Tracing resource-constrained embedded systems using eBPF

Adrian Ratiu, Collabora Ltd
Agenda

• About me
• Embedded / IoT woes
• How does eBPF fit in?
• Quick eBPF / BCC introduction, benefits
• Approaches to eBPF on embedded devices
• Trade-offs, specific projects pros/cons
• Ways forwards
About me.

I enjoy working in a company of awesome FOSS-oriented people at Collabora work with companies who “get it” when using FOSS work to help companies “get it” and be successful
I also really enjoy

Taking systems apart and modifying them

Projects like OpenEmbedded/Yocto, Buildroot/OpenWRT

Always looking for new tech to improve development and debugging of embedded devices

Learning about eBPF (just a user, not an expert)

A strong dislike of locked-down devices / that lock owner usage without very good reasons
Embedded and the IoT

- “Smart” devices everywhere
- Increasingly powerful, complex, connected hardware
- Much more capable than default software installations allow
- Software complexity is also rising
  (embedded systems now programmed in JavaScript)
- Obvious privacy, security and vendor lock-in concerns
Embedded problems

Devices are more powerful & run modern SW stacks yet they are still very hard (harder?) to develop, debug, maintain and extend
Embedded problems

Why?
Embedded problems

Why?

Increased SW/HW complexity
+
embedded-specific
resource constraints trade-offs
Resource constraints / trade-offs

- Enough memory to run just a specific pre-built workload
- Cross-compiling and flashing/provisioning
- Special “Embedded Linux” distributions
- Boot time or RT deadline requirements
- Ergonomics trade-offs, lack of HW ports
- Licensing requirements (no GPLv3...)
- Weird HW combinations, countless HW revisions
- Throw-away HW, planned obsolescence
- Low quality Out-Of-Tree drivers, non-discoverable busses
- <Add your own pet-peeve here>
Creative solutions against constraints

- Debug symbol servers and remote GDB sessions
- Booting rootfs over the network
- Special protocols for diagnostics/log/trace
- Debug vs Release images, “developer mode”
- Random hacks like not loading video drivers to preserve splash screen
Creative solutions against constraints

- Debug symbol servers and remote GDB sessions
- Booting rootfs over the network
- Special protocols for diagnostics/log/trace
- Debug vs Release images, “developer mode”
- Random hacks like not loading video drivers to preserve splash screen

Here comes eBPF
Wait a minute

Embedded-eBPF sounds like a solution in search of a problem...
Wait a minute

**Embedded-eBPF sounds like a solution in search of a problem...**

*It kind of is.*

“Embedded” engineers drooling over tools of “Cloud” engineers

Would like to have same system observability powers

Precedent: SMP now used on embedded devices
Explaining eBPF / BCC in a few slides!

BCC automates

VM bytecode

Kernel 🆕 Userspace

Links at the end for better learning resources.
VM running bytecode in the Linux kernel

Bytecode loaded from userspace via bpf() syscall
   Verified for safety, unsafe => syscall rejects bytecode

Bytecode compiled to native machine code

Native code inserted in execution paths
   Event-driven programming

Native code runs and collects data

Data shared with userspace
User process

sys_bpf() load

eBPF Bytecode verifier

Validation successful

JiT compiler
Bytecode -> native code

sys_open handler

sys_open()

Attach/insert code at instruction
How does userspace produce that bytecode?

0:  79 12 60 00 00 00 00 00    r2 = *(u64 *)(r1 + 96)
1:  7b 2a 98 ff 00 00 00 00    *(u64 *)(r10 - 104) = r2
2:  79 17 70 00 00 00 00 00    r7 = *(u64 *)(r1 + 112)
3:  85 00 00 00 0e 00 00 00    call 14
4:  bf 06 00 00 00 00 00 00    r6 = r0
5:  b7 09 00 00 00 00 00 00    r9 = 0
6:  7b 9a c0 ff 00 00 00 00    *(u64 *)(r10 - 64) = r9
7:  bf 73 00 00 00 00 00 00    r3 = r7
8:  07 03 00 00 18 00 00 00    r3 += 24
9:  bf a1 00 00 00 00 00 00    r1 = r10
11: 07 01 00 00 c0 ff ff ff    r1 += -64
12: b7 02 00 00 08 00 00 00    r2 = 8
13: 85 00 00 00 04 00 00 00    call 4
How does userspace produce that bytecode?

Directly write it byte by byte!

0:  79 12 60 00 00 00 00 00    r2 = *(u64 *)(r1 + 96)
1:  7b 2a 98 ff 00 00 00 00    *(u64 *)(r10 - 104) = r2
2:  79 17 70 00 00 00 00 00    r7 = *(u64 *)(r1 + 112)
3:  85 00 00 00 0e 00 00 00    call 14
4:  bf 06 00 00 00 00 00 00    r6 = r0
5:  b7 09 00 00 00 00 00 00    r9 = 0
6:  7b 9a c0 ff 00 00 00 00    *(u64 *)(r10 - 64) = r9
7:  bf 73 00 00 00 00 00 00    r3 = r7
8:  07 03 00 00 18 00 00 00    r3 += 24
9:  bf a1 00 00 00 00 00 00    r1 = r10
11: 07 01 00 00 c0 ff ff ff    r1 += -64
12: b7 02 00 00 08 00 00 00    r2 = 8
13: 85 00 00 00 04 00 00 00    call 4
Clang can translate “restricted C” into eBPF bytecode. Much easier than assembling bytes like the 1960s.

Still hard to write userspace interaction.
Clang can translate “restricted C” into eBPF bytecode
Much easier than assembling bytes like the 1960s

**Still hard to write userspace interaction**

**BCC: the BPF Compiler Collection**

Framework to ease writing userspace eBPF programs
Abstracts Clang and sys_bpf() interaction
“restricted C” compiled & loaded in kernel on-the-fly
Provides Python, Lua and Go bindings
Provides production ready BCC-tools
#!/usr/bin/env python
from bcc import BPF

csrc = ""
#include <uapi/linux/ptrace.h>

int kprobe__do_sys_open(struct pt_regs *ctx)
{
    char file_name[256];
    bpf_probe_read(&file_name, sizeof(file_name), PT_REGS_PARM1(ctx));
    bpf_trace_printk(fmt, sizeof(fmt), file_name);
}
"

b = BPF(text=csrc)
b.attach_kprobe(event="do_sys_open", fn_name="kprobe__do_sys_open")

while True:
    time.sleep(1)
BCC program

```python
#!/usr/bin/env python
from bcc import BPF

csrc = ""
#include <uapi/linux/ptrace.h>

int kprobe__do_sys_open(struct pt_regs *ctx)
{
    char file_name[256];
    bpf_probe_read(&file_name, sizeof(file_name), PT_REGS_PARM1(ctx));
    bpf_trace_printk(fmt, sizeof(fmt), file_name);

    return 0;
}

b = BPF(text=csrc)
b.attach_kprobe(event="do_sys_open", fn_name="kprobe__do_sys_open")
while True:
    time.sleep(1)
```
Real power comes with the BCC tools
eBPF meets embedded

general / embedded-specific problems

multiple approaches

advantages / disadvantages

trade-offs, no silver bullet
General problem: portability / cross-compilation

Poking “outside” from the eBPF VM into rest of the system

- VM has generic 64 bit instructions/registers/pointers
- Difficulty accessing 32 bit kernel/user data structures
- VM is capable of 32 bit register subaddressing
- Pointer arithmetic hacks can access 32bit offset data

*Very fragile, not portable*

- Better solution: BPF Type Format adds type info to compiled eBPF
  (part of C.O.R.E.)
General problem: portability / cross-compilation

Portable eBPF (Compile Once, Run Everywhere)

• Dream: run precompiled eBPF an any machine and expect it to work

• Slimmer version of BCC using BTF info, no Clang runtime compilation
  (structure offsets built in BTF sections, macro identifiers → BPF variables)

• Current runtime compilation uses version/config specific C headers
  • Backwards, not forwards compatible
  • Manually copying non-UAPI structures to “restricted C”
  • Big variation of Linux kernel configs → header structures

• Kernel >= 5.2 can remove header filesystem dependency (kinda unrelated)
• Work on-going, significant work merged for v5.4 and v5.5
General problem: Standardization

• No reusable standard library of eBPF / C code
• Each program writes its own eBPF from scratch
  (most of these code snippets are small)
• Majority of tools built around Clang
• GCC support is very new, sees little adoption
  (ELF format dictated by whatever Clang produces / kernel accepts)
• Portability (CORE) needed before standardization
General problem: Security and unprivileged eBPF

Running eBPF programs requires root / CAP_SYS_ADMIN

- eBPF code is assumed not malicious
- CAP_BPF will be added to restrict attack surface
- Unprivileged eBPF unlikely to happen

Care must be taken when running eBPF code in production

- Don’t run arbitrary eBPF supplied by untrusted users
- Use additional security mechanisms like apparmor, verified boot

Awesome (as always) relevant LWN.net article and comments:
  [https://lwn.net/Articles/796328/](https://lwn.net/Articles/796328/)
Special problem: Real Time Linux and eBPF

Using eBPF on RT systems can cause latency spikes
  • EBPF unconditionally disables preemption
  • Can add up to 500 us delay depending on workload

Currently PREEMPT_RT is made incompatible with eBPF via config

A proper solution should be possible in the future (TODO, WIP)

Awesome (as always) relevant LWN.net article and comments:
https://lwn.net/Articles/802884/
Approach 1: Precompiled eBPF + custom userspace

**PRO:**
- Lightest footprint possible (few kb C program)
- Kernel provides helper libbpf (useful starting point)

**CON:**
- Need to write from scratch
- Userspace sys_bpf() interaction
- Can get complex, hard to maintain
- No pre-existing community

Some examples provided by Linux kernel tree in samples/bpf/
Approach 2: Use BCC directly

**PRO:**
- Vanilla upstream BCC
- Full framework capabilities
- All BCC-tools available
- Well tested, good performance

**CON:**
- Installs and links against Clang
- Depends on Python (bcc-tools)
- ~ 300 MB storage

Will benefit from C.O.R.E., but will still require python

Example project: Androdeb
(Requires > 2GB storage)
**Approach 3: BPFd**

**PRO:**
- 100 kb bin + libc dependency
- Full framework capabilities
- All BCC-tools available

**CON:**
- Hard to maintain BCC<>BPFd interaction
- Host + target + transport architecture, not great security
- Non-trivial latency

---

Project abandoned due to high maintenance cost
**Approach 4: DSL compiler from scratch - Ply**

**PRO:**
- 50 kb bin + libc dependency
- High level, AWK-inspired DSL
- Self-contained
- Easy to build & deploy

**CON:**
- Lack of kernel/user interaction control
- Lack of BCC-tools diversity
- Under heavy development
- Ply binary is not portable

```plaintext
ply 'kprobe:i2c_transfer { print(stack); }'
```
Approach 5: Replace BCC Python userspace with Go

**PRO:**
- ~2 mb static-compiled eBPF loader
- Full control over kernel/user interaction
- Good coverage of BCC API bindings

**CON:**
- BCC-tools need rewriting in Go :)
- Not much documentation

---

Full execsnoop reimplementation:

https://github.com/iovisor/gobpf/blob/master/examples/bcc/execsnoop/execsnoop.go
Ways forward

- C.O.R.E. needs to be as successful as possible
  (Lighter BCC + portable eBPF)
- Special-purpose projects can continue shipping precompiled eBPF (Approach 1)
- Gobpf can eliminate the Python dependency (Rust?)
- BPFd reached a dead end
- Ply is standalone, will continue its awesomeness
- eBPF on embedded is already quite useful today
- Much work remaining
  (RT reconciliation, eBPF libraries, GCC addition + ELF std)
Recommended learning resources:

• LWN.net eBPF articles https://lwn.net/

• Brendan Gregg’s blog: http://www.brendangregg.com/blog/

• BPF Performance Tools: Linux System and Application Observability, by Brendan Gregg, published by Addison Wesley (2019)

• Collabora eBPF blog posts


• Internet Search has wealth of information on eBPF
Thank you!