we placed assertions after the memcpy to ensure it copies the data correctly from *src* to *dst*.

In summary, the experiments demonstrate the applicability and efficiency of RVX in analyzing real-world embedded binaries and finding bugs.

References

1. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. pp. 209–224 (2008)
2. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing mayhem on binary code. In: IEEE S & P. pp. 380–394 (2012)
3. Chipounov, V., Kuznetsov, V., Candea, G.: S2E: a platform for in-vivo multi-path analysis of software systems. In: ASPLOS. pp. 265–278 (2011)
4. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: CAV. pp. 519–531 (2007)
5. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: NDSS (2008)
6. Herdt, V., Große, D., Le, H.M., Drechsler, R.: Extensible and configurable RISC-V based virtual prototype. In: Forum on Specification and Design Languages. pp. 5–16 (2018)
7. Herdt, V., Große, D., Le, H.M., Drechsler, R.: Early concolic testing of embedded binaries with virtual prototypes: A RISC-V case study. In: DAC. pp. 188:1–188:6 (2019)
8. IEEE Std. 1666: IEEE Standard SystemC Language Reference Manual (2011)
9. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Krügel, C., Vigna, G.: SOK: (state of) the art of war: Offensive techniques in binary analysis. In: IEEE S & P. pp. 138–157 (2016)

[7] `https://github.com/riscv/riscv-newlib/blob/master/newlib/libc/machine/riscv/memcpy.c`