Stories from BPF security auditing at Google

Brendan Jackman

Agenda

- History/refresher: KRSI
- BPF Atomics
- Ringbuffers
- What’s next?

Refresher: KRSI

Detection & Response w/ Telemetry

Various
Linux
Computers

(**Google**'s machines - not your Chromebook!)

Clever
Software
Pipeline

Clever
Security
Engineers

Security telemetry on Linux: our journey

- Audit is not flexible or fast enough
- Kernel module was awful to maintain
- Turned to BPF, but we often struggled to find simple places to attach our programs
- The **BPF LSM** was born.
- LSMs get a **semantic** (internal) API for security information
- Designed for enforcement, and now we use them for audit too.

BPF Atomics

BPF programs are concurrent

So how do you generate a globally-unique integer?

per-CPU arrays...

`bpf_spin_lock`...

At the BPF office hours...

`drake_no.png`

Atomics helpers?

`drake_yes.png`

Atomics instructions!

+	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x00					ALU_ADD_K	JMP_JA		ALU64_ADD_K
0x08					ALU_ADD_X			ALU64_ADD_X
0x10					ALU_SUB_K	JMP_JEQ_K	JMP32_JEQ_K	ALU64_SUB_K
0x18	LD_IMM_DW				ALU_SUB_X	JMP_JEQ_X	JMP32_JEQ_X	ALU64_SUB_X
0x20	LD_ABS_W*	LDX_PROBE_MEM_W*			ALU_MUL_K	JMP_JGT_K	JMP32_JGT_K	ALU64_MUL_K
0x28	LD_ABS_H*	LDX_PROBE_MEM_H*			ALU_MUL_X	JMP_JGT_X	JMP32_JGT_X	ALU64_MUL_X
0x30	LD_ABS_B*	LDX_PROBE_MEM_B*			ALU_DIV_K	JMP_JGE_K	JMP32_JGE_K	ALU64_DIV_K
0x38		LDX_PROBE_MEM_DW*			ALU_DIV_X	JMP_JGE_X	JMP32_JGE_X	ALU64_DIV_X
0x40	LD_IND_W*				ALU_OR_K	JMP_JSET_K	JMP32_JSET_K	ALU64_OR_K
0x48	LD_IND_H*				ALU_OR_X	JMP_JSET_X	JMP32_JSET_X	ALU64_OR_X
0x50	LD_IND_B*				ALU_AND_K	JMP_JNE_K	JMP32_JNE_K	ALU64_AND_K
0x58					ALU_AND_X	JMP_JNE_X	JMP32_JNE_X	ALU64_AND_X
0x60		LDX_MEM_W	ST_MEM_W	STX_MEM_W	ALU_LSH_K	JMP_JSGT_K	JMP32_JSGT_K	ALU64_LSH_K
0x68		LDX_MEM_H	ST_MEM_H	STX_MEM_H	ALU_LSH_X	JMP_JSGT_X	JMP32_JSGT_X	ALU64_LSH_X
0x70		LDX_MEM_B	ST_MEM_B	STX_MEM_B	ALU_RSH_K	JMP_JSGE_K	JMP32_JSGE_K	ALU64_RSH_K
0x78		LDX_MEM_DW	ST_MEM_DW	STX_MEM_DW	ALU_RSH_X	JMP_JSGE_X	JMP32_JSGE_X	ALU64_RSH_X
0x80	👀				ALU_NEG	JMP_CALL		ALU64_NEG
0x88	👀							
0x90	👀				ALU_MOD_K	JMP_EXIT		ALU64_MOD_K
0x98	👀				ALU_MOD_X			ALU64_MOD_X
0xa0	👀				ALU_XOR_K	JMP_JLT_K	JMP32_JLT_K	ALU64_XOR_K
0xa8	👀				ALU_XOR_X	JMP_JLT_X	JMP32_JLT_X	ALU64_XOR_X
0xb0	👀				ALU_MOV_K	JMP_JLE_K	JMP32_JLE_K	ALU64_MOV_K
0xb8	👀				ALU_MOV_X	JMP_JLE_X	JMP32_JLE_X	ALU64_MOV_X
0xc0	👀			STX_XADD_W	ALU_ARSH_K	JMP_JSLT_K	JMP32_JSLT_K	ALU64_ARSH_K
0xc8	👀				ALU_ARSH_X	JMP_JSLT_X	JMP32_JSLT_X	ALU64_ARSH_X
0xd0	👀				ALU_END_TO_LE	JMP_JSLE_K	JMP32_JSLE_K	
0xd8	👀			STX_XADD_DW	ALU_END_TO_BE	JMP_JSLE_X	JMP32_JSLE_X	
0xe0	👀					JMP_CALL_ARGS*		
0xe8	👀							
0xf0	👀					JMP_TAIL_CALL*		
0xf8	👀							

+	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x00					ALU_ADD_K	JMP_JA		ALU64_ADD_K
0x08					ALU_ADD_X			ALU64_ADD_X
0x10					ALU_SUB_K	JMP_JEQ_K	JMP32_JEQ_K	ALU64_SUB_K
0x18	LD_IMM_DW				ALU_SUB_X	JMP_JEQ_X	JMP32_JEQ_X	ALU64_SUB_X
0x20	LD_ABS_W*	LDX_PROBE_MEM_W*			ALU_MUL_K	JMP_JGT_K	JMP32_JGT_K	ALU64_MUL_K
0x28	LD_ABS_H*	LDX_PROBE_MEM_H						
0x30	LD_ABS_B*	LDX_PROBE_MEM_B						
0x38		LDX_PROBE_MEM_D						
0x40	LD_IND_W*							
0x48	LD_IND_H*							
0x50	LD_IND_B*							
0x58								
0x60		LDX_MEM_W	ST_M					
0x68		LDX_MEM_H	ST_M					
0x70		LDX_MEM_B	ST_MEM_B	STX_MEM_B	ALU_RSH_K	JMP_JSGE_K	JMP32_JSGE_K	ALU64_RSH_K
0x78		LDX_MEM_DW	ST_MEM_DW	STX_MEM_DW	ALU_RSH_X	JMP_JSGE_X	JMP32_JSGE_X	ALU64_RSH_X
0x80					ALU_NEG	JMP_CALL		ALU64_NEG
0x88								
0x90					ALU_MOD_K	JMP_EXIT		ALU64_MOD_K
0x98					ALU_MOD_X			ALU64_MOD_X
0xa0					ALU_XOR_K	JMP_JLT_K	JMP32_JLT_K	ALU64_XOR_K
0xa8					ALU_XOR_X	JMP_JLT_X	JMP32_JLT_X	ALU64_XOR_X
0xb0					ALU_MOV_K	JMP_JLE_K	JMP32_JLE_K	ALU64_MOV_K
0xb8					ALU_MOV_X	JMP_JLE_X	JMP32_JLE_X	ALU64_MOV_X
0xc0				STX_XADD_W👀	ALU_ARSH_K	JMP_JSLT_K	JMP32_JSLT_K	ALU64_ARSH_K
0xc8					ALU_ARSH_X	JMP_JSLT_X	JMP32_JSLT_X	ALU64_ARSH_X
0xd0					ALU_END_TO_LE	JMP_JSLE_K	JMP32_JSLE_K	
0xd8				STX_XADD_DW	ALU_END_TO_BE	JMP_JSLE_X	JMP32_JSLE_X	
0xe0						JMP_CALL_ARGS*		
0xe8								
0xf0						JMP_TAIL_CALL*		
0xf8								

```
struct bpf_insn {
        __u8    code;           /* opcode */
        __u8    dst_reg:4;      /* dest register */
        __u8    src_reg:4;      /* source register */
        __s16   off;            /* signed offset */
        __s32   imm;            /* signed immediate constant */ 👀
};
```

Proprietary + Confidential

Old representation

```
struct bpf_insn i = {
      .code = BPF_STX | BPF_XADD | BPF_DW,
      .imm = 0, // otherwise verifier rejects insn
      .dst_reg = BPF_REG_0,
      .src_reg = BPF_REG_1,
}
```

New representation

```
struct bpf_insn i = {
      .code = BPF_STX | BPF_ATOMIC | BPF_DW,
      .imm = BPF_ADD,
      .dst_reg = BPF_REG_0,
      .src_reg = BPF_REG_1,
}
```

Same bit-representation!

New instructions

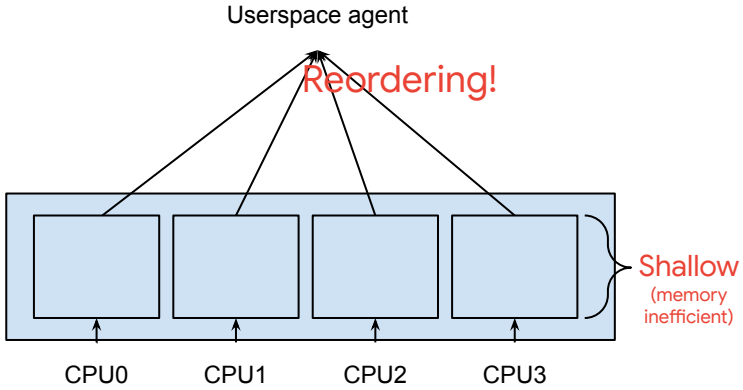
```
struct bpf_insn i = {
      .code = BPF_STX | BPF_ATOMIC | BPF_DW,
      .imm = BPF_ADD | BPF_FETCH,
      .dst_reg = BPF_REG_0,
      .src_reg = BPF_REG_1,
}
```

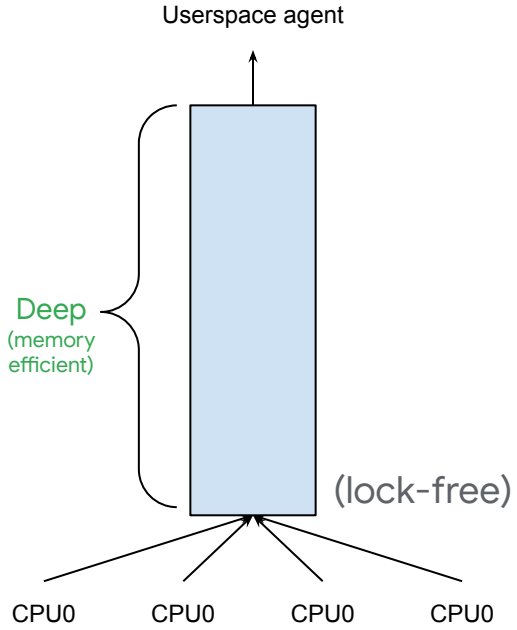
```
struct bpf_insn i = {
      .code = BPF_STX | BPF_ATOMIC | BPF_DW,
      .imm = BPF_OR,
      .dst_reg = BPF_REG_0,
      .src_reg = BPF_REG_1,
}
```

```
struct bpf_insn i =
      .code = BPF_ST
      .imm = BPF_XOR
      .dst_reg = BPF
      .src_reg = BPF
}
```

Ringbuffers

Ring buffers: perf buffer vs BPF ringbuf

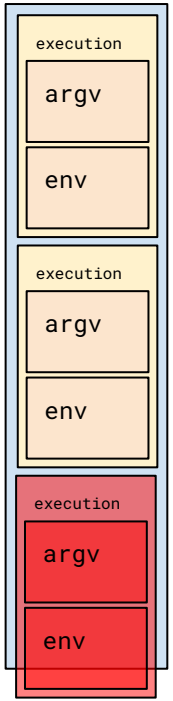




Perf Buffer enforces one-ring-per-CPU

BPF ringbuf gives flexibility to make these tradeoffs as you desire

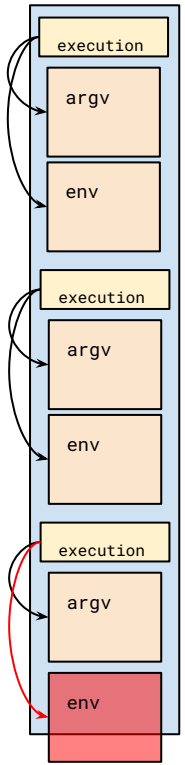
Ring buffers: promises



Outputting all data at once means that all is lost if the

ringbuf is full

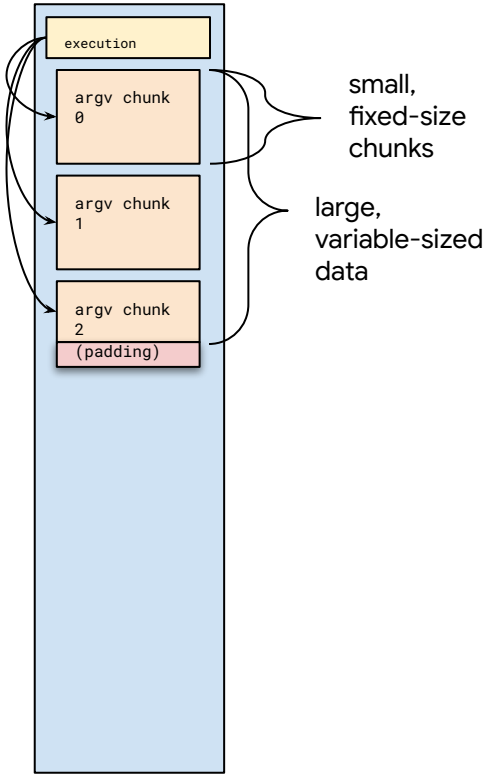
Ring buffers: promises



Promise system:

- Don’t lose the whole event if ringbuf is full
- This also lets us defer producing data until later

Ring buffers: chunking



- Verifier likes to know buffer sizes in advance
- But allocating max-possible size is bad
- Break down large data into fixed-size chunks

BPF Across Multiple Kernel Versions

Kernel Diversity

Various kernel versions... only one userspace binary.

How do we do “feature negotiation?”

If your program uses unsupported features, the verifier rejects it.

Linear program fallback

Most feature-complete version of the prog

Most widely-supported version of the prog

Top tip: field renames

drake_no.png

```
SEC("lsm/something")
int BPF_PROG(something *s)
{
      if (s->new_field > 3)
            return 1;
      return 0;
}
```



```
SEC("lsm/something")
int BPF_PROG(something *s)
{
      if (s->old_field > 3)
            return 1;
      return 0;
}
```

drake_yes.png

```
SEC("lsm/something")
int BPF_PROG(something *s)
{
      int field = 0;

      if (bpf_core_field_exists(s->new_field))
            field = s->new_field;
      else
            field = s->old_field;

      if (s->new_field > 3)
            return 1;
      return 0;
}
```

What’s next?

What's next?

- DNS auditing
- Enforcement with KRSI
- Less kernel implementation details

Thank You