eBPF in rCore

for observability with lower costs

eBPF

run user provided code inside the kernel

- security: a verifier validates that the program does not crash or otherwise harm the system, and it runs to completion within acceptable time limit
- performance: a JIT compilation step translates the generic bytecode of the program into the machine specific instruction set

eBPF VM

a general purpose RISC instruction set with fixed length instruction encoding

eleven 64 bit registers with 32 bit subregisters:

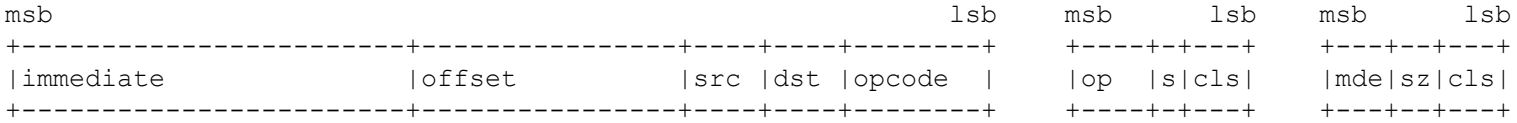
r0-r9 are general purpose, while r10 is read-only frame pointer

calling convention: r1-r5 for arguments, r0 for return value, r6-r9 are callee saved

a fixed size stack of 512 bytes

arbitrary memory access (with checks) thoughout the kernel’s address space

eBPF instructions



- 32 bit signed imm, 16 bit signed offset, 4 bit src reg, 4 bit dst reg, 8 bit opcode
- byteorder is not part of the spec, usually the same as host byteorder
- instructions are grouped in to classes (cls) having different encoding for opcode
- operation (op) and source (s, register or imm) for ALU and jump instructions
- mode (mde) and size (sz) for load and store instructions

eBPF instructions

- ALU instructions are boring, the typical ADD, SUB, MUL and more
- apart from several non generic instructions for accessing packet data, load and store instructions differs only on the size of operation: byte, half word, word or double word
- jump instructions are interesting, they can only perform relative jumps with offset encoded in immediate, which makes ebpf programs naturally relocatable and eaiser to statically analyze

eBPF call instruction

call is for function call

unlike other jump instructions, call is speciall, it calls *helper* functions provided by eBPF runtime by an index encoded in imm

for example:

```
long bpf_trace_printk(const char *fmt, u32 fmt_size, ...)
```

can be called by “call 6”

eBPF maps

with the help of helper functions, the 512 bytes stack and the lack of heap no longer hinders the usability of eBPF programs, as the eBPF runtime can provide complex and persistent data structures in the form of helper functions

namely, maps

```
void *bpf_map_lookup_elem(struct bpf_map *map, const void *key)

long bpf_map_update_elem(struct bpf_map *map, const void *key, const void *value, u64 flag)

long bpf_map_delete_elem(struct bpf_map *map, const void *key)
```

eBPF program

an eBPF program is called by the kernel, just like it calls helper functions, following the same calling convention

as there is a limit of five arguments in function calls, the current design is to pass all arguments as a struct, passing the struct pointer as the first argument

which leaves us with the following function signature

```
long program(void *context)
```

eBPF hooks

eBPF programs are event-driven, and get called when kernel passes certain hook points. within linux, there are many pre-defined hooks including system calls, function entry/exit, network events, and several others

still, we can hook nearly any memory address with a framework called kprobe

kprobe

kprobe operates by replacing the first byte of the hooked instruction to int3, and registering an interrupt handler for breakpoints

after gaining access to the the control flow, kprobe calls user defined functions, eBPF in our case, single steps the replaced instruction, and then brings the execution back to normal

design goals

- a verifier and interpreter for eBPF in rust
- a JIT compiler targeting RISC-V
- a kprobe like framework for rCore
- a wide range of helper functions
- evaluation the performance in syscall tracing

prior art

rbpf and ubpf

- mostly identical, one in rust, one in c
- intented for use in userland
- partial and non-compliant implementaion of the spec
- JIT targeting only x86_64
- no longer actively maintained

current progress

- interpreter: mostly ok
- verifier and JIT: absent
- kprobe: static probes (can be disarmed)
- helper functions: only printk

future plan

- the priority is on kprobe as it’s applicable to fields beyond eBPF
- verifier might be stricter than it should be as the halting problem is undecidable
- JIT for all architectures supported by rCore
- provide helper functions in the form of eBPF programs

demo

- eBPF programs can be written in restricted C, and compiled with clang

  clang -target bpf -O2

- kprobe hook
- breakpoint handler
- ebpf program