eBPF and Kprobes on rCore

Kprobes

Kprobes enables you to dynamically break into any kernel routine and collect debugging and performance information non-disruptively. You can trap at almost any kernel code address[1], specifying a handler routine to be invoked when the breakpoint is hit.

[1] E.g. which may cause a double fault

Kprobes

Kprobes makes a copy of the probed instruction and replaces the probed instruction with a breakpoint instruction. When a CPU hits the breakpoint instruction, control passes to Kprobes via trap handler. Kprobes executes the handler associated with the probe, then single step the probed instruction and restore execution back to normal.

Kprobes - implementation notes

- single step a single instruction out of it’s context is unsafe
  - privilege level differs
  - address space differs
  - requires more portion of kernel address space to be WX
- simulate the execution of the instruction is hard
  - literally embedding half a qemu in kernel
  - with more handling related to exceptions
- or we can limit the type of instructions we can probe
  - those with no side effects
  - those only make use of registers/immediates

Kprobes - implementation notes

- what interests us most
  - function entry points (namely, the prologue)
  - function returns
- what is the first instruction in the prologue
  - expanding the stack
  - addi
  - c.addi
  - c.addi16sp
- how to redirect the control flow on function return
  - change ra on function entry

Kprobes - instruction decode

- normal or compressed instructions
    - let length = if inst[0] & 0b11 == 0b11 { 4 } else { 2 };
- obscure encoding of immediates (for software implementations)
    - c.addi16sp
    - 011 imm[9] 00010 imm[4|6|8:7|5] 01

Kprobes - register/unregister

- merely copying bytes
- require fence.i

Kprobes - trap handler

- (probe.handler)(cx); // execute user defined handler with trapframe context
- cx.general.sp = cx.general.sp.wrapping_add(probe.addisp); // simulate inst
- cx.sepc = cx.sepc.wrapping_add(probe.length); // skip the probed instruction

eBPF - with Kprobes

- replace statically compiled handlers (usually shipped with kernel modules) with eBPF programs
- hook eBPF programs into critical parts of kernel to monitor or modify it’s behavior, and even replace parts of kernel
- for a deeper dive into eBPF, see eBPF in rCore

in practise

- the address to hook (from function names)
  - rust symbol name mangling scheme is not meant for readability
  - with a RFC improving the situation [rust-symbol-name-mangling-v0]
  - support by most upstream tools, but not enabled by default on build
- async functions
  - async functions are not functions, but ast level constructs
  - transformed by compiler into Generator that implements Future
  - we can probe on the poll method of the Future
- stack and register informations
  - gdb info scope
  - objdump -d

current progress

- fully featured eBPF interpreter
- in-depth documents on the eBPF ISA
- robust kprobes framework on rCore, with support for SMP
- an eBPF verifier using abstract interpretation to be bridged from c++
- automatic generation of symbol table/stack layout/register allocation infomation for writing portable eBPF programs in progress

demo - verification of termination

good:

```
        for (int i = 0; i < 100; i++)
```

bad:

```
        for (int i = 0; 1; i++)
```

demo - locate probe target

```
nm -g kernel/target/riscv64/debug/rcore | rustfilt | rg ebpf
```

- list all external symbols
- demangle names
- find symbols related to eBPF

demo - extract information

gdb kernel/target/riscv64/debug/rcore

- info scope // for a overview
- disassemble // to track registers
- pahole // to understand struct layout

demo - write the eBPF program

- the TrapFrame is passed as the first argument
- call to helper functions can be written as function pointers