C++ Summit 2020

Linux内核的语言虚拟机王者-EBPF

冯富秋

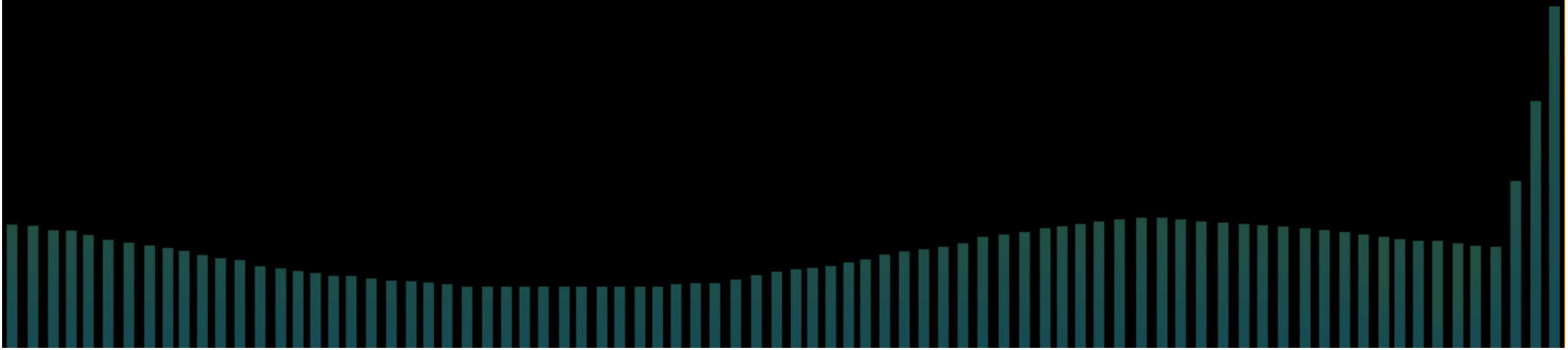
阿里云智能系统技术负责人

致力于阿里超大规模数据中心的稳定性和可靠性建设

Today's agenda

1. Introduction
2. How to do tracing in ultra-large-scale data centers
3. cBPF Introduction, History & Implementation
4. eBPF Introduction, History & Implementation
5. eBPF Tracing and Monitoring
6. Alibaba Tracing and Monitoring SIG

1 Introduction



$ WHOAMI

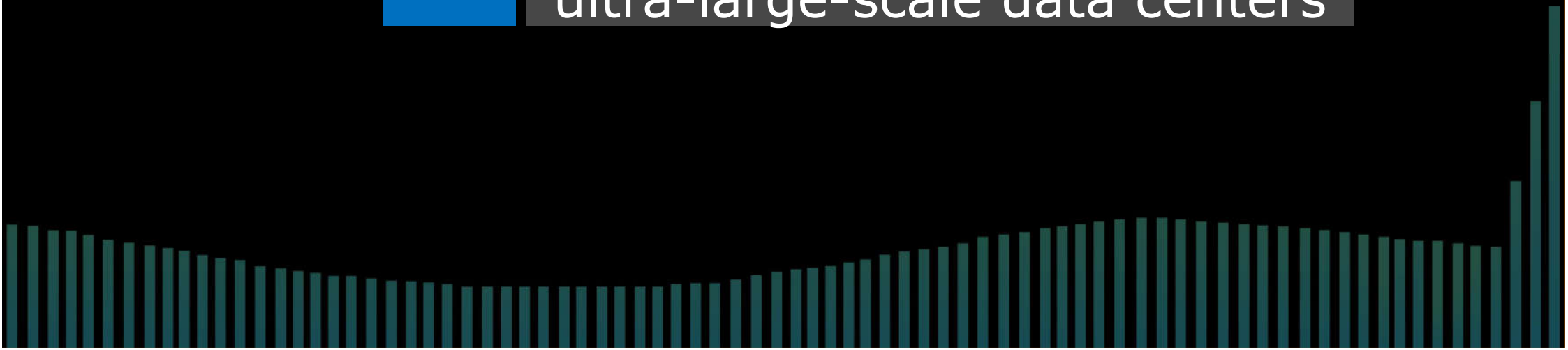
冯富秋（Tinnal）

- Senior Technical Expert， Team Leader @ Alibaba
- System Technology Team

- What I do:
  - Build customer competitiveness by OS
  - Stability and reliability construction of ultra-large-scale data centers。

2

How to do Tracing/Monitoring in ultra-large-scale data centers



2.1 How to do tracing in ultra-large-scale data centers

Runtime environment of ultra-large-scale data centers

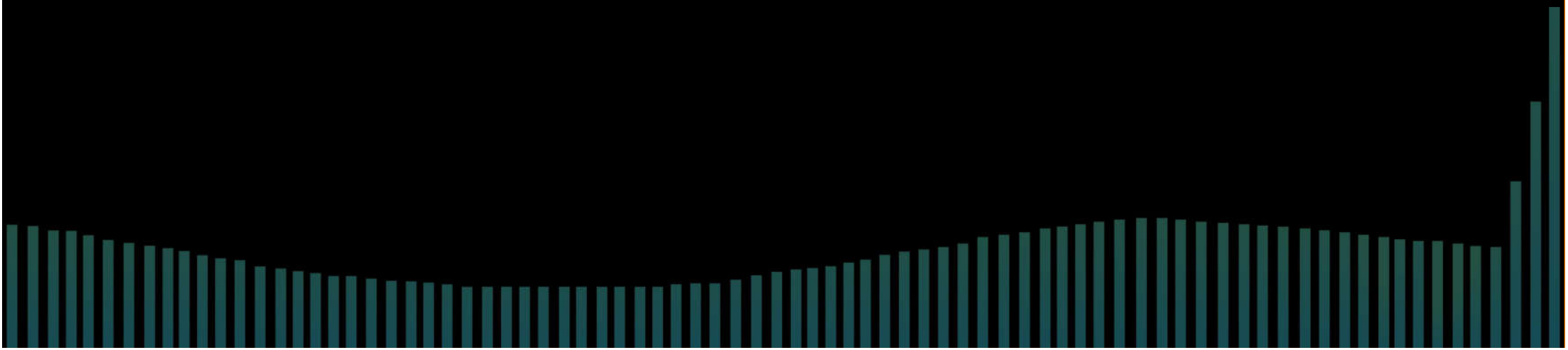
- Heterogeneous computer architecture
- Ultra high reliability requirements
- Must have minimal overhead
- Static mix dynamic tracing/monitoring instrumentation

Tracer/Front-end	Data sources	Data collection
Ftrace	kprobe, uprobe, tracepoints, USDT	ftrace (sysfs)
Perf	perf_events (+kprobes, tracepoints...) kprobe,	perf ring buffer
SystemTap	uprobe, tracepoints, USDT Specific events, or	kernel module
LTTng	user space tracing syscalls (not sure how)	kernel module
Sysdig	DTrace probes... but not on Linux!	Sysdig ring buffer
DTrace		

Limitations:

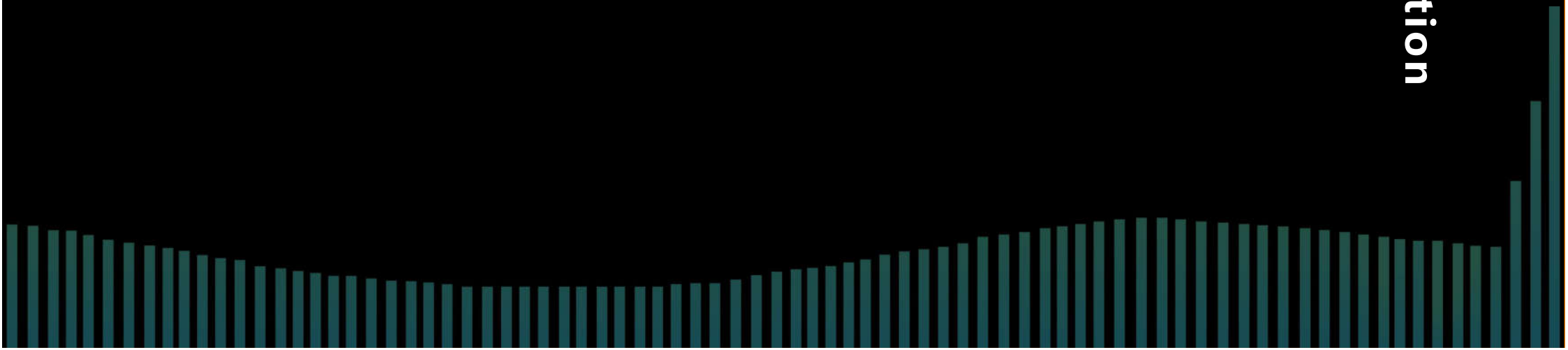
- SystemTap and LTTng require building and inserting kernel modules
- ftrace, perf, LTTng lack programmability
- SystemTap not user friendly, could crash the kernel
- Sysdig limited to system calls
- DTrace very powerful but not in Linux kernel (Some out-of-tree ports available)

3 cBPF Introduction, History & Implementation



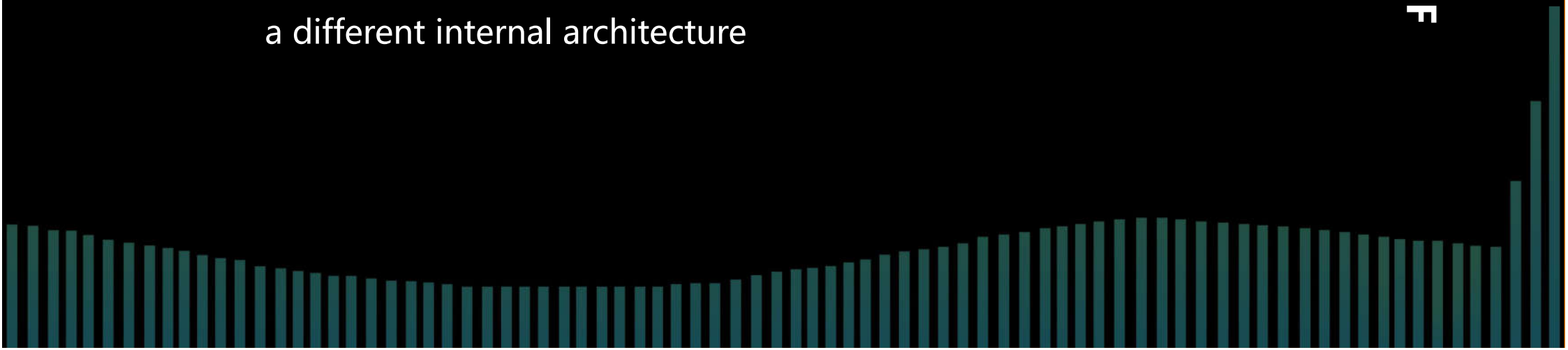
What is cBPF?

- cBPF – Classic BPF
  - Also known as “Linux Packet Filtering”
- Network packet filtering, Seccomp
- Filter Expression→Bytecode→Interpret
- Small, in-kernel VM, Register based,few instructions

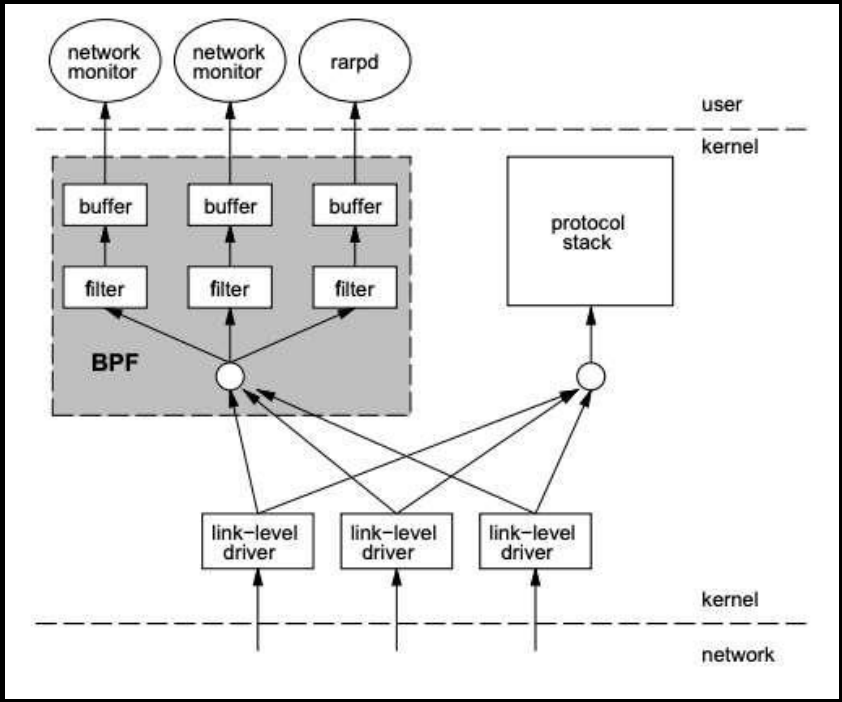


3.2 History of BPF

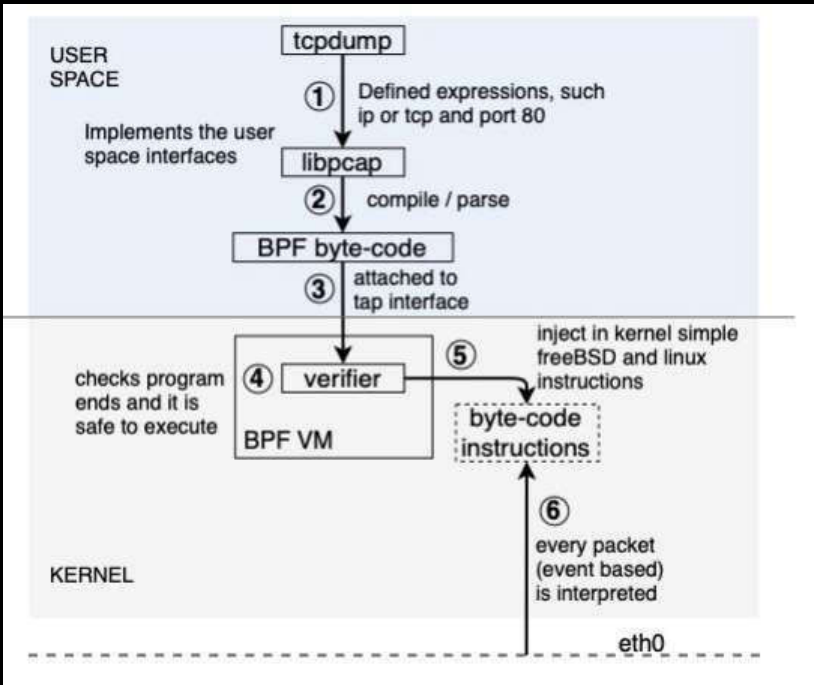
- Before BPF, each OS (Sun, DEC, SGI etc) had its own packet filtering API
- In 1993: Steven McCanne & Van Jacobsen released a paper titled the *BSD Packet Filter (BPF)*
- In 1997: Implemented as “Linux Socket Filter” in kernel 2.1.75
- While maintaining the BPF language (for describing filters), uses a different internal architecture



3.3 BPF implementation



https://www.tcpdump.org/papers/bpf-usenix93.pdf



https://static.sched.com/hosted_files/kccnceu19/b8/KubeCon-Europe-2019-Beatriz_Martinez_eBPF.pdf

3.3 BPF implementation

Structure

```
struct sock_filter { /* Filter block */
__u16 code; /* Actual filter code */
__u8 jt; /* Jump true */
__u8 jf; /* Jump false */
__u32 k; /* Generic multiuse field */ };
```

characteristic

- two 32-bit registers: A, X
- 16 * 32-bit slots stack
- full integer arithmetic
- explicit load/store from packet
- conditional branches (with two destinations: jump true/false)

3.3 BPF implementation

Instruction

Inst.	AddrMode	Description
ld	1, 2, 3, 4, 12	Load word into A
<…>		
ldx	3, 4, 5, 12	Load word into X
<…>		
st	3	Store A into M[]
stx	3	Store X into M[]
jmp	6	Jump to label
ja	6	Jump to label
jeq	7, 8, 9, 10	Jump on A == <x>
<…>		
add	0, 4	A + <x>
sub	0, 4	A - <x>
<…>		
tax		Copy A into X
txa		Copy X into A
ret	4, 11	Return

Addressing mode

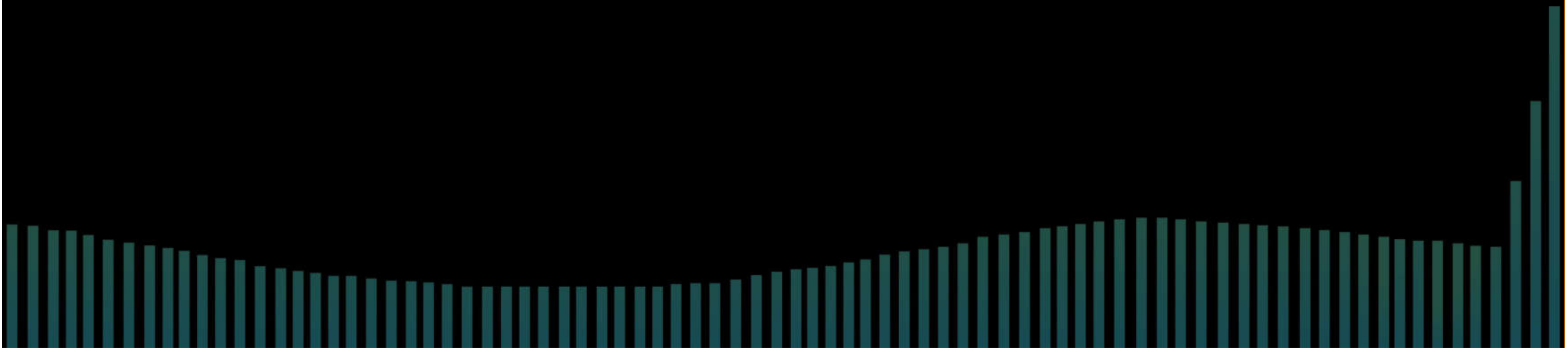
AddrMode	Syntax	Description
0	x/%x	Register X
1	[k]	BHW at byte offset k in the packet
2	[x + k]	BHW at the offset X + k in the packet
3	M[k]	Word at offset k in M[]
4	#k	Literal value stored in k
5	4*([k]&0xf)	Lower nibble * 4 at byte offset k in the packet
6	L	Jump label L
7	#k,Lt,Lf	Jump to Lt if true, otherwise jump to Lf
8	x/%x,Lt,Lf	Jump to Lt if true, otherwise jump to Lf
9	#k,Lt	Jump to Lt if predicate is true
10	x/%x,Lt	Jump to Lt if predicate is true
11	a/%a	Accumulator A
12	extension	BPF extension

3.3 BPF implementation

```
# tcpdump -i eth0 tcp dst port 22 –d

(000) ldh [12]                  # Ethertype
(001) jeq #0x86dd jt 2 jf 6     # is IPv6?
(002) ldb [20]                  # IPv6 next header field
(003) jeq #0x6 jt 4 jf 15       # is TCP?
(004) ldh [56]                  # TCP dst port
(005) jeq #0x16 jt 14 jf 15     # is port 22?
(006) jeq #0x800 jt 7 jf 15     # is IPv4?
(007) ldb [23]                  # IPv4 protocol field
(008) jeq #0x6 jt 9 jf 15       # is TCP?
(009) ldh [20]                  # IPv4 flags + frag. offset
(010) jset #0x1fff jt 15 jf 11  # fragment offset is != 0?
(011) ldxb 4*([14]&0xf)         # x := 4 * header_length (words)
(012) ldh [x + 16]              # TCP dst port
(013) jeq #0x16 jt 14 jf 15     # is port 22?
(014) ret #262144               # trim to 262144 bytes, return packet
(015) ret #0                     # drop packet
```

4 eBPF Introduction, History & Implementation



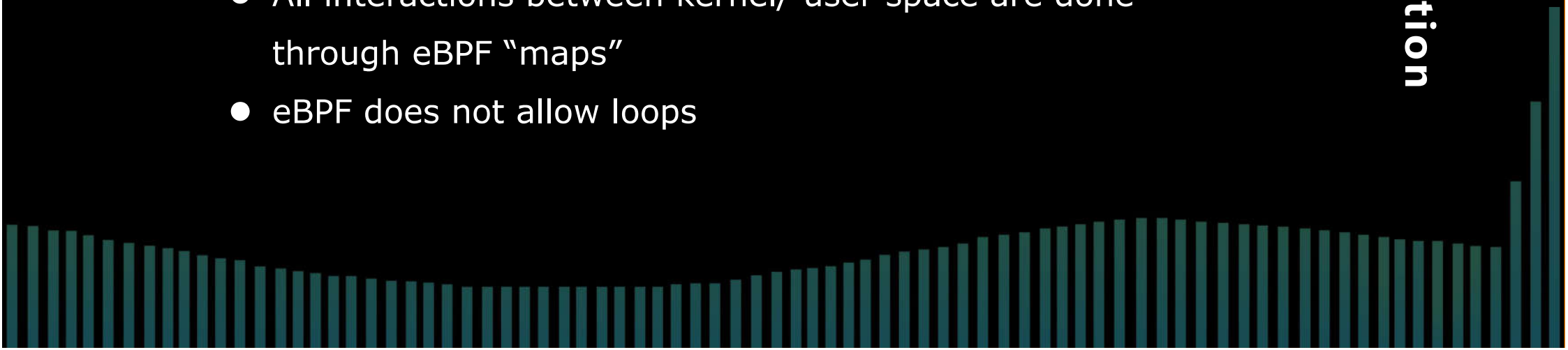
"crazy stuff"
-Alexei Starovoitov (eBPF lead)

"eBPF is Linux's new superpower"
- Gaurav Gupta

"eBPF does to Linux what JavaScript does to HTML"
— **Brendan Gregg**

What is eBPF?

- eBPF – extended Berkeley Packet Filter
- User-defined, sandboxed bytecode executed by the kernel
- VM that implements a RISC-like assembly language in kernel space
- All interactions between kernel/ user space are done through eBPF “maps”
- eBPF does not allow loops



What can BPF be used for

NETWORKING

- Load-balancing
  - Katran(Facebook)
- General networking
  - Cilium
- Extending the TCP stack
- Network Monitoring
  - Flowmill
  - Weaveworks

FIREWALLS

Bpfilter (Linux 4.18)

DDOS MITIGATION

Use of eBPF & XDP to perform infra-wide DDoS mitigation

- Facebook
- Cloudflare

PROFILE >< TRACING

- Sysdig
- bpftrace

CHAOS ENGINEERING

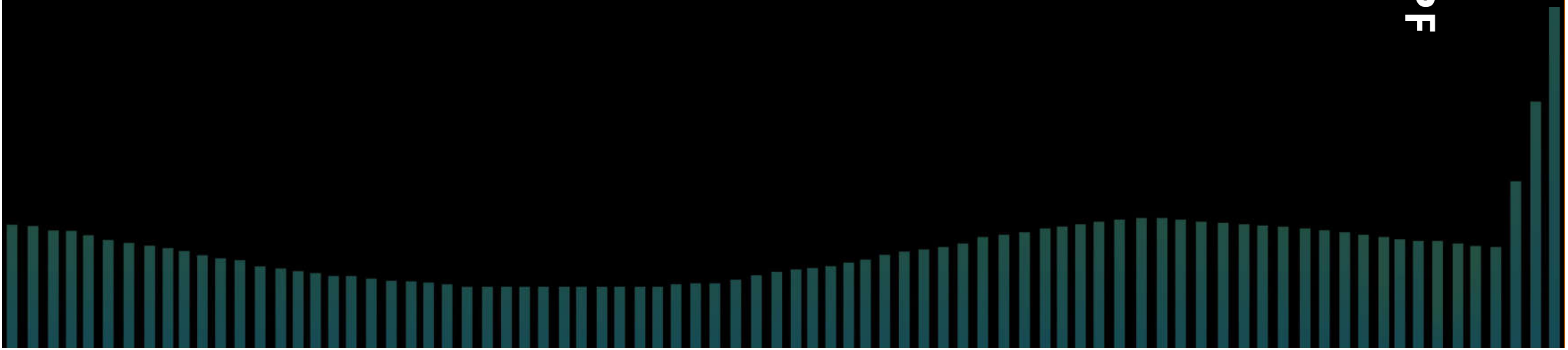
Use Cilium to inject latency, packet-loss, L7 HTTP errors (via a Go extension)

DEVICE DRIVERS

eBPF provides a pseudo device driver possible to extend this in multiple ways

4.2 History of eBPF

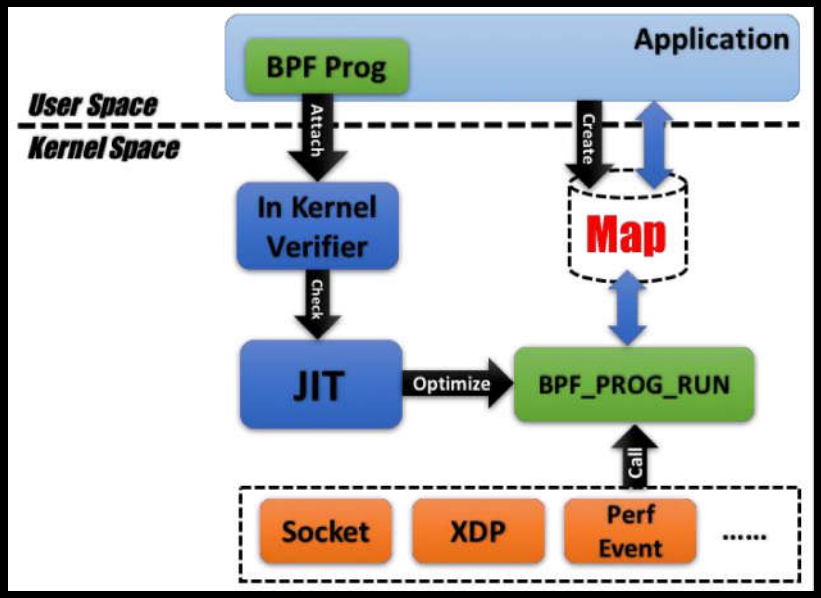
- Development started in 2011 - A JIT for packet filters
- Add tracing filters with BPF in 2013
- eBPF first included with kernel 3.18 in 2014
- BPF backend for LLVM upstreamed in Jan 2015



design goals

- parse, lookup, update, modify network packets
- loadable as kernel modules on demand, on live traffic
- safe on production system
- performance equal to native x86 code
- fast interpreter speed (good performance on all architectures)
- calls into bpf and calls from bpf to kernel should be free (no FFI overhead)

Architecture of eBPF

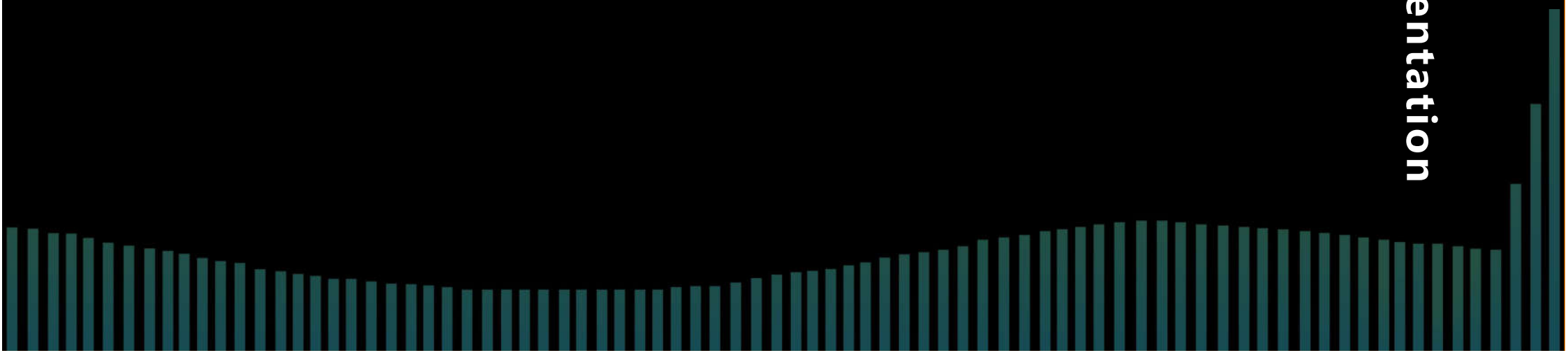


Design Intent

- take a mix of real CPU instructions
  (10% classic BPF + 70% x86 + 25% arm64 + 5% risc)
- analyze x86/arm64/risc calling conventions and define a common one for this ‘renamed’ instruction set
- make instruction encoding fixed size (for high interpreter speed)
- reuse classic BPF instruction encoding (for trivial classic->extended conversion)

Core changes of the new internal format

- Number of registers increase from 2 to 10 (+1 stack register)
- Register width increases from 32-bit to 64-bit
- Conditional jt/jf targets replaced with jt/fall-through
- Introduces bpf_call insn and register passing convention for zero overhead calls from/to other kernel functions



Performance

Just-In-Time compilation: BPF instructions → native code

- Alternative to kernel interpreter, brings performance
- Supported architectures:
  ARM32, ARM64, MIPS, PowerPC64, RiscV, Sparc64, s390, x86_32, x86_64
- Hardware offload: NFP (Netronome)
- all registers map one-to-one
- most of instructions map one-to-one
- bpf ‘call’ instruction maps to x86 ‘call’

Verifier – Ensure Termination and Safety

BPF programs come from user space: make sure they terminate / are safe

Termination:

- Direct Acyclic Graph (DAG), inspect instructions, prune safe paths Maximum: 4096 instructions... OH WAIT now "up to 1 million" for root No back edge (loops)
- Except function calls
- May change soon (bounded loops)

Safety:

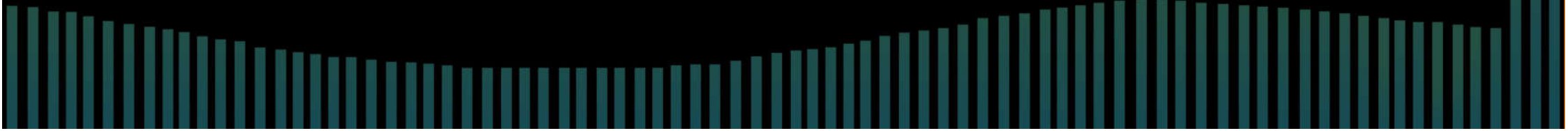
- No unreachable code, no jump out of range
- Registers and stack states are valid for every instruction
- Program does not read uninitialised registers/memory
- Program only interacts with relevant context (prevent out-of-bound/random memory access)
- ...

Verifier – Ensure Termination and Safety

"One of the more interesting features in this cycle is the ability to attach eBPF programs (**user-defined, sandboxed bytecode executed by the kernel**) to kprobes. This allows user- defined instrumentation on a live kernel image that can never crash, hang or interfere with the kernel negatively."

– Ingo Molnár (Linux developer)

Source: https://lkml.org/lkml/2015/4/14/232



BPF calling convention

BPF calling convention was carefully selected to match a subset of amd64/arm64 ABIs to avoid extra copy in calls:

- R0 – return value
- R1..R5 – function arguments
- R6..R9 – callee saved
- R10 – frame pointer

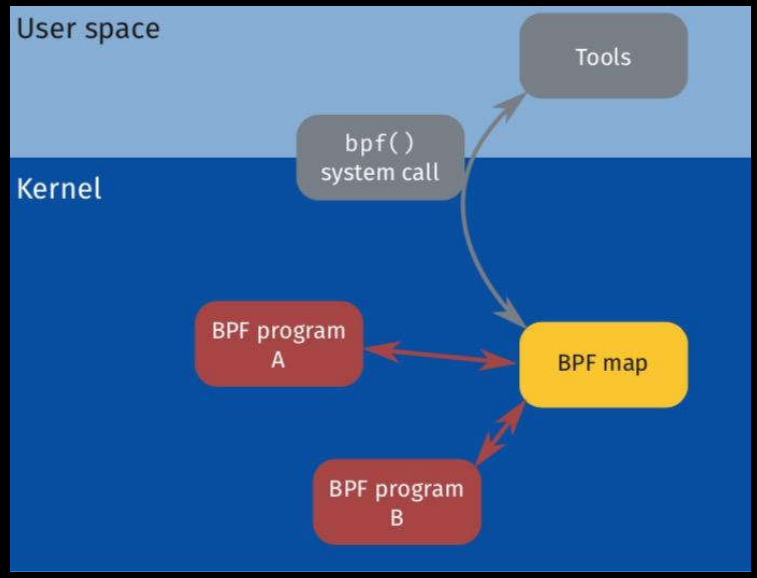
X86 Mapping

```
R0  -  rax
R1  -  rdi
R2  -  rsi
R3  -  rdx
R4  -  rcx
R5  -  r8
R6  -  rbx
R7  -  r13
R8  -  r14
R9  -  r15
R10 -  rbp
```

BPF maps

"Maps": special kernel memory area accessible to a program

- Shared between: several program runs, several programs, user space
- Typically, "key/value" storage: hash map, array
- Some of them have a "per-CPU" version
- Generally, RCU-protected; also, spinlocks now available in BPF

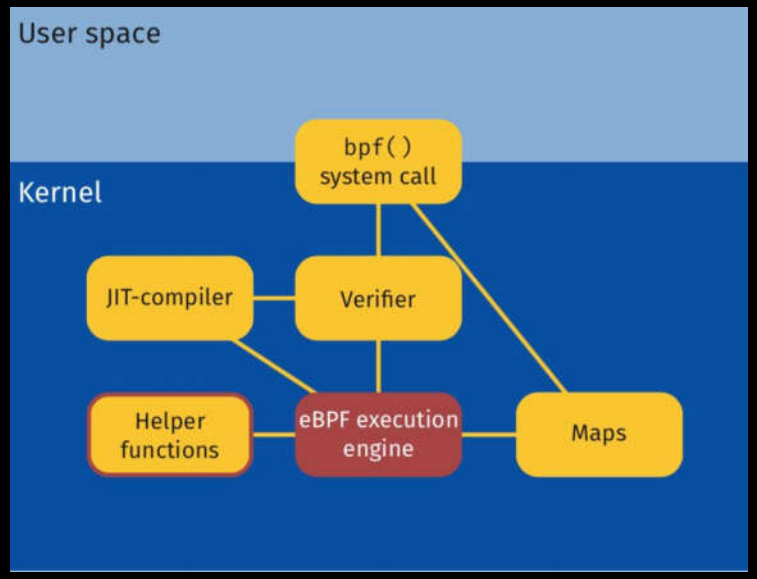


BPF maps

"Standard library" of functions implemented in the kernel

- Can be called from BPF programs
  Ease some tasks, manipulate maps, context, ..., e.g.:
  - Map lookup, update
  - Get kernel time
  - printk() equivalent
  - Change packet size
  - Redirect a packet
  - Safely dereference a kernel pointer
- Up to 5 arguments
  Some of them restricted to GPL-compatible BPF programs

More than 100 helper functions in the kernel already



Extended BPF assembler

```
int bpf_prog(struct bpf_context *ctx)
{
      u64 loc = ctx->arg2;
      u64 init_val = 1;
u64 *value;

      value = bpf_map_lookup_elem(&my_map, &loc);
if (value)
            *value += 1;
      else
            bpf_map_update_elem(&my_map, &loc,
            &init_val, BPF_ANY);
      return 0;
}
```

compiled by LLVM from C to bpf asm

```
1: *(u64 *)(r10 -8) = r1
2: r1 = 1
3: *(u64 *)(r10 -16) = r1
4: r1 = map_fd
6: r2 = r10
7: r2 += -8
8: call 1
9: if r0 == 0x0 goto pc +4
10: r1 = *(u64 *)(r0 +0)
11: r1 += 1
12: *(u64 *)(r0 + 0) = r1
13: goto pc+8
14: r1 = map_fd
16: r2 = r10
17: r2 += -8
18: r3 = r10
19: r3 += -16
20: r4 = 0
21: call 2
22: r0 = 0
23: exit
```

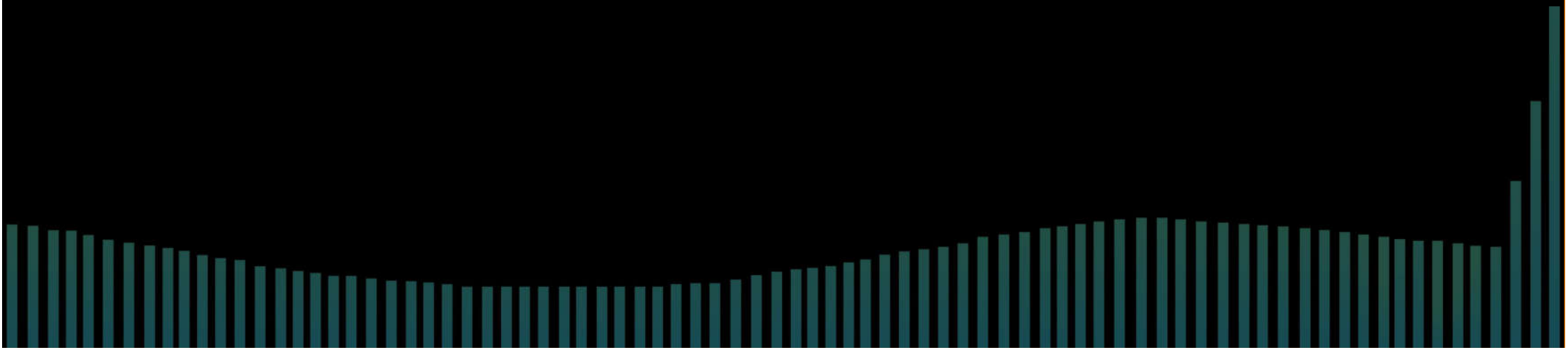
BPF compilers

- BPF backend for LLVM is in trunk and will be released as part of 3.7
- BPF backend for GCC is being worked on
- C front-end (clang) is used today to compile C code into BPF
- tracing and networking use cases may need custom languages
- BPF backend only knows how to emit instructions (calls to helper functions look like normal calls)

```
clang -O2 -g -emit-llvm -c prog.c -o - | \ llc -
march=bpf -mcpu=probe -filetype=obj -o prog.o
```

Other alternatives: Lua, Rust, ...

5 ebpf Tracing and Monitoring

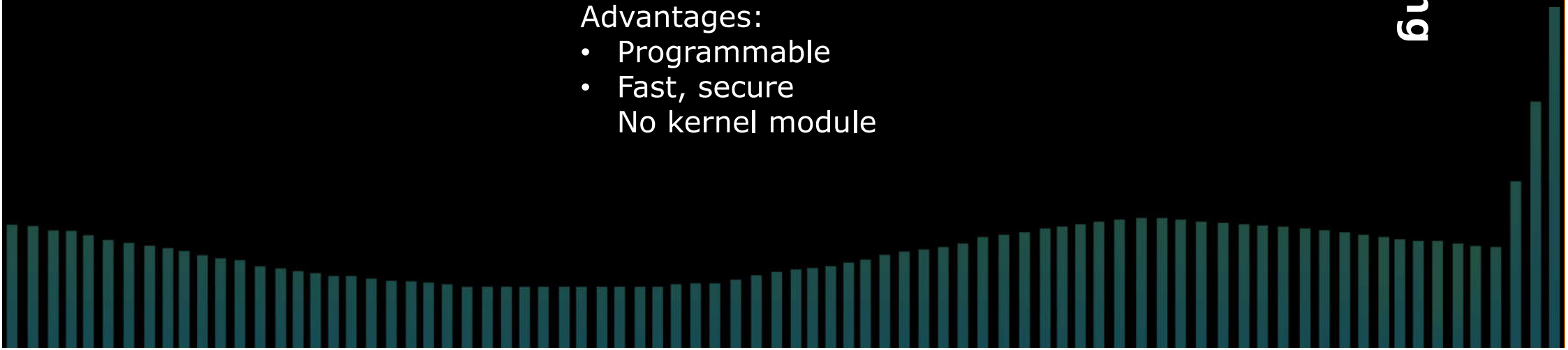


5.1 BPF for Tracing

BPF can attach to:
- kprobes/kretprobes
- uprobes/uretprobes
- tracepoints, USDT
- perf_events

Data collection:
- ftrace/sysfs
- perf ring buffer
- BPF maps

Advantages:
- Programmable
- Fast, secure
  No kernel module

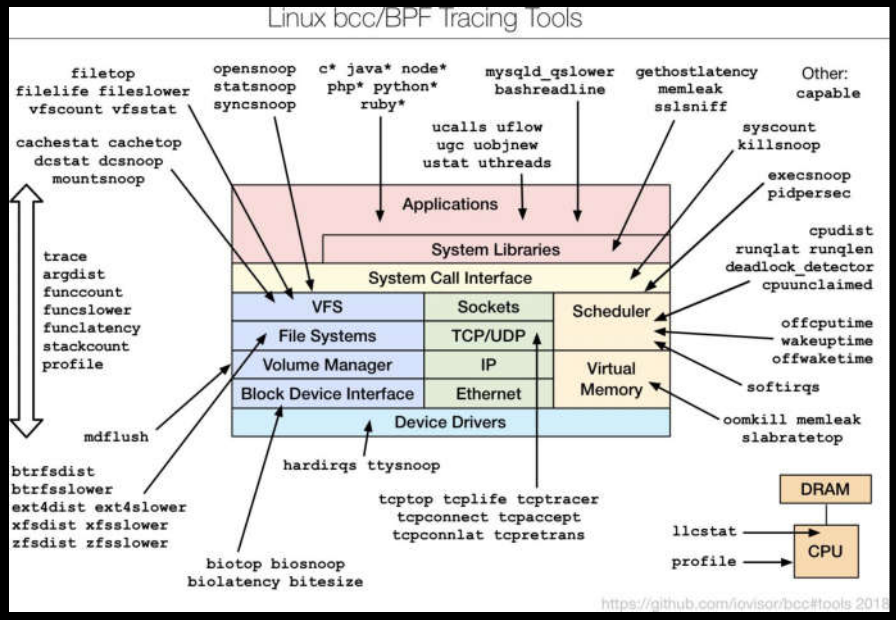


5.2 BPF Tracing Tools: bcc

bcc: Framework for BPF tools, mostly a set of Python wrappers

- Framework for BPF tools, mostly a set of Python wrapper.
- Also comes with a set of tools (87 tracing tools for now, +examples)

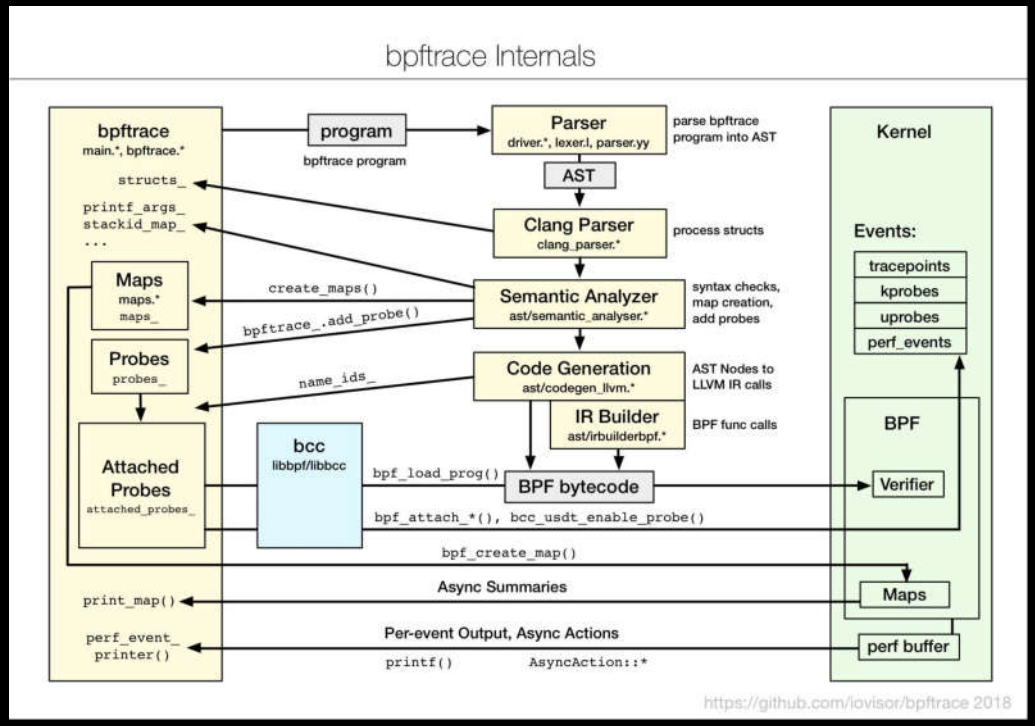
```
from bcc import BPF
b = BPF(text="""
#include <uapi/linux/ptrace.h>
int trace_open(struct pt_regs *ctx, int dfd,
        const char __user *filename, int flags)
{
        u64 id = bpf_get_current_pid_tgid();
        u32 pid = id >> 32;
        bpf_trace_printk("%d: open(%s, %x)\\n",
                pid, filename, flags);
        return 0;
}
""")
b.attach_kprobe(event="do_sys_open",
        fn_name="trace_open").trace_print()
```



5.3 BPF Tracing Tools: bpftrace

bpftrace is a high-level tracing language for Linux enhanced Berkeley Packet Filter (eBPF) available in recent Linux kernels (4.x)

- Awk-like syntax
- Sits on top of bcc
- Embeds a number of built-in functions and
- variables “Linux equivalent to DTrace”
- Express programs as one-liners, or very short scripts
- Also comes with a number of ready-to-use scripts
- (Has very good documentation!)

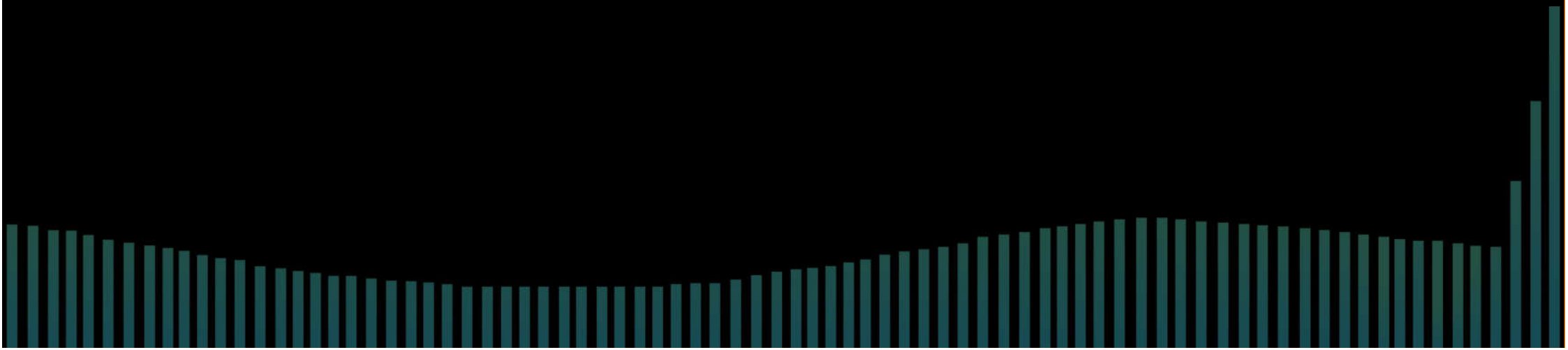


Our simple example to monitor open():

```
bpftrace -e 'kprobe:do_sys_open { printf("%d-%s: %s\n", pid, comm, str(arg1)) }'
```

6

Alibaba Tracing and Monitoring SIG



阿里巴巴跟踪诊断技术SIG(特别兴趣组)

小组简介

跟踪诊断技术是操作系统中必不可少的基础能力。在Alibaba Cloud Linux系统开源实践过程中，既有基于 eBPF 技术的bcc 工具集、NX 套件，又有内核各个子系统的 SLI 和 tracing 框架，也有在实际业务生产场景中落地的 Diagnose-tool。这些工具和能力都将在跟踪诊断技术 SIG 中呈现。

小组目标

为Alibaba Cloud Linux开源操作系统，提供一个全栈覆盖内核与核心组件的跟踪和诊断工具、平台，增强 OpenAnolis 全栈的可观察性与可靠性。

OpenAnolis