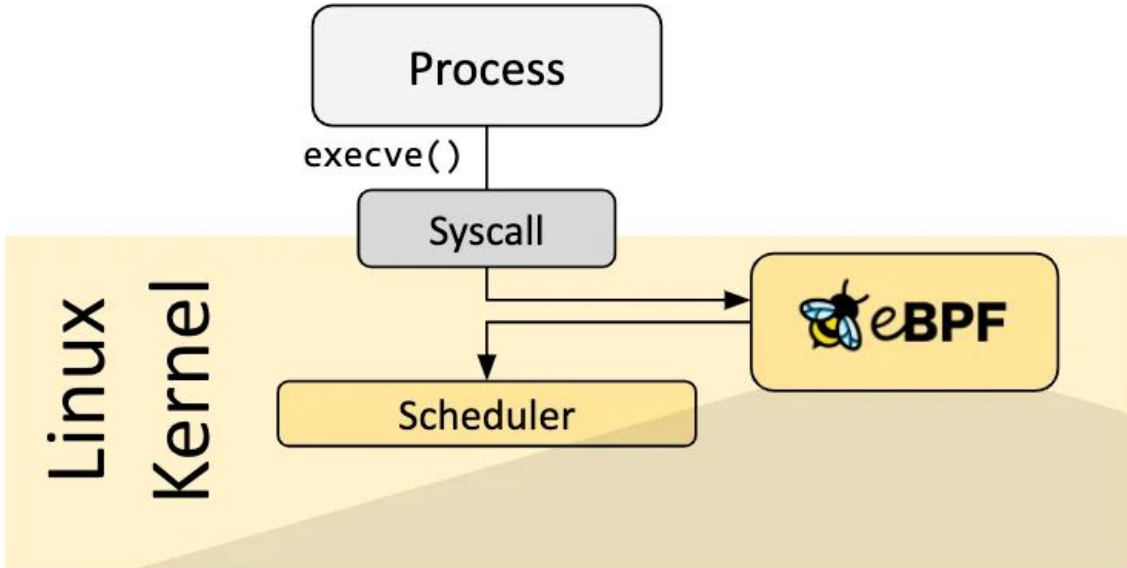

KubeCon
CloudNativeCon
Europe 2023
TiKV

Past, Present, and Future of eBPF in Cloud Native Observability

Natalie Serrino, New Relic
Frederic Branczyk, Polar Signals

Intro to eBPF



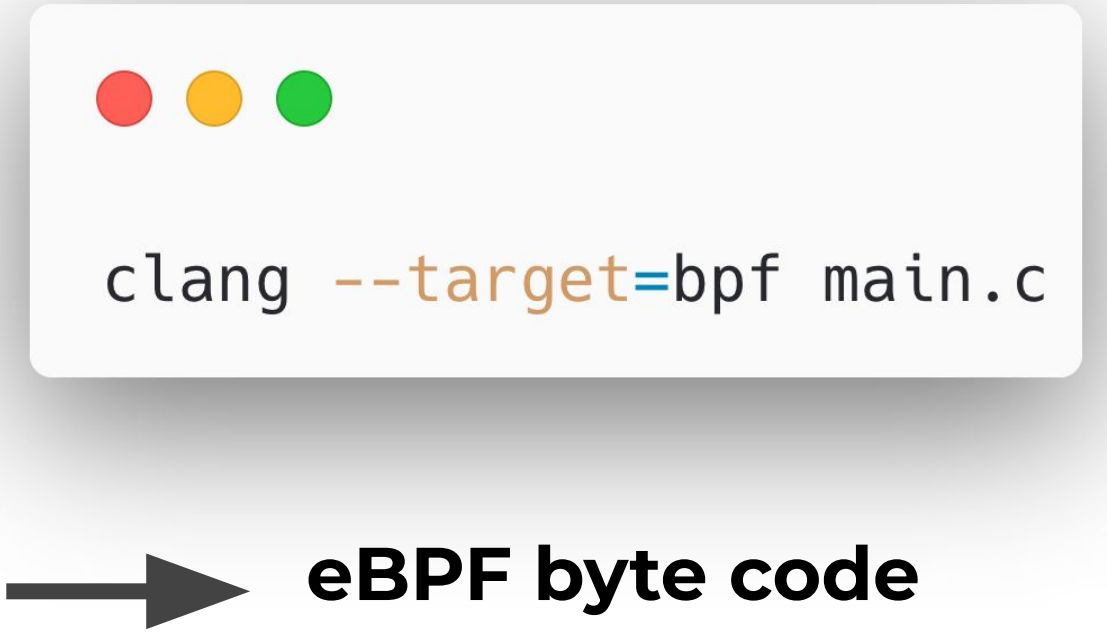

```c
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

        bpf_get_current_comm(&event.comm, sizeof(event.comm));
        comm_events.perf_submit(ctx, &event, sizeof(event));

        return 0;
}
```

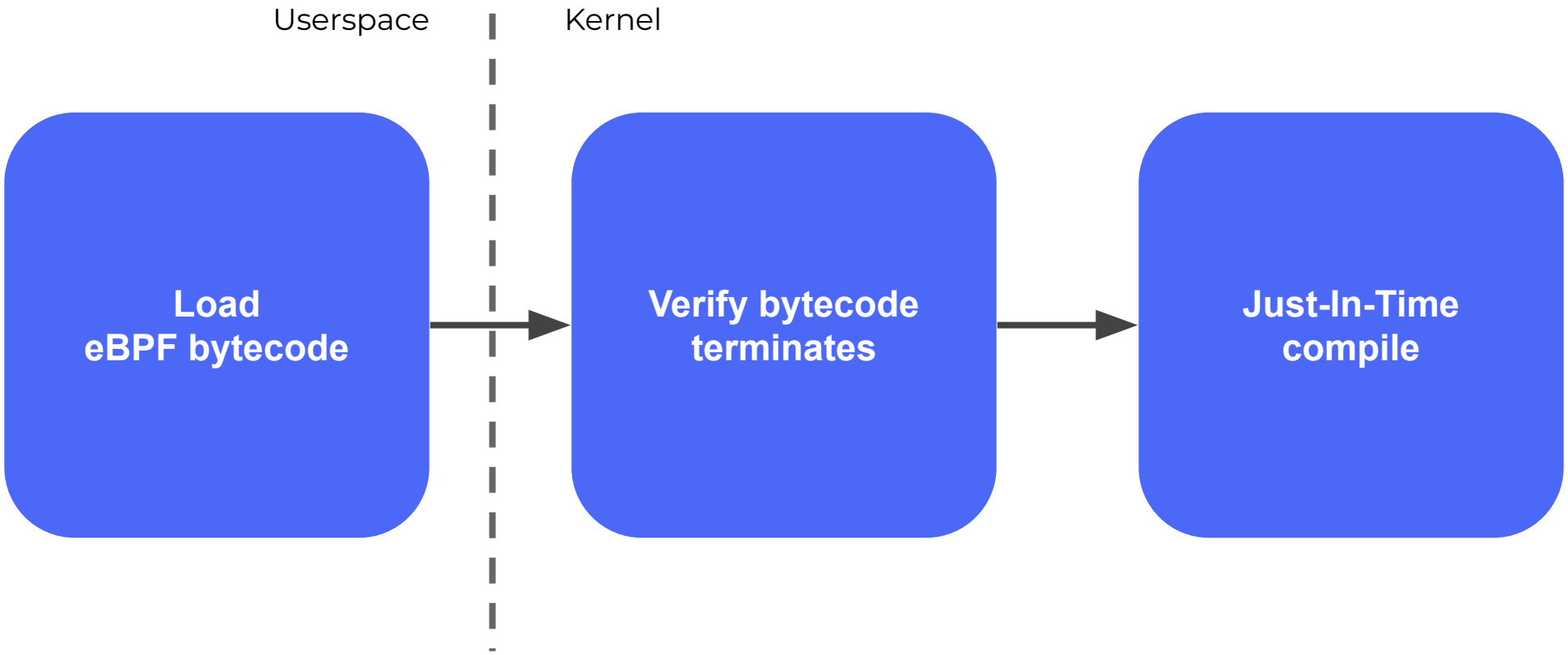
Hooks

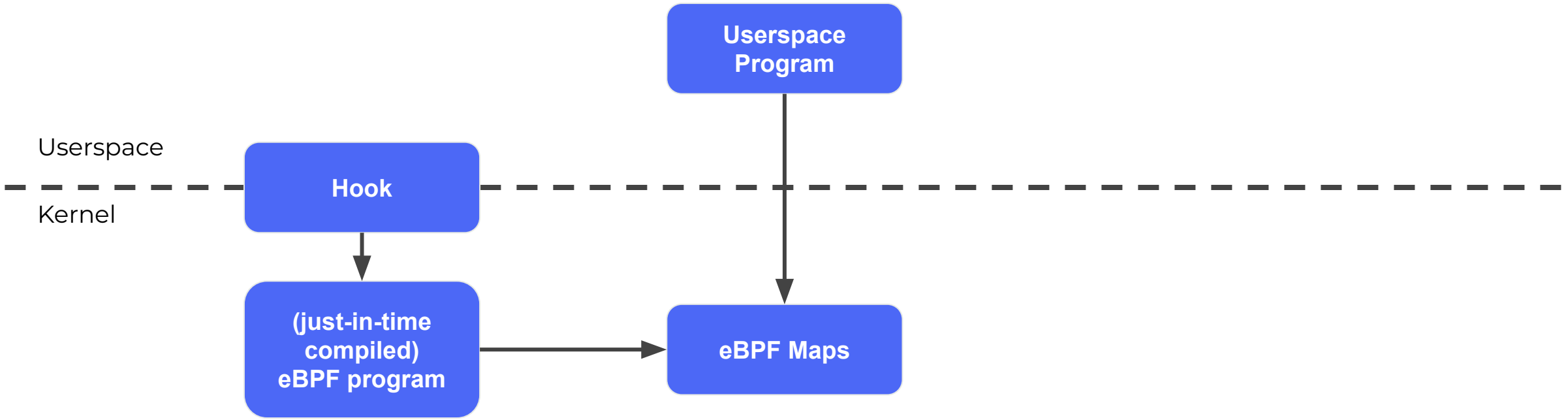
- Pre-defined (syscalls, kernel-functions, tracepoints, network events...)
- Custom hooks
  - kprobe
  - uprobe
  - perf_event

Compiling an eBPF program

Compiling an eBPF program




Userspace
Kernel
Load
eBPF bytecode
Verify bytecode
terminates
Just-In-Time
compile


Userspace Program
Userspace
Hook
Kernel
(just-in-time compiled) eBPF program
eBPF Maps

eBPF has been a buzzword

Let’s resolve some misconceptions

eBPF

That networking thing right?

FALSE

eBPF == all languages are supported

Demos are easy, actual wide language support is very hard, and not widespread!

The opposite: eBPF means interpreted languages are impossible

Also incorrect. It’s difficult, but possible.

Past

History and initial use cases of eBPF

Crossing The Chasm

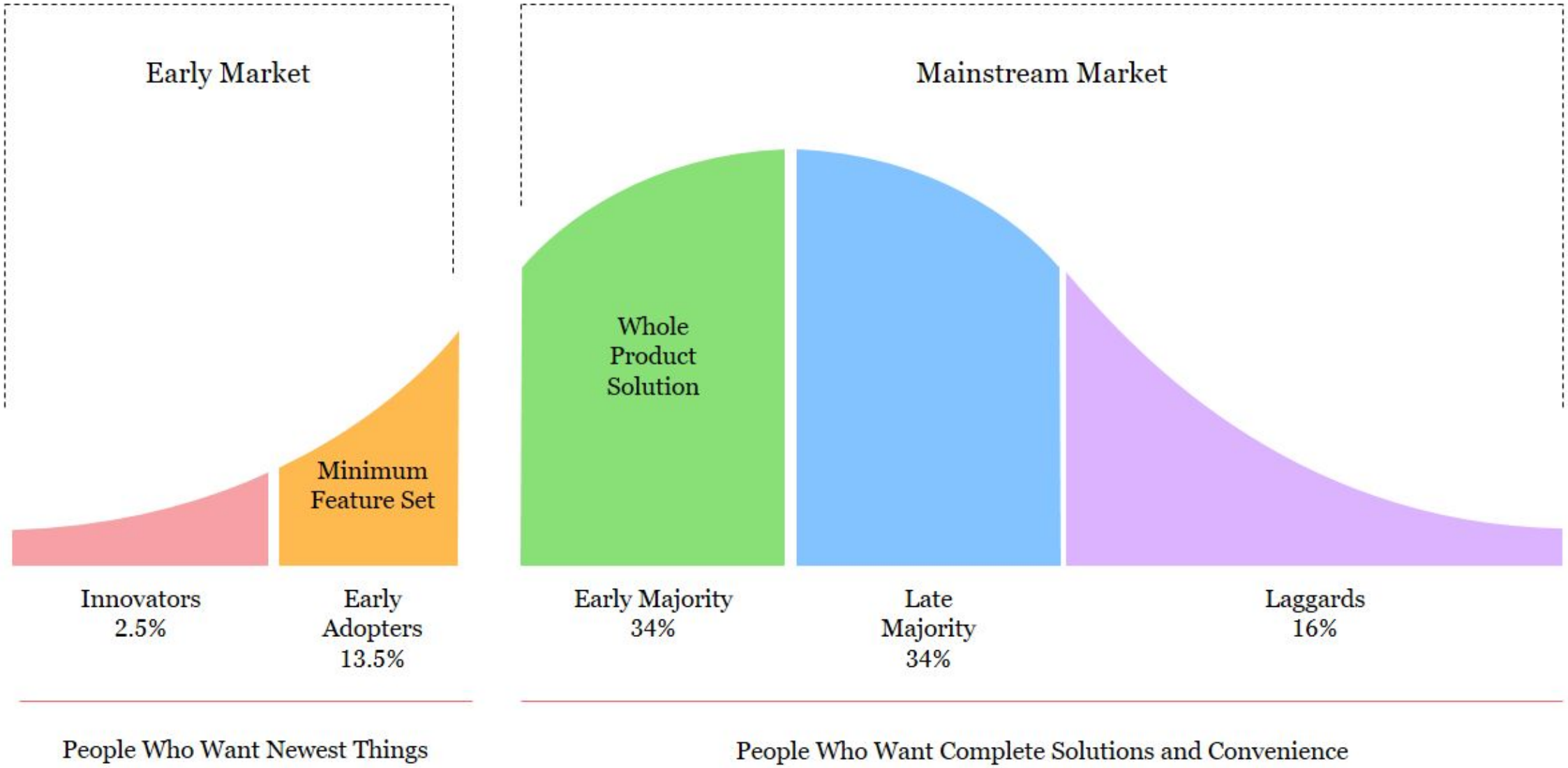


Image credit: Geoffrey A Moore

Crossing The Chasm

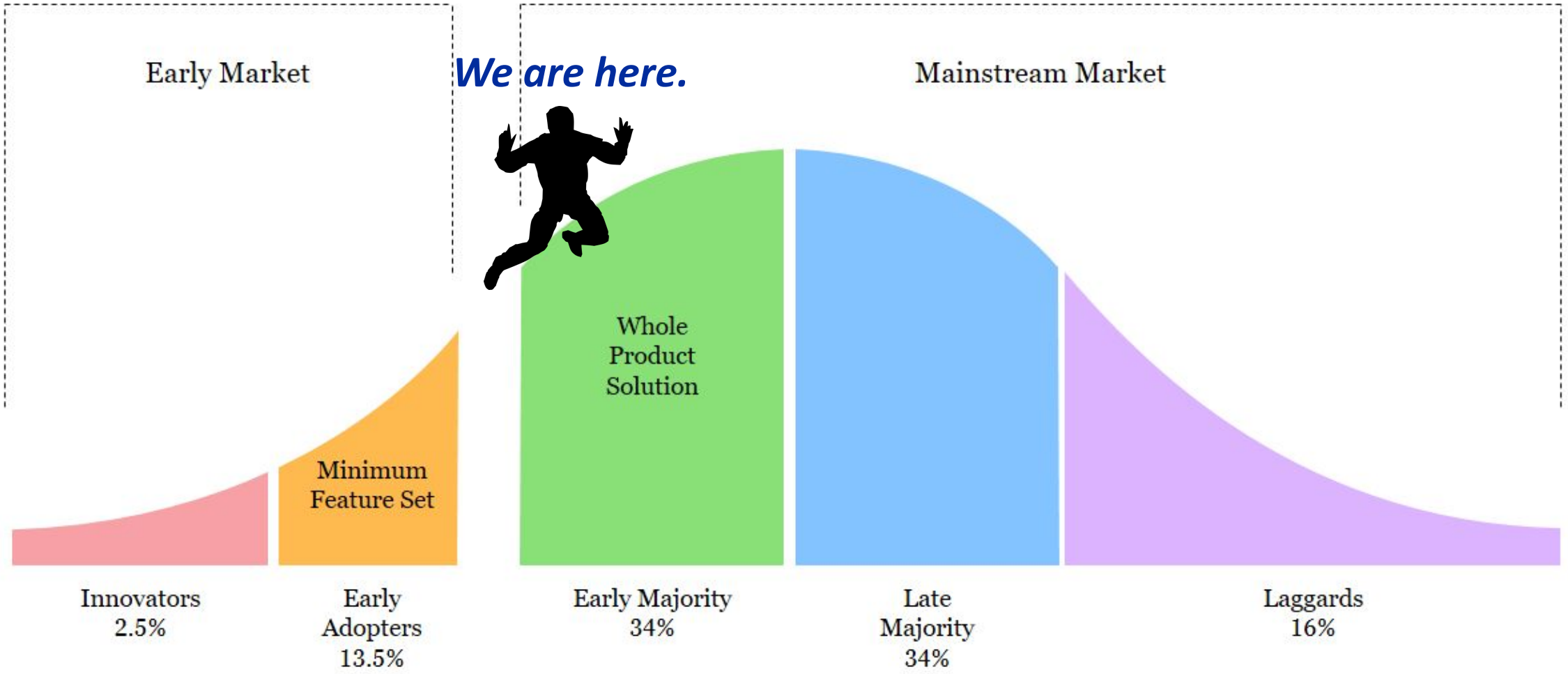
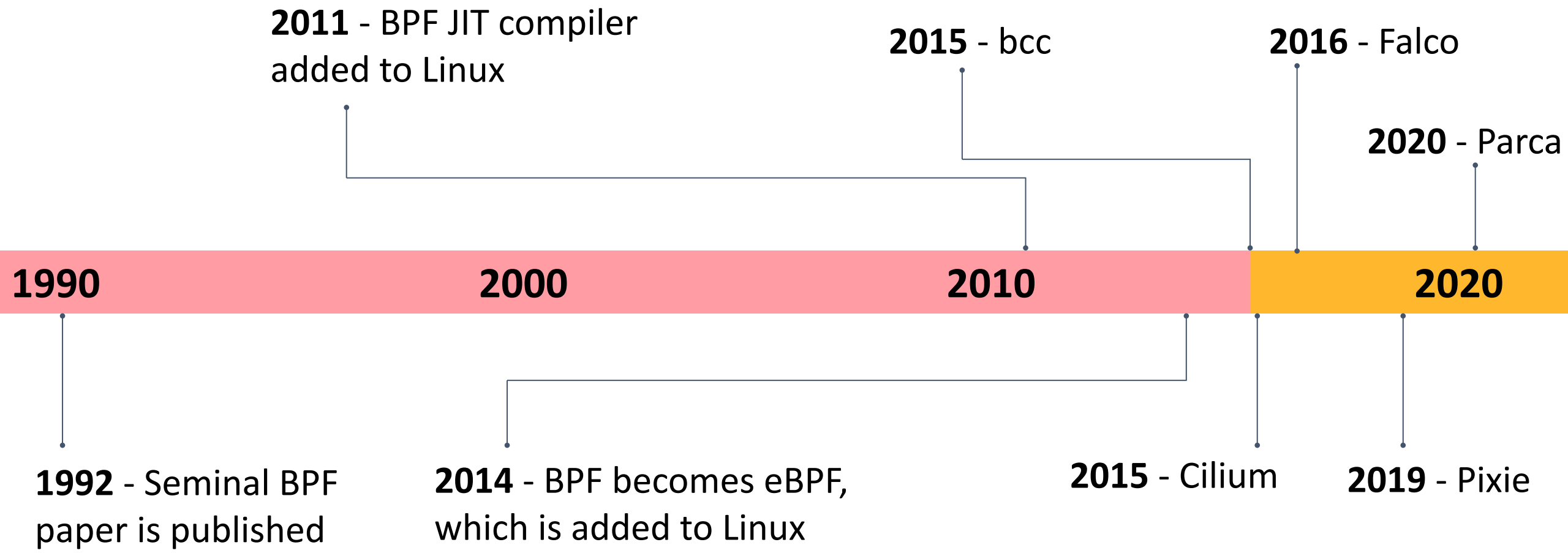


Image credit: Geoffrey A Moore

How did we get here?



What *was* the Berkeley Packet Filter and <u>why</u> was it created?

When: Late 80s/early 90s

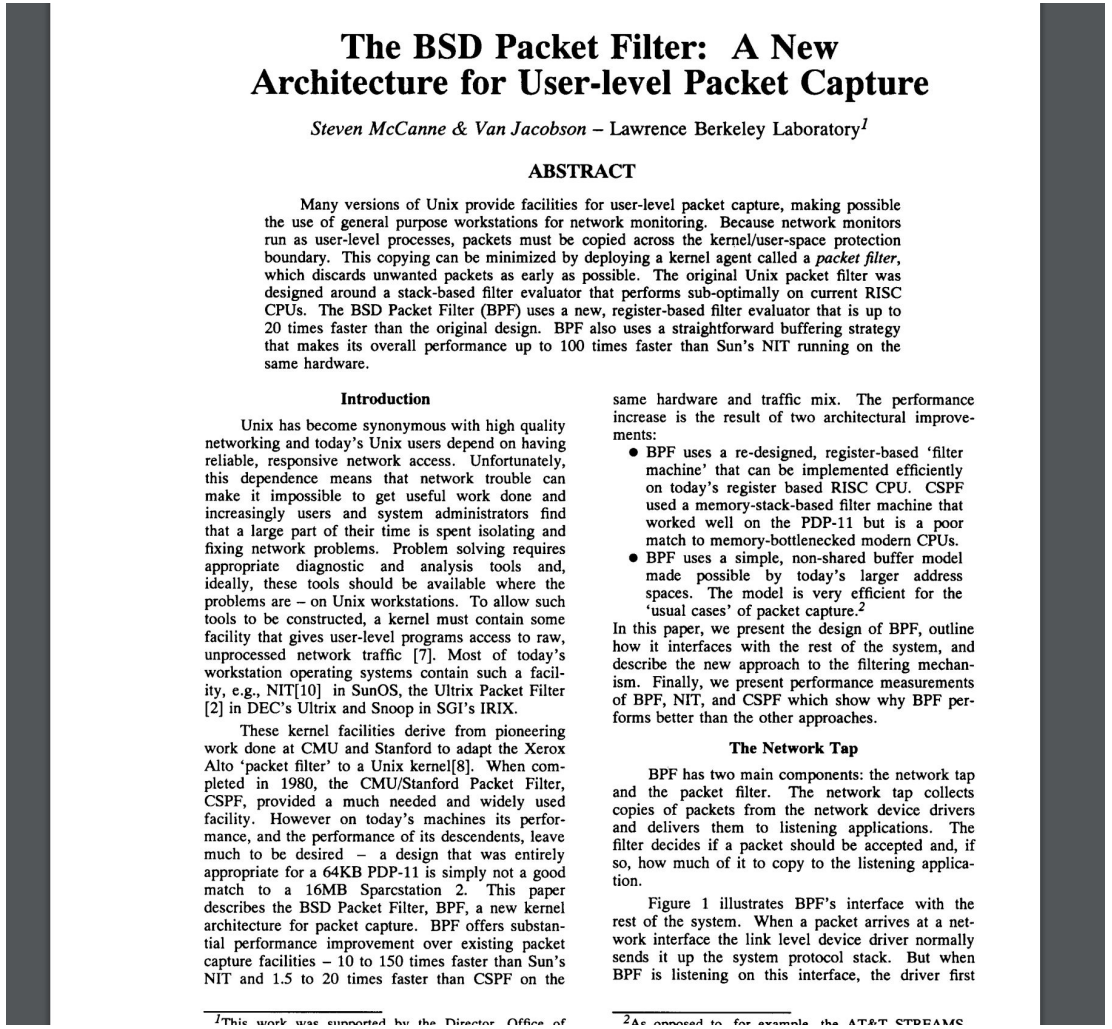
Who: Steven McCanne and Van Jacobson, Lawrence Berkeley Laboratory

Use case: Diagnosing and fixing network problems (packet filtering)

Motivation: Allow user-space programs to define rules for interacting with raw, unprocessed packets

Problem: Previous approaches were slow

Solution: a new packet filtering architecture, which allows the userspace programs to define packet rules that execute in the kernel



The BSD Packet Filter: A New Architecture for User-level Packet Capture

Steven McCanne & Van Jacobson – Lawrence Berkeley Laboratory[1]

ABSTRACT

Many versions of Unix provide facilities for user-level packet capture, making possible the use of general purpose workstations for network monitoring. Because network monitors run as user-level processes, packets must be copied across the kernel/user-space protection boundary. This copying can be minimized by deploying a kernel agent called a *packet filter*, which discards unwanted packets as early as possible. The original Unix packet filter was designed around a stack-based filter evaluator that performs sub-optimally on current RISC CPUs. The BSD Packet Filter (BPF) uses a new, register-based filter evaluator that is up to 20 times faster than the original design. BPF also uses a straightforward buffering strategy that makes its overall performance up to 100 times faster than Sun's NIT running on the same hardware.

Introduction

Unix has become synonymous with high quality networking and today's Unix users depend on having reliable, responsive network access. Unfortunately, this dependence means that network trouble can make it impossible to get useful work done and increasingly users and system administrators find that a large part of their time is spent isolating and fixing network problems. Problem solving requires appropriate diagnostic and analysis tools and, ideally, these tools should be available where the problems are – on Unix workstations. To allow such tools to be constructed, a kernel must contain some facility that gives user-level programs access to raw, unprocessed network traffic [7]. Most of today's workstation operating systems contain such a facility, e.g., NIT[10] in SunOS, the Ultrix Packet Filter [2] in DEC's Ultrix and Snoop in SGI's IRIX.

These kernel facilities derive from pioneering work done at CMU and Stanford to adapt the Xerox Alto 'packet filter' to a Unix kernel[8]. When completed in 1980, the CMU/Stanford Packet Filter, CSPF, provided a much needed and widely used facility. However on today's machines its performance, and the performance of its descendents, leave much to be desired – a design that was entirely appropriate for a 64KB PDP-11 is simply not a good match to a 16MB Sparcstation 2. This paper describes the BSD Packet Filter, BPF, a new kernel architecture for packet capture. BPF offers substantial performance improvement over existing packet capture facilities – 10 to 150 times faster than Sun's NIT and 1.5 to 20 times faster than CSPF on the same hardware and traffic mix. The performance increase is the result of two architectural improvements:

- BPF uses a re-designed, register-based 'filter machine' that can be implemented efficiently on today's register based RISC CPU. CSPF used a memory-stack-based filter machine that worked well on the PDP-11 but is a poor match to memory-bottlenecked modern CPUs.
- BPF uses a simple, non-shared buffer model made possible by today's larger address spaces. The model is very efficient for the 'usual cases' of packet capture.[2]

In this paper, we present the design of BPF, outline how it interfaces with the rest of the system, and describe the new approach to the filtering mechanism. Finally, we present performance measurements of BPF, NIT, and CSPF which show why BPF performs better than the other approaches.

The Network Tap

BPF has two main components: the network tap and the packet filter. The network tap collects copies of packets from the network device drivers and delivers them to listening applications. The filter decides if a packet should be accepted and, if so, how much of it to copy to the listening application.

Figure 1 illustrates BPF's interface with the rest of the system. When a packet arrives at a network interface the link level device driver normally sends it up the system protocol stack. But when BPF is listening on this interface, the driver first

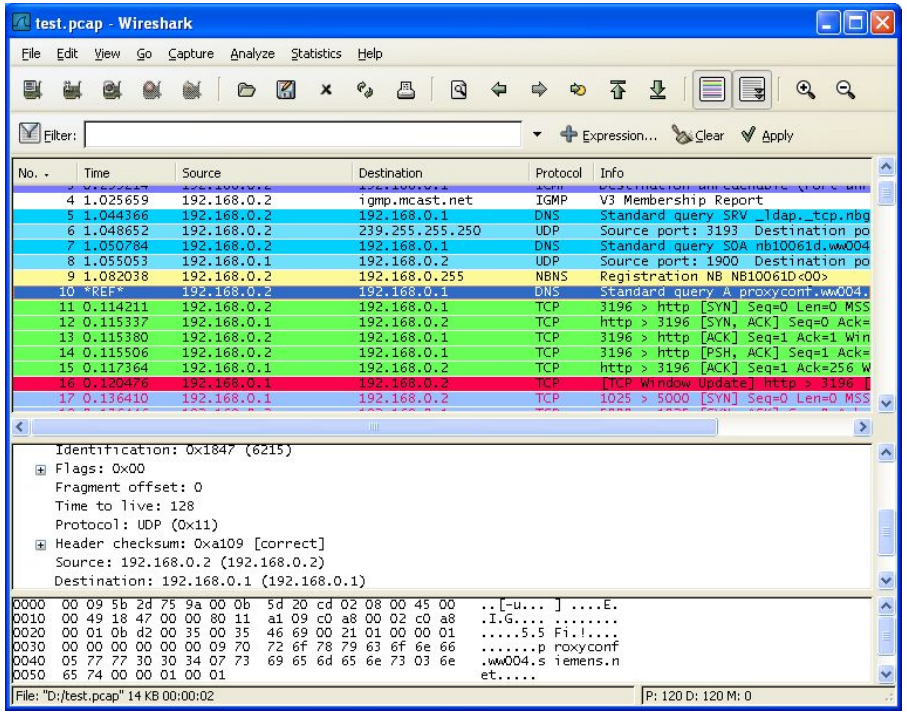
[1]This work was supported by the Director, Office of

[2]As opposed to, for example, the AT&T STREAMS

What happened with BPF from 1992-2011?

- State of the art for packet filtering
- Integrated into tcpdump and Wireshark
- BPF JIT added to Linux in 2011

Then, one day in 2014 …



From BPF to eBPF

The first version of eBPF was added to Linux in 2014.

It built upon BPF in some game-changing ways.

So then all hell broke loose.

eBPF programs are written in C

eBPF can invoke kernel functions

eBPF can view and edit raw kernel memory
*(safely, with the help of the new **verifier**)*

eBPF exists. Now what?

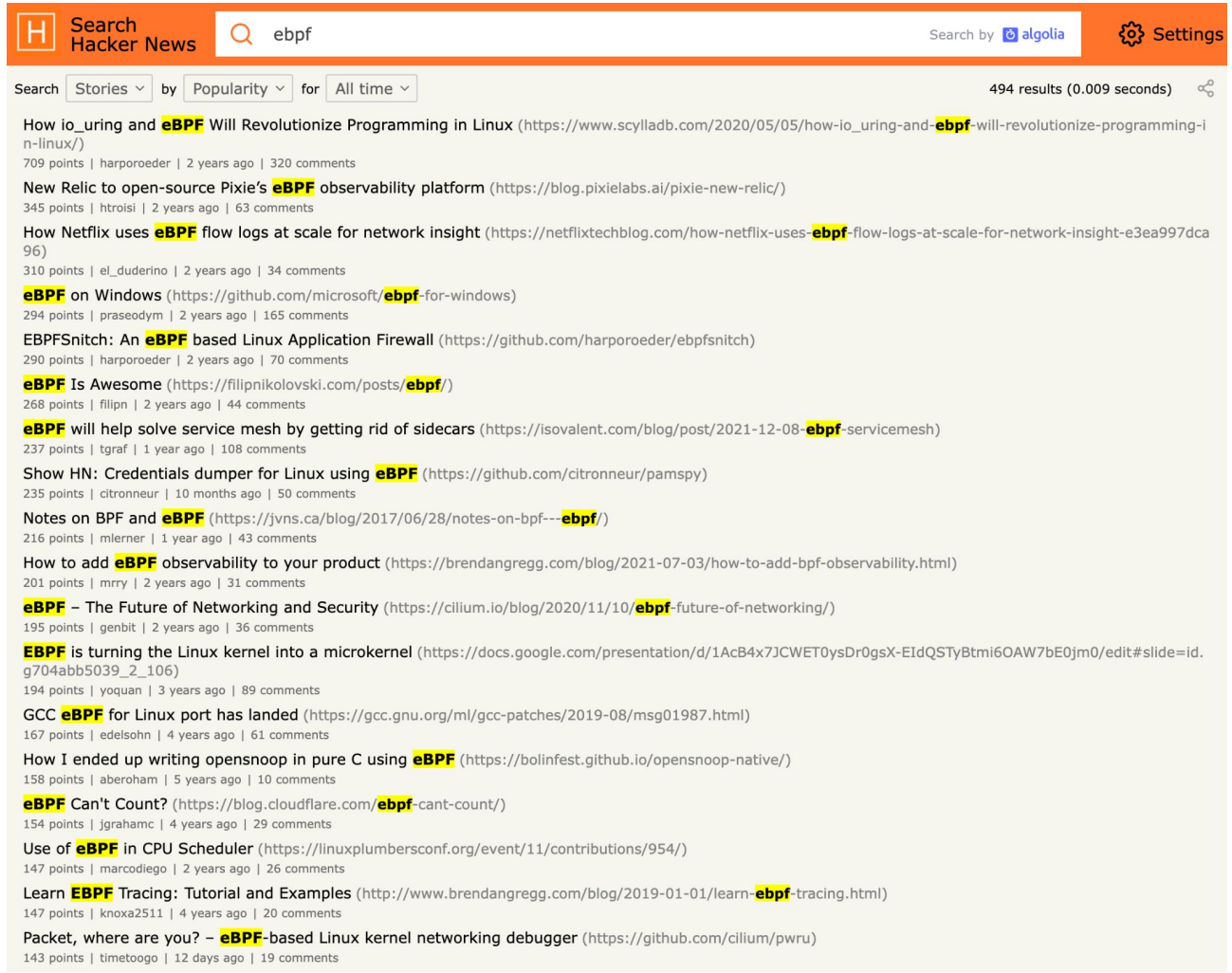
It's 2014, eBPF is brand new and super powerful. But there are some obstacles:

- Writing eBPF programs is technically difficult
- It's still relatively obscure (for now)
- People mostly think of it as a packet filter

Enter: libbpf, bpftrace, and bcc

- Writing eBPF programs is now more accessible
- More people try it
- Use cases beyond networking are gaining awareness

An explosion of adoption



Adoption [edit]

eBPF has been adopted by a number of large-scale production users, for example:

- Meta uses eBPF through their Katran layer 4 load-balancer for all traffic going to facebook.com[42][43][44][29]
- Google uses eBPF in GKE, developed and uses BPF LSM to replace audit and it uses eBPF for networking[27][
- Cloudflare uses eBPF for load-balancing and DDoS protection and security enforcement[48][49][50][51][52]
- Netflix uses eBPF for fleet-wide network observability and performance diagnosis[53][54]
- Dropbox uses eBPF through Katran for layer 4 load-balancing[55]
- Android uses eBPF for NAT46 and traffic monitoring[56][57][58]
- Alibaba uses eBPF for Kubernetes Pod load-balancing[59]
- Datadog uses eBPF for Kubernetes Pod networking and security enforcement[60][61][62]
- Trip.com uses eBPF for Kubernetes Pod networking[63][64]
- Microsoft ported eBPF and XDP to Windows[65][66][67]
- Seznam uses eBPF through Cilium for layer 4 load-balancing[68]
- CapitalOne uses eBPF for Kubernetes Pod networking[69]
- Apple uses eBPF for Kubernetes Pod security[70]
- Sky uses eBPF for Kubernetes Pod networking[71]
- Walmart uses eBPF for layer 4 load-balancing[72][73]
- Huawei uses eBPF through their DIGLIM secure boot system[74]
- Ikea uses eBPF for Kubernetes Pod networking[75]

eBPF FOUNDATION

KubeArmor

parca

PIXIE

Pyroscope

Falco

cilium

Mini Demo #1

Request tracing with Pixie

It was supposed to be so easy!

Present

The present of eBPF

3. `uprobe` / `uretprobe`: Dynamic Tracing, User-Level

To list available uprobes, you can use any program to list the text segment symbols from a binary, such as `objdump` and `nm`. For example:

```
# objdump -tT /bin/bash | grep readline
00000000007003f8 g    DO .bss   0000000000000004  Base        rl_readline_state
0000000000499e00 g    DF .text  00000000000001c5  Base        readline_internal_char
00000000004993d0 g    DF .text  0000000000000126  Base        readline_internal_setup
000000000046d400 g    DF .text  000000000000004b  Base        posix_readline_initialize
000000000049a520 g    DF .text  0000000000000081  Base        readline
[...]
```

This has listed various functions containing "readline" from /bin/bash. These can be instrumented using `uprobe` and `uretprobe`.

Examples:

```
# bpftrace -e 'uretprobe:/bin/bash:readline { printf("read a line\n"); }'
Attaching 1 probe...
read a line
read a line
read a line
^C
```

Reality

```
1 $ nm /usr/sbin/mariadbd
2 nm: /usr/sbin/mariadbd: no symbols
```

Binaries don’t typically have symbols

debuginfod

```
$ file /usr/sbin/mariadbd
/usr/sbin/mariadbd: ELF 64-bit LSB pie executable, ARM aarch64, version 1 (GNU/Linux),
dynamically linked, interpreter /lib/ld-linux-aarch64.so.1,
BuildID[sha1]=a7dcb33714ad2d574f324cdb99b1dc14fc0613d8, for GNU/Linux 3.7.0, stripped

$ curl https://debuginfod.elfutils.org/buildid/a7dcb33714ad2d574f324cdb99b1dc14fc0613d8/debuginfo
#      https://debuginfod.elfutils.org/buildid/<build-id>/debuginfo
```

clients

debuginfod client-side support is under construction or already available in a variety of binary-related utilities. We summarize current upstream status [2023-03] below. Note that distros may lag behind upstream developments.

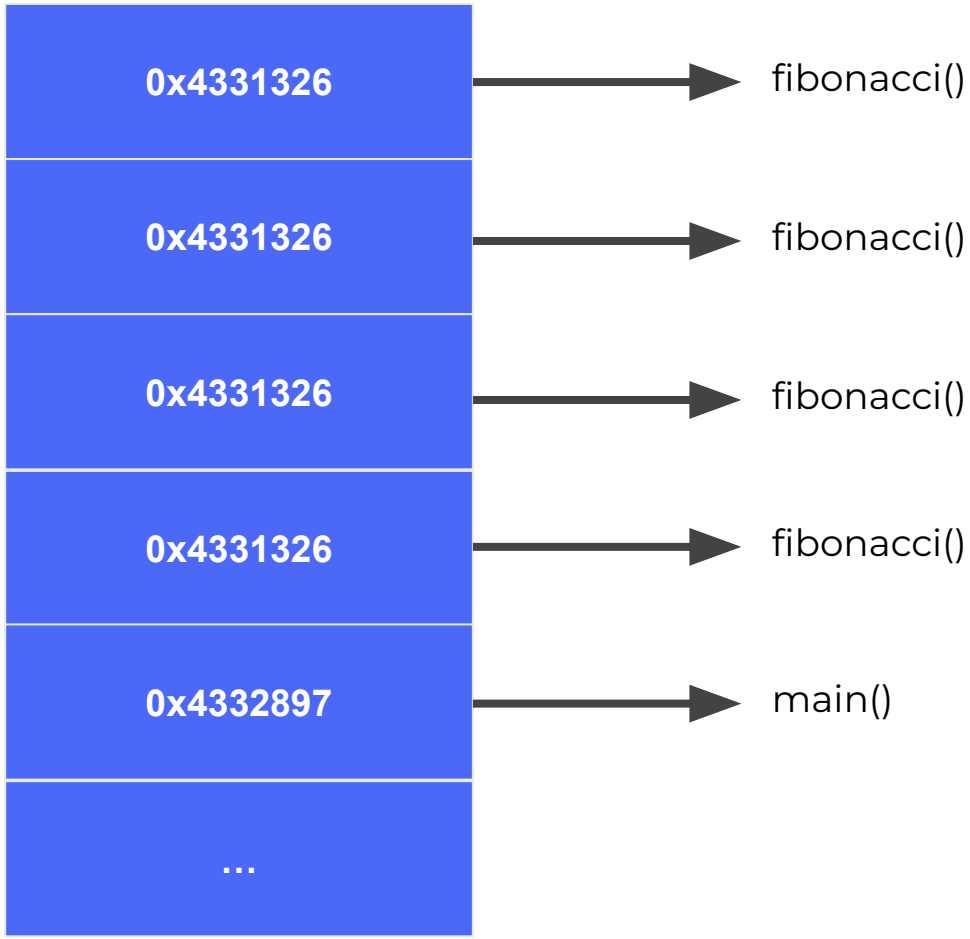
tool	status
elfutils	released in version 0.178, 2019-11
systemtap	automatic via elfutils
dwarves	automatic via elfutils
dwgrep	automatic via elfutils
ltrace	automatic via elfutils
libabigail	automatic via elfutils
binutils	released in version 2.34, 2020-02
gdb	released in version 10.1, 2020-10
dyninst	released in version 11.0 2021-04
valgrind	released in version 3.17.0, 2021-03
annocheck	released in version 9.03, 2020-01
delve	released in version 1.7.2, 2021-09
llvm	symbolizer merged, server merged, lldb help wanted, see also
bpftrace	released in version 0.21.0, 2021-07
perf	released in linux 5.10, 2021-01
systemd-coredumpd	help wanted
retrace/abrt/faf	in progress amerey@redhat.com
vtune	help wanted
pixie	help wanted
sentry symbolicator	partly released partly help wanted
VS Code	partly automatic via gdb
WinDbg	released in version 1.2104.13002.0, 2021-04
UDB	released in version 6.5
parca	released in version 0.8.0, 2022-02

https://sourceware.org/elfutils/Debuginfod.html

Example program: Fibonacci

```
 1 func main() {
 2   n := fibonacci(os.Args[1])
 3   fmt.Println(n)
 4 }
 5
 6 func fibonacci(n int) int {
 7     if n <= 1 {
 8         return n
 9     }
10     return FibonacciRecursion(n-1) + FibonacciRecursion(n-2)
11 }
```

Captured stack from eBPF

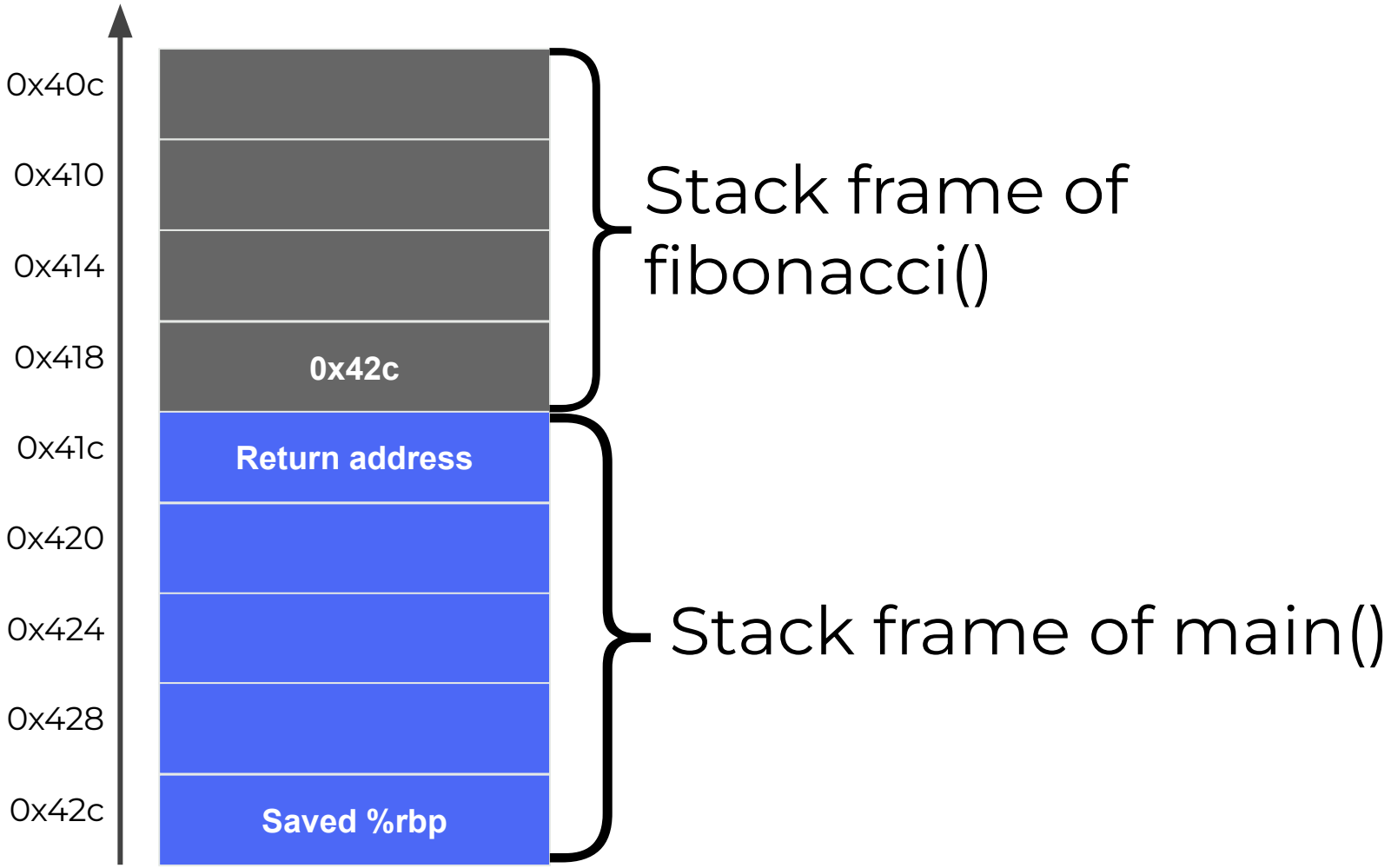


```c
typedef struct stack_count_key {
  u32 pid;
  int user_stack_id;
  int kernel_stack_id;
} stack_count_key_t;

#define MAX_STACK_ADDRESSES 1024
#define MAX_STACK_DEPTH 127

BPF_HASH(counts, stack_count_key_t, u64);
BPF_STACK_TRACE(stack_traces, MAX_STACK_ADDRESSES);

int profile_cpu(struct bpf_perf_event_data *ctx) {
  // ... extract PID ...

  stack_count_key_t key = {
      .pid = tgid,
  };

  int stack_id = bpf_get_stackid(ctx, &stack_traces, BPF_F_USER_STACK);
  key.user_stack_id = stack_id;

  int kernel_stack_id = bpf_get_stackid(ctx, &stack_traces, 0);
  key.kernel_stack_id = kernel_stack_id;

  u64 zero = 0; u64 *count;
  count = bpf_map_lookup_or_try_init(&counts, &key, &zero);
  if (!count)
    return 0;

  __sync_fetch_and_add(count, 1);
  return 0;
}
```

Full code at:
https://github.com/parca-dev/parca-agent/blob/main/bpf/cpu/cpu.bpf.c

How do we get a stack? Frame pointers!

Register	Value
%rsp	0x40c
%rbp	0x418



Frame Pointers

==

Linked List

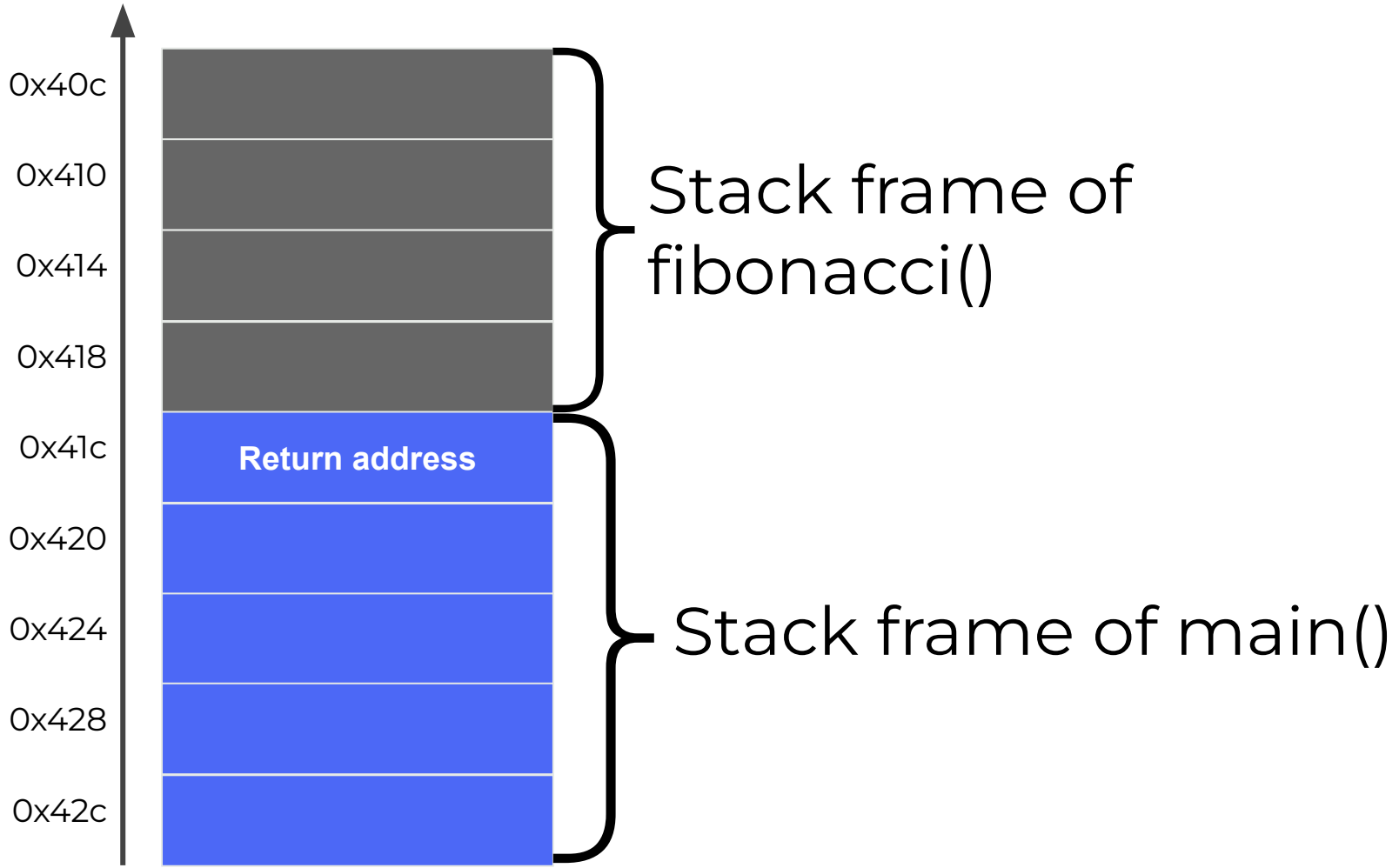
Evil compiler optimizations

```
# Don't do this
clang -fomit-frame-pointer main.c

# Do this
clang -fno-omit-frame-pointer main.c
```

How do we get a stack? ~~Frame pointers!~~

Register	Value
%rsp	0x40c
~~%rbp~~	~~0x418~~

Location	CFA	rbp	ra
0x40c	rsp+8	u	c-8
0x410	rsp+16	c-16	c-8
0x414	rbp+16	c-16	c-8
0x418	rsp+8	c-16	c-8



Linux Plumbers Conference 2022:

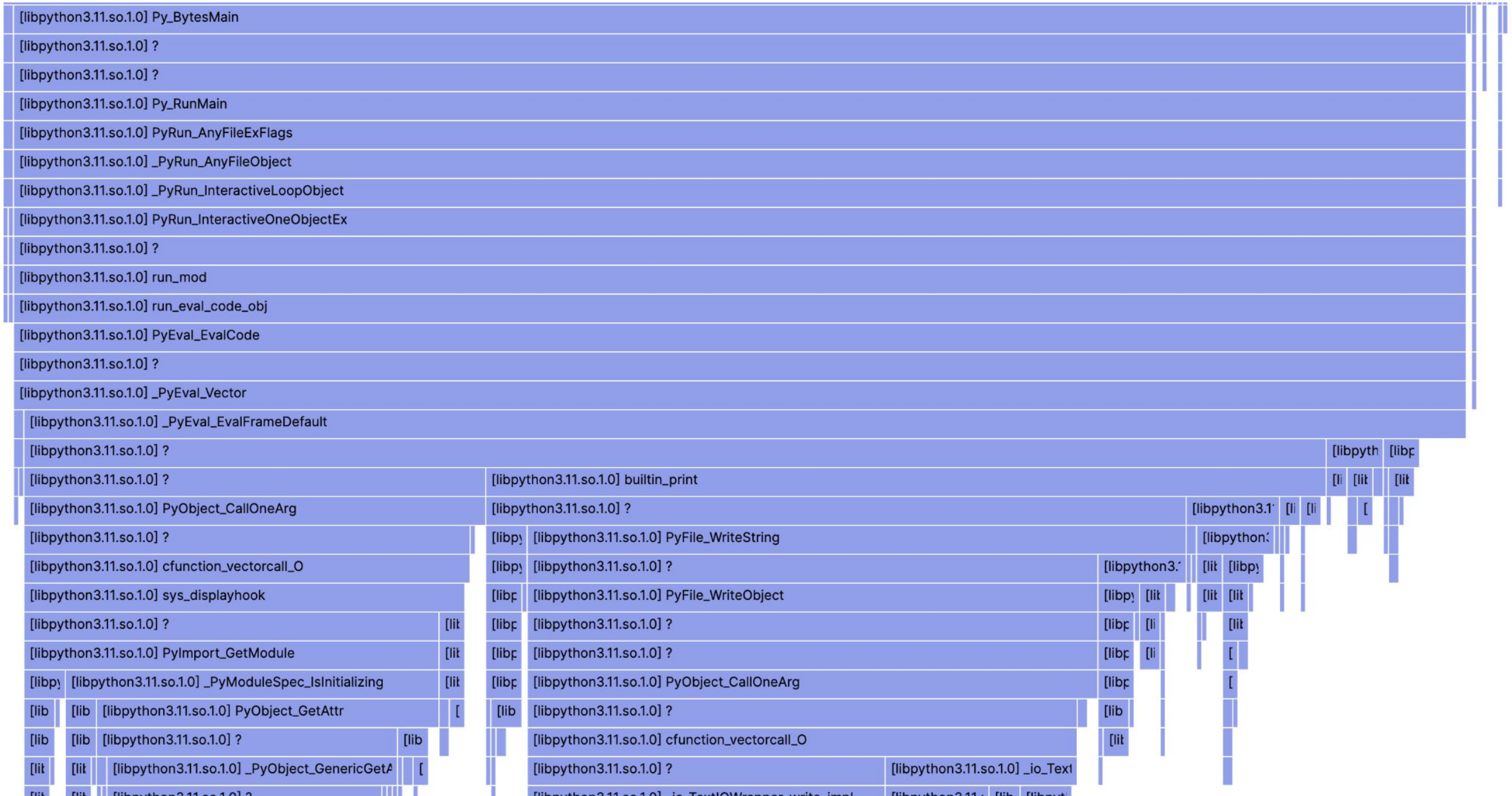
Developing eBPF profilers for polyglot cloud-native applications

My point:

Stack Walking for *any* native binaries works!

What about interpreters?

Interpreters



Some prior art:

github.com/javierhonduco/rbperf

Parca Demo: https://parca.dev/


[kernel.kallsyms]
libblkid.so.1.1.0
libc-2.31.so
libmount.so.1.1.0
libsystemd-shared-245.so
systemd
Everything else
root
[libc-2.31.so] __libc_start_main
[systemd]0x3c7e3
[systemd]0xec2b5
[libsystemd-shared-245.so] sd_event_run
[libsystemd-shared-245.so] sd_event_dispatch
[libsystemd-shared-245.so] sd_event_source_set_priority
[systemd]0x55a42
[systemd]0x553b2
[libmount.so.1.1.0] __mnt_table_parse_mtab
[libmount.so.1.1.0] mnt_table_parse_file
[libmount.so.1.1.0] mnt_table_parse_next
[libmount.so.1.1.0] mnt_table_parse_stream
[libc-2.31.so] _IO_getdelim
[libc-2.31.so] _IO_new_file_underflow
[libc-2.31.so] __libc_read
[libc-2.31.so] __libc_read
[[kernel.kallsyms]] entry_SYSCALL_64_after_h
[[kernel.kallsyms]] do_syscall_64
[[kernel.kallsyms]] __x64_sys_read
[[kernel.kallsyms]] ksys_read
[[kernel.kallsyms]] vfs_read
[[kernel.kallsyms]] __vfs_read
[[kernel.kallsyms]] seq_read
[[kernel.kallsyms]] m_show
[[kernel.kallsyms]] show_mountinfo
[systemd]0x55433
[systemd]0x54936
[systemd]0x52838
[systemd]0xe2bfe

Future

The future of eBPF

Where are we today with eBPF?

Demonstrated applicability to a wide set of use cases

- Networking (Load balancing, routing, firewalling…)
- Security (System call filtering, access control…)
- Observability (Application profiling, performance monitoring…)

Breadth and depth of coverage is hardening

- Improving support in programming languages
- Adoption of USDTs for common libraries
- Cloud providers are looking to add eBPF to serverless platforms

Higher-level APIs have been established

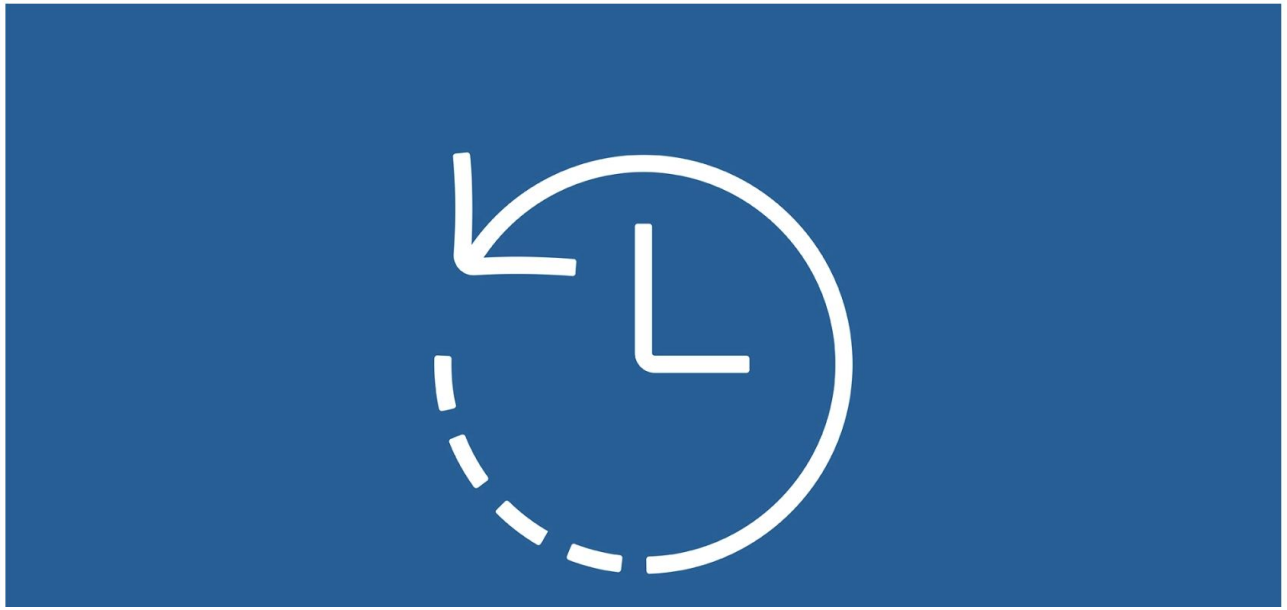
Time-travel debuggers?

Engineering at Meta

Open Source | Platforms | Infrastructure Systems | Physical Infrastructure | Video Engineering & AR/VR

POSTED ON APRIL 27, 2021 TO DEVINFRA

Reverse debugging at scale



https://engineering.fb.com/2021/04/27/developer-tools/reverse-debugging/

What frontiers might be addressed next?

- ⛷️ *Performance*: Efficiently supporting large numbers of probes reading large amounts of data
- 🎯 *Accessibility*: Identifying what, where, and how to instrument
- 📊 *Analysis*: Interpreting the low-level “firehose” of raw data

Mini Demo #3

Generating eBPF probes with LLMs

Session QR Codes will be
sent via email before the event

Please scan the QR Code above
to leave feedback on this session