第二届 eBPF开发者大会
www.ebpftravel.com

Bpftime: Userspace eBPF runtime

https://github.com/eunomia-bpf/bpftime

https://arxiv.org/abs/2311.07923

Yu Tong, Yusheng Zheng
yunwei356@gmail.com

中国·西安

Agenda

- Why a new userspace eBPF runtime?
  - Kernel Uprobe Performance Issues
  - Kernel eBPF Security Concerns and limited configurable
  - Other userspace eBPF runtime limitations
  - Existing Non-kernel eBPF Usecases
- Introduction to bpftime
- How it works
- Examples & benchmark
- Roadmap
- Q&A

Why bpftime?

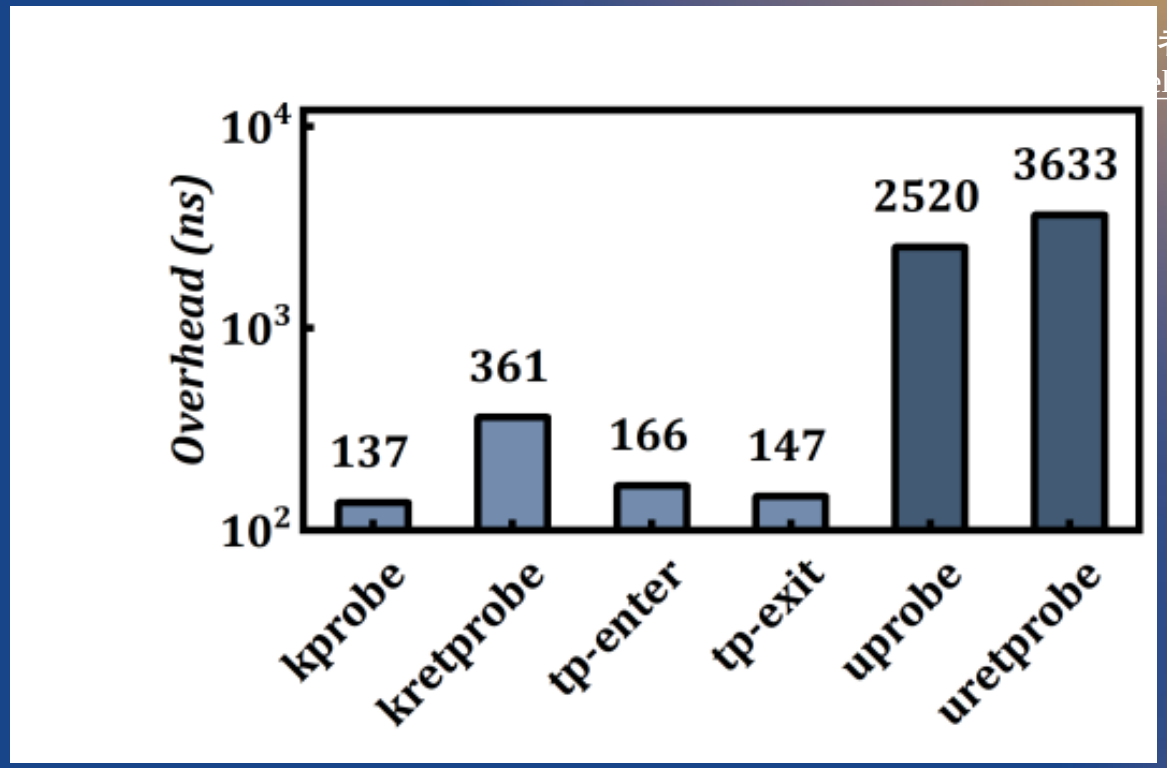
Uprobe: User-level dynamic tracing

1. Kernel Uprobe Performance Issues:

- Current UProbe implementation necessitates two kernel context copies.
- Results in significant performance overhead.
- Not suitable for real-time monitoring in latency-sensitive applications.

And Kernel Syscall tracepoint:

Syscall tracepoints will hook all syscalls and require filter for specific process



Uprobe's Wide Adoption in Production

- Traces user-space protocols: SSL, TLS, HTTP2.
- Monitors memory allocation and detects leaks.
- Tracks threads and goroutine dynamics.
- Provides passive, non-instrumental tracing.
- And more...

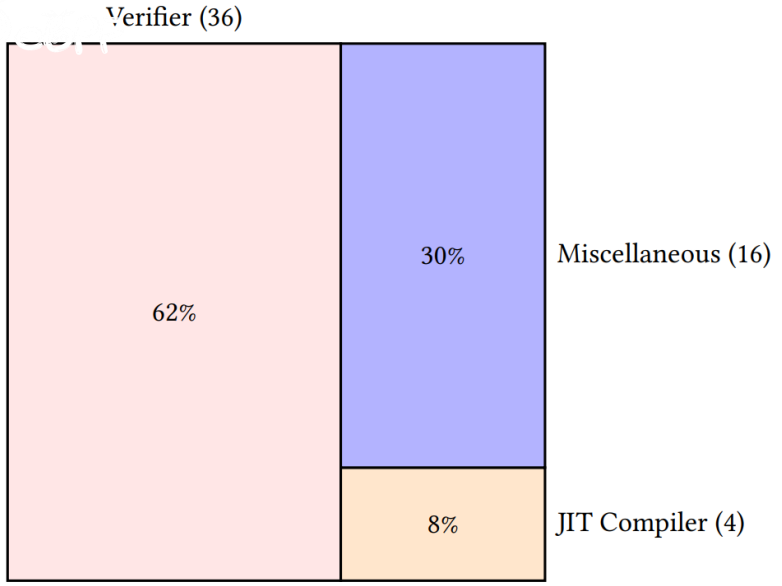


Figure 1: A tally of eBPF-related CVEs from 2010 to 2023. There are a total of 56 CVEs, the majority of which were discovered in the verifier.

Table 2: The offensive eBPF helpers.

ID	Helper Name	Functionality
H1	bpf_probe_write_user	Write any process's user space memory
H2	bpf_probe_read_user	Read any process's user space memory
H3	bpf_override_return	Alter return code of a kernel function
H4	bpf_send_signal	Send signal to kill any process
H5	bpf_map_get_fd_by_id	Obtain eBPF programs' eBPF maps fd

Why bpftime?

2. Kernel eBPF Security Concerns

eBPF programs run in kernel mode, requiring root access.

- Increases attack surface, posing risks like container escape.
- Inherent vulnerabilities in eBPF can lead to Kernel Exploits.

Limitations of Kernel eBPF

- Older kernel version, unprivileged environments, non-Linux system may not have access to kernel eBPF
- Verifier limited the operation of eBPF, config or extend eBPF may require kernel changes

Existing Non-kernel eBPF Usecases

- **Qemu+uBPF**: Combines Qemu with uBPF. Video.
- **Oko**: Extends Open vSwitch-DPDK with BPF. Enhances tools for better integration. GitHub.
- **Solana**: Userspace eBPF for High-performance Smart Contract. GitHub.
- **DPDK eBPF**: Libraries for fast packet processing. Enhanced by Userspace eBPF.
- **eBPF for Windows**: Brings eBPF toolchains and runtime to Windows kernel.

Papers:

- **Rapidpatch**: Firmware Hotpatching for Real-Time Embedded Devices
- **Femto-Containers**: Lightweight Virtualization and Fault Isolation For Small Software Functions on Low-Power IoT Microcontrollers

Networks + plugins + edge runtime + smart contract + hot patch + **Windows**

Bpftime: Userspace eBPF runtime

bpftime, a **full-featured**, **high-performance** eBPF runtime designed to operate in userspace：

- Fast Uprobe, USDT and Syscall hook capabilities
  - Userspace uprobe can be 10x faster than kernel uprobe
  - No mannual instrumentation or restart required, similar to kernel probe
  - Trace the user functions, syscalls or modify user function behavior
- Compatible with kernel eBPF toolchains and libraries
  - No need modify eBPF App
- Interprocess maps or kernel maps support, work together with kernel eBPF
  - Support “offload to userspace” and verify with kernel verifier
- New LLVM JIT/AOT vm for eBPF, which can be used as an independent library
- May use features like ringbuffer in lower kernel versions

Current support features

Userspace eBPF shared memory map types:

- BPF_MAP_TYPE_HASH
- BPF_MAP_TYPE_ARRAY
- BPF_MAP_TYPE_PROG_ARRAY
- BPF_MAP_TYPE_RINGBUF
- BPF_MAP_TYPE_PERF_EVENT_ARRAY
- BPF_MAP_TYPE_PERCPU_ARRAY
- BPF_MAP_TYPE_PERCPU_HASH

User-kernel shared maps:

- BPF_MAP_TYPE_HASH
- BPF_MAP_TYPE_ARRAY
- BPF_MAP_TYPE_PERCPU_ARRAY
- BPF_MAP_TYPE_PERF_EVENT_ARRAY

Prog types can attached in userspace:

- tracepoint:raw_syscalls:sys_enter
- tracepoint:syscalls:sys_exit_*
- tracepoint:syscalls:sys_enter_*
- uretprobe:*
- uprobe:*
- usdt:*

You can also define **other static tracepoints** and prog types in userspace app.

Support ~**30 kernel helper functions**

Support **kernel** or **userspace verifier**

Test JIT with **bpf_conformance**

Running mode of bpftime

- **Run eBPF in userspace only**
  - Can run without kernel on non-linux systems
  - Not very suitable for large eBPF applications
  - maps in shm can't be used by kernel eBPF programs
- **Run eBPF in userspace with kernel eBPF, a bpftime-daemon**
  - Compatible with kernel uprobe in behavior
    - Attach to new process or running process automatically
  - Support mix of uprobe and kprobe, socket…
  - Similar to fuse: userspace daemon + kernel code
    - No modify kernel, using eBPF module to monitor or change the behavior of BPF syscalls

Get started

- Use uprobe to monitor userspace malloc function in libc, with hash maps in userspace
- bpftime load/start
- Try eBPF in **GitHub codespace**

To get started, you can build and run a libbpf based eBPF program starts with `bpftime` cli:

```
make -C example/malloc # Build the eBPF program example
bpftime load ./example/malloc/malloc
```

In another shell, Run the target program with eBPF inside:

```
$ bpftime start ./example/malloc/test
Hello malloc!
malloc called from pid 250215
continue malloc...
malloc called from pid 250215
```

You can also dynamically attach the eBPF program with a running process:

```
$ ./example/malloc/test & echo $! # The pid is 101771
[1] 101771
101771
continue malloc...
continue malloc...
```

And attach to it:

```
$ sudo bpftime attach 101771 # You may need to run make install in root
Inject: "/root/.bpftime/libbpftime-agent.so"
Successfully injected. ID: 1
```

You can see the output from original program:

```
$ bpftime load ./example/malloc/malloc
...
12:44:35
        pid=247299      malloc calls: 10
        pid=247322      malloc calls: 10
```

Examples

Use uprobe to monitor userspace malloc function in libc, with hash maps, compatible with kernel

Run daemon

```
$ sudo SPDLOG_LEVEL=Debug build/daemon/bpftime_daemon
[2023-10-24 11:07:13.143] [info] Global shm constructed. shm_open_type 0 for bpftime_maps_shm
```

Run malloc example

```
$ sudo example/malloc/malloc
libbpf: loading object 'malloc_bpf' from buffer
11:08:11
11:08:12
11:08:13
```

Trace malloc calls in target

```
$ sudo example/malloc/victim
malloc called from pid 12314
continue malloc...
```

The other console will print the malloc calls in the target process.

```
20:43:22
        pid=113413      malloc calls: 9
```

Examples

- Use syscall tracepoint to monitor open and close syscall, with ring buffer for output

https://github.com/eunomia-bpf/bpftime

Usage

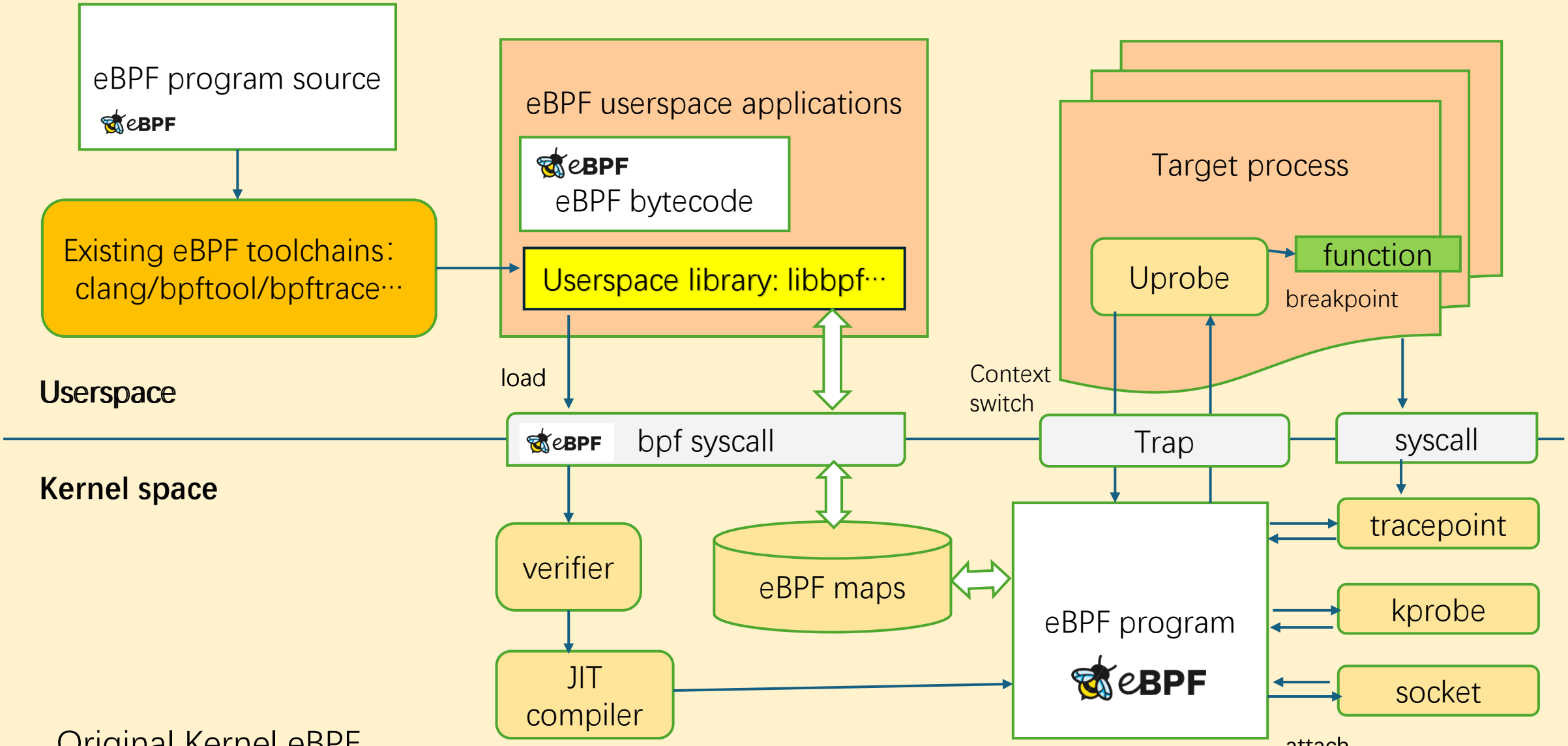
```
$ sudo ~/.bpftime/bpftime load ./example/opensnoop/opensnoop
[2023-10-09 04:36:33.891] [info] manager constructed
[2023-10-09 04:36:33.892] [info] global_shm_open_type 0 for bpftime_maps_shm
[2023-10-09 04:36:33][info][23999] Enabling helper groups ffi, kernel, shm_map by default
PID    COMM               FD ERR PATH
72101  victim              3   0 test.txt
72101  victim              3   0 test.txt
72101  victim              3   0 test.txt
72101  victim              3   0 test.txt
```

In another terminal, run the victim program:

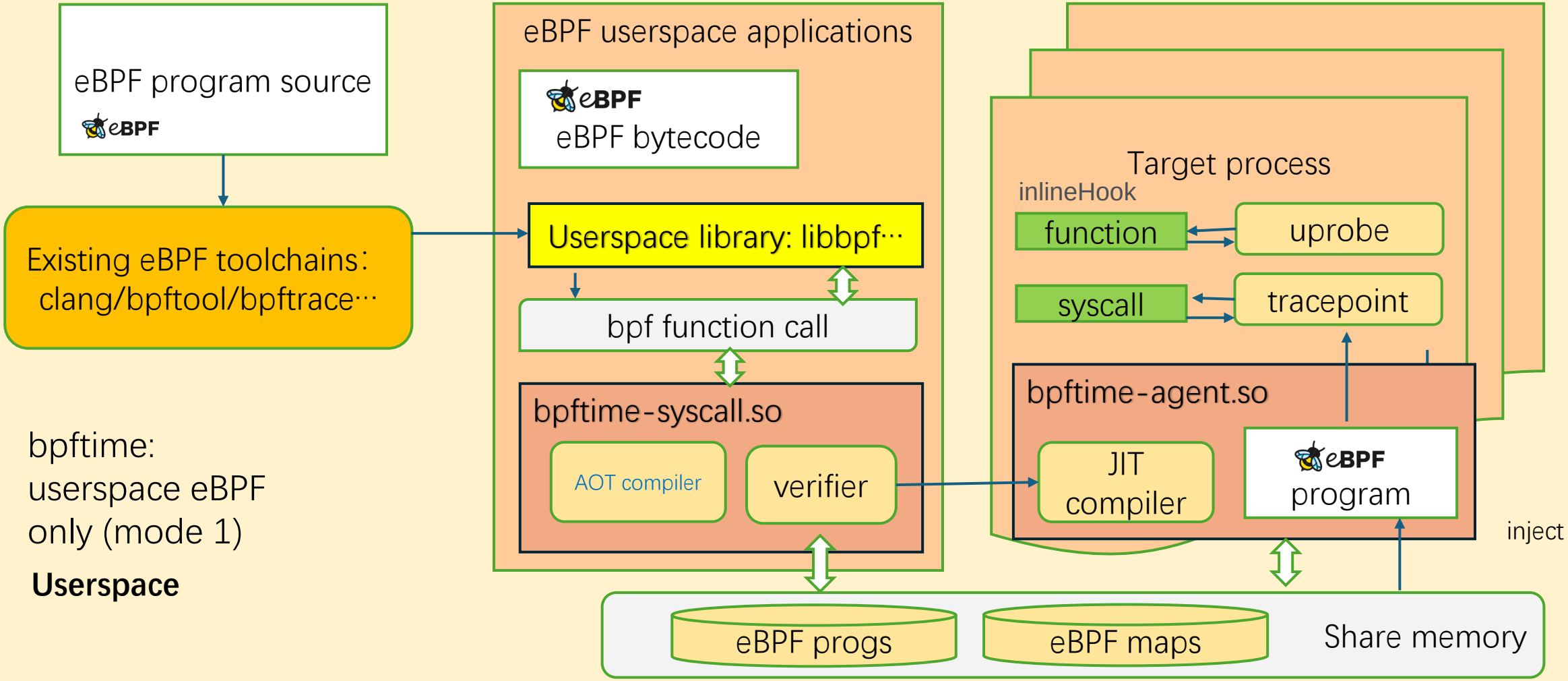
```
$ sudo ~/.bpftime/bpftime start -s example/opensnoop/victim
[2023-10-09 04:38:16.196] [info] Entering new main..
[2023-10-09 04:38:16.197] [info] Using agent /root/.bpftime/libbpftime-agent.so
[2023-10-09 04:38:16.198] [info] Page zero setted up..
[2023-10-09 04:38:16.198] [info] Rewriting executable segments..
[2023-10-09 04:38:19.260] [info] Loading dynamic library..
...
test.txt closed
Opening test.txt
test.txt opened, fd=3
Closing test.txt...
```

Run eBPF in userspace only

- Can run tools like bcc and bpftrace without modification



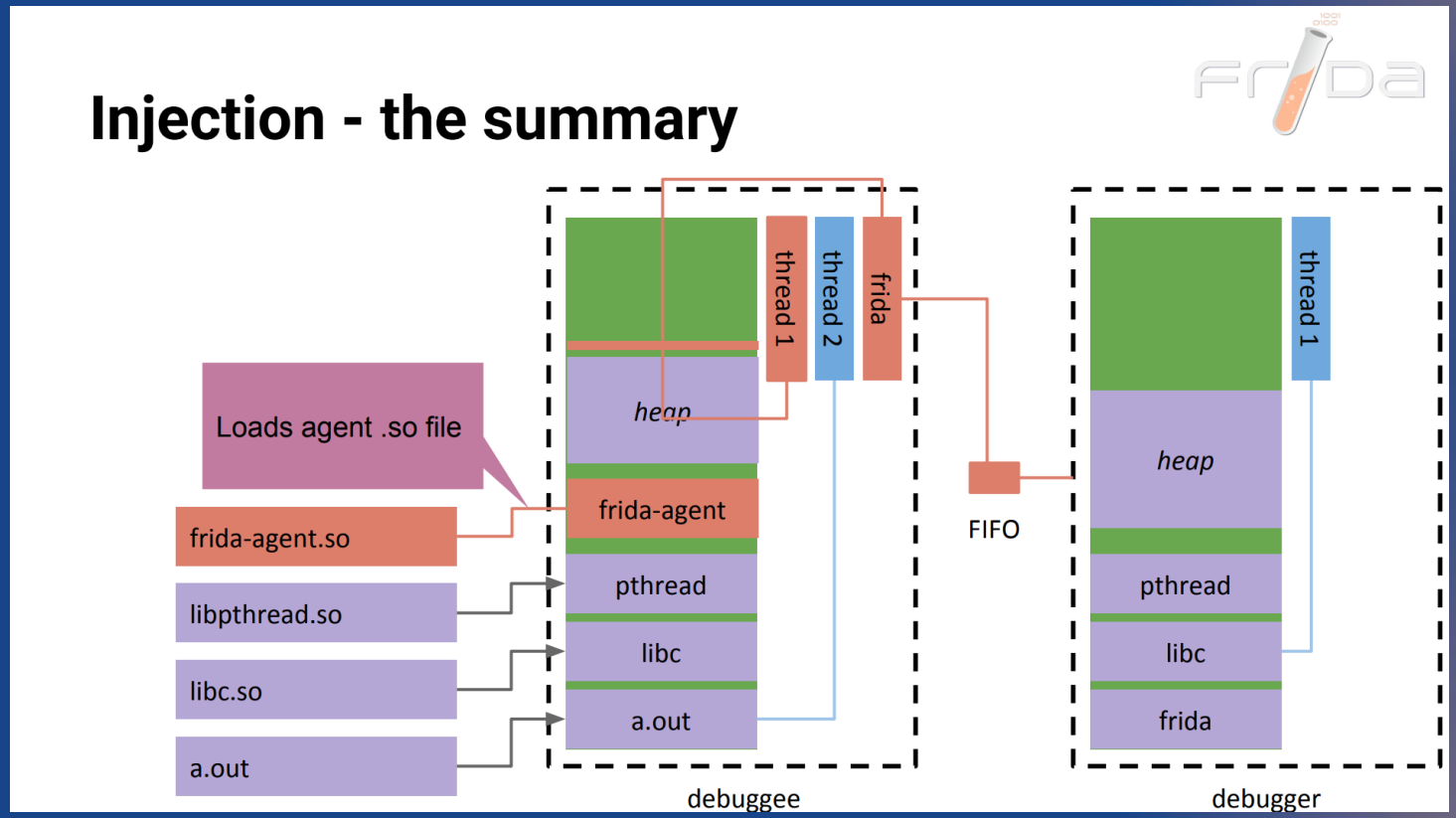
Original Kernel eBPF design: for reference


eBPF program source
eBPF
Existing eBPF toolchains:
clang/bpftool/bpftrace…
eBPF userspace applications
eBPF
eBPF bytecode
Userspace library: libbpf…
bpf function call
bpftime-syscall.so
AOT compiler
verifier
Target process
inlineHook
function
uprobe
syscall
tracepoint
bpftime-agent.so
JIT compiler
eBPF program
inject
eBPF progs
eBPF maps
Share memory
bpftime:
userspace eBPF
only (mode 1)
Userspace
Kernel space
attach
socket
kprobe

How it works: injection

Support two types of injecting runtime share library:

- For a running process: Ptrace (Based on Frida)
- At the beginning of a new process: LD_PRELOAD



How it works: userspace hook

Current hook implementation is based on binary rewriting:

- Userspace function hook: frida-gum
- Syscall hooks: zpoline and pmem/syscall_intercept.
- Can be easily extend with new trampoline methods

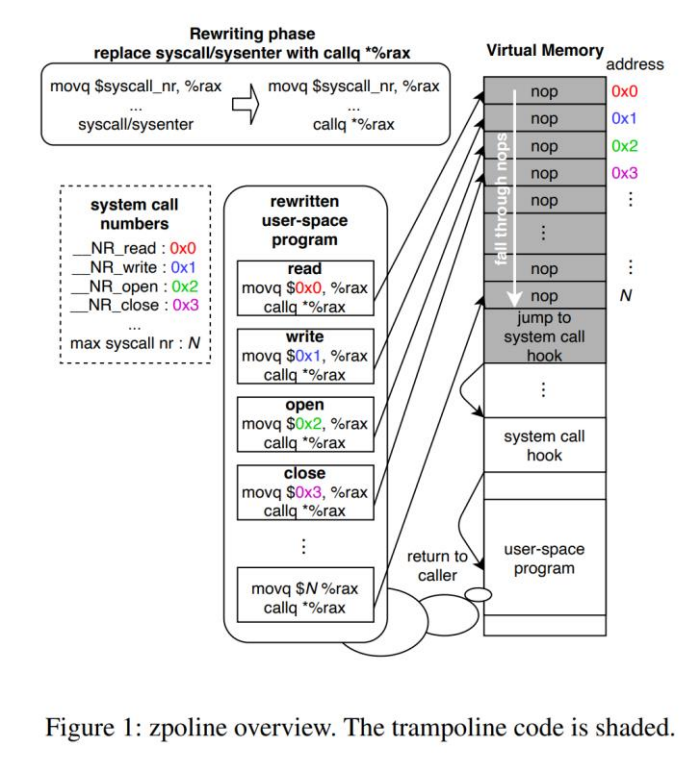
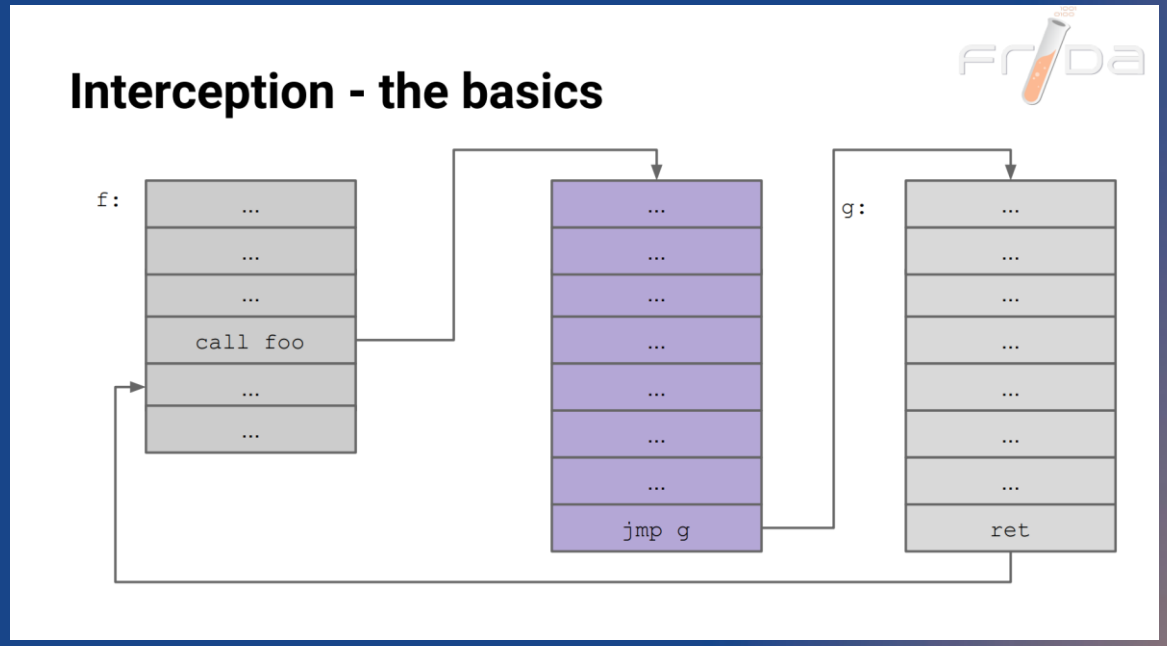
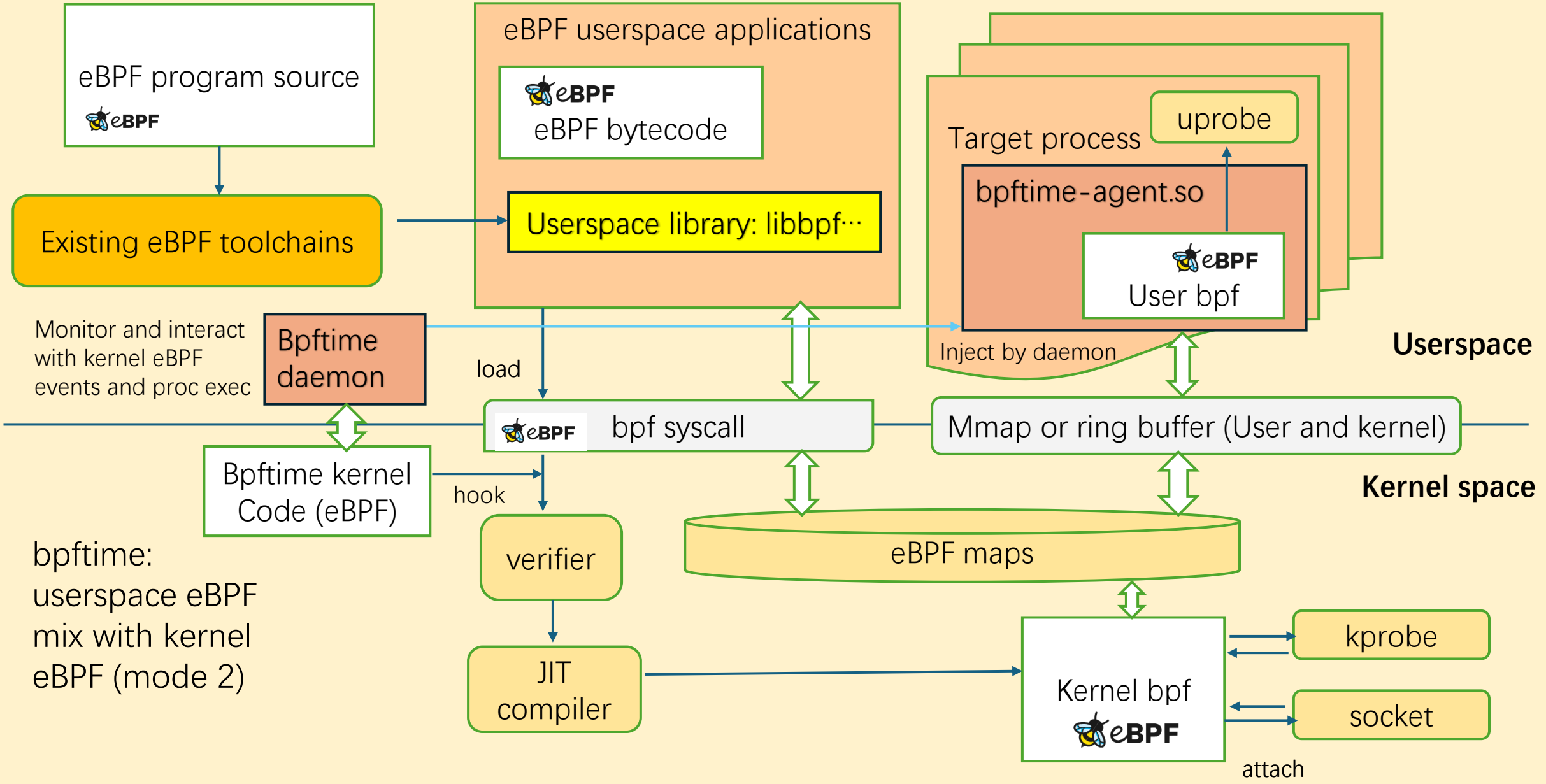


Figure 1: zpoline overview. The trampoline code is shaded.



eBPF in userspace work with kernel

- Can run complex observability agents like deepflow
- Transparently work with kernel eBPF
- Using kernel eBPF maps
- “Offload” eBPF to userspace


eBPF program source
eBPF
Existing eBPF toolchains
eBPF userspace applications
eBPF
eBPF bytecode
Userspace library: libbpf···
Target process
uprobe
bpftime-agent.so
eBPF
User bpf
Inject by daemon
Userspace
Monitor and interact with kernel eBPF events and proc exec
Bpftime daemon
load
eBPF
bpf syscall
Mmap or ring buffer (User and kernel)
Kernel space
Bpftime kernel Code (eBPF)
hook
verifier
eBPF maps
JIT compiler
Kernel bpf
eBPF
kprobe
socket
attach
bpftime: userspace eBPF mix with kernel eBPF (mode 2)

Evaluation & Cases

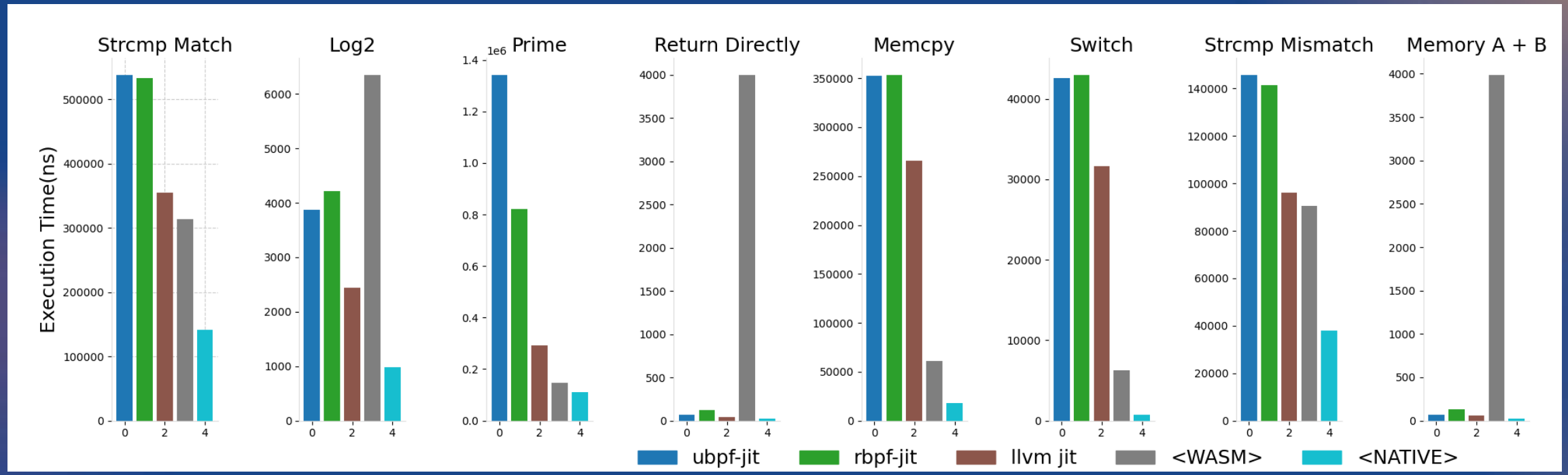
Existing eBPF use cases can be run without or with minor fixes

- bcc tools, bpftrace and ebpf_exporter
  - Bash, Memory alloc, SSL/TLS, get host latency
  - Opensnoop, Sigsnoop, syscount
- Deepflow
  - A complex Application Observability project using eBPF

Benchmark: attach overhead

How is the performance of `userspace uprobe` compared to `kernel uprobes`?

Probe/Tracepoint Types	Kernel (ns)	Userspace (ns)
Uprobe	3224.172760	314.569110
Uretprobe	3996.799580	381.270270
Syscall Hook	151.82801	232.57691
Embedding (Static Tracepoints)	Not avaliable	110.008430

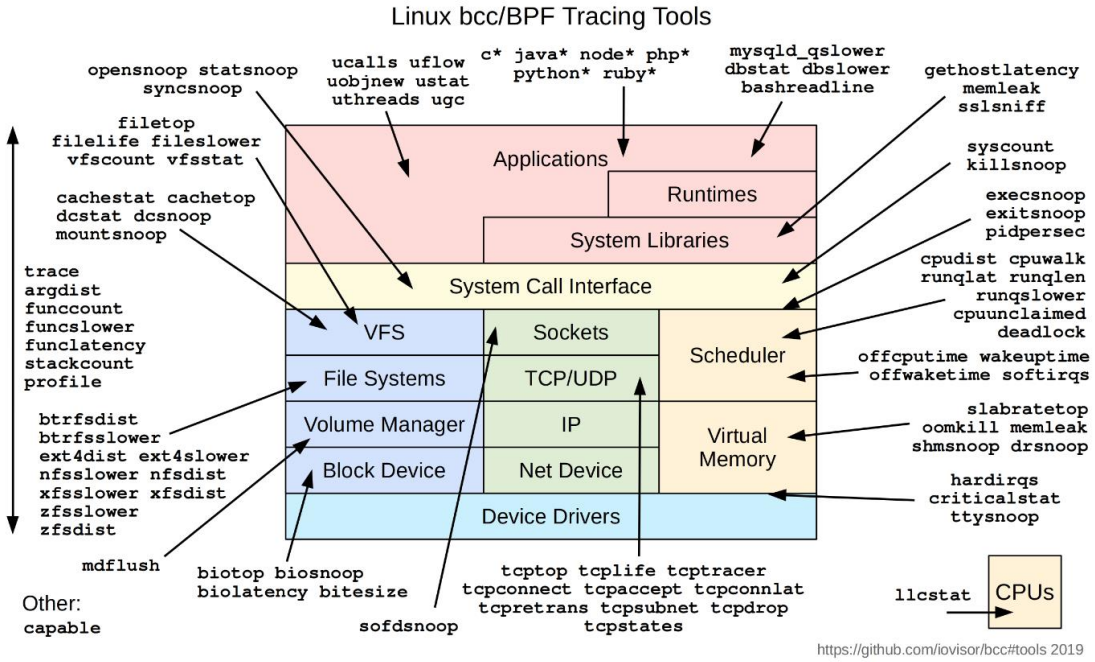


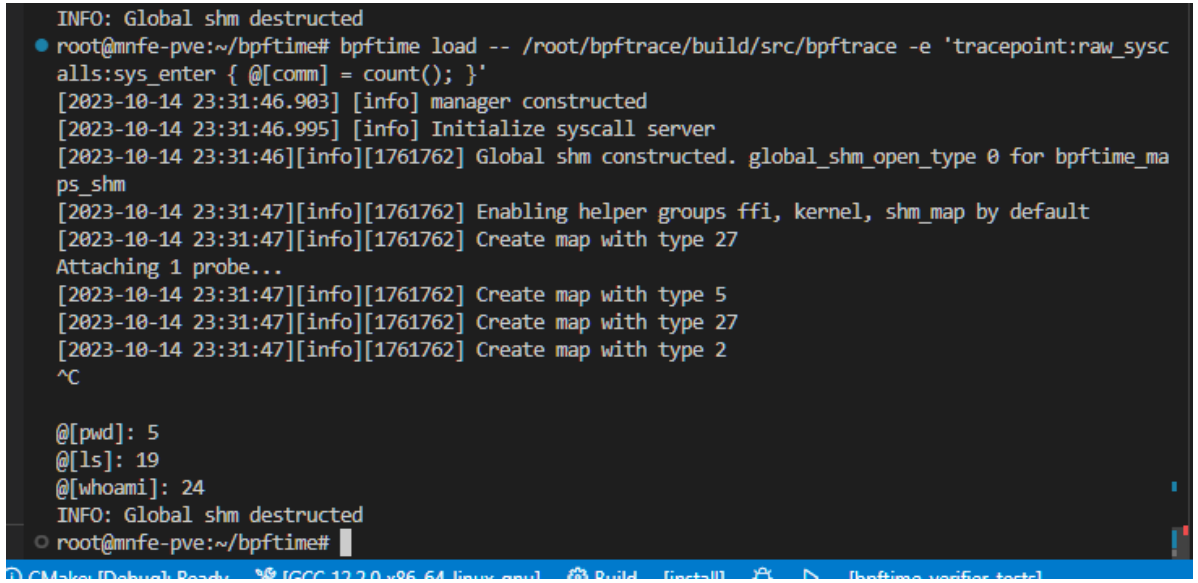
Benchmark: JIT

- LLVM jit can be the fastest
- LLVM is heavy? AOT is also support for embedding device

Bpftrace and BCC

- **Bpftrace:** can be running entirely in userspace, without kernel support eBPF, tracing syscall or uprobe
- **BCC:** the tools from top half of the picture can be run in userspace, tracing **Applications**, **Runtimes** and **System Call Interface.**
- We have ported and tested some of **bcc/libbpf-tools** and **bpftrace**, such as funclatency, bashreadline
- **Prometheus ebpf_exporter** is working as well





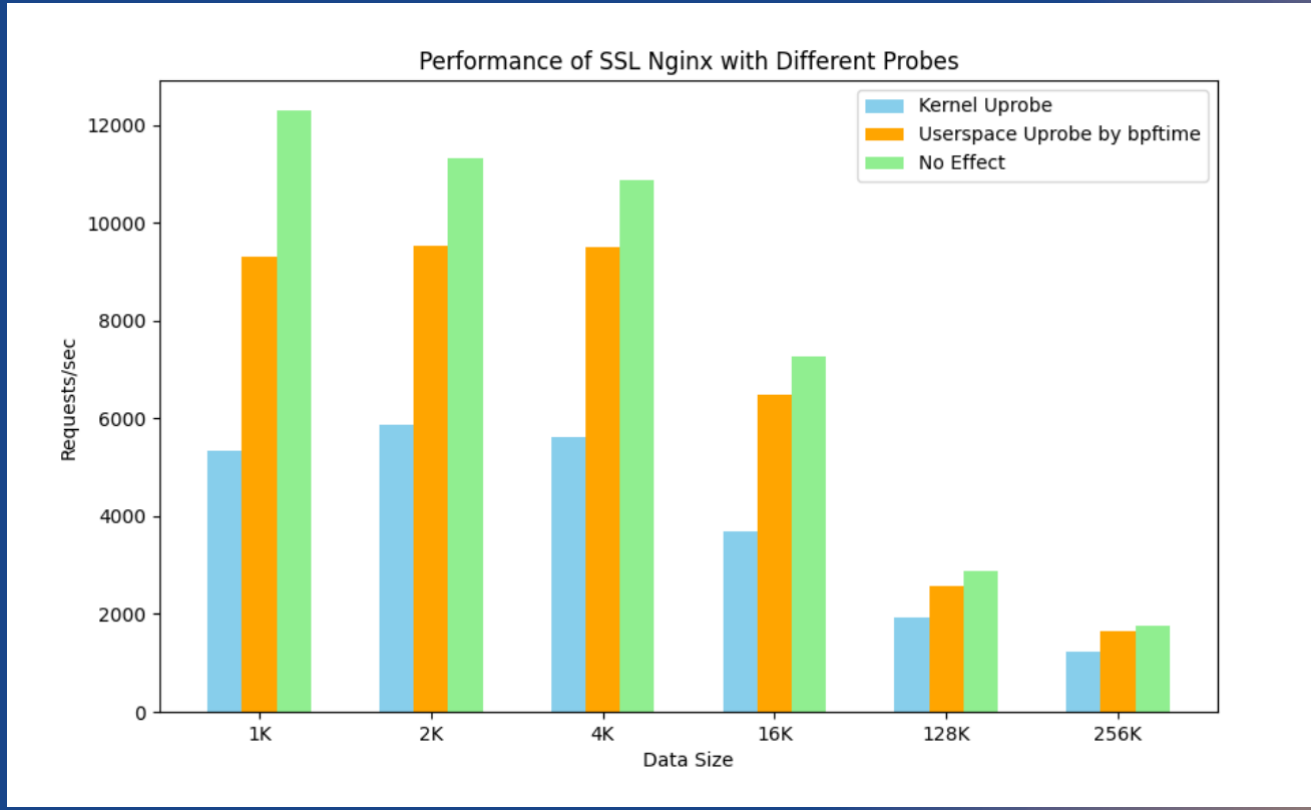
https://github.com/eunomia-bpf/bpftime/tree/master/example/bpftrace

Kernel vs. User sslsniff

sslsniff: a bcc tool to captures SSL/TLS data in userspace

Compared to no SSL interception:

- Kernel SSL Sniff reduces requests/sec by **57.98%**, transfer/sec by **58.06%**
- Userspace SSL Sniff reduces requests/sec by **12.35%**, transfer/sec by **12.30%**

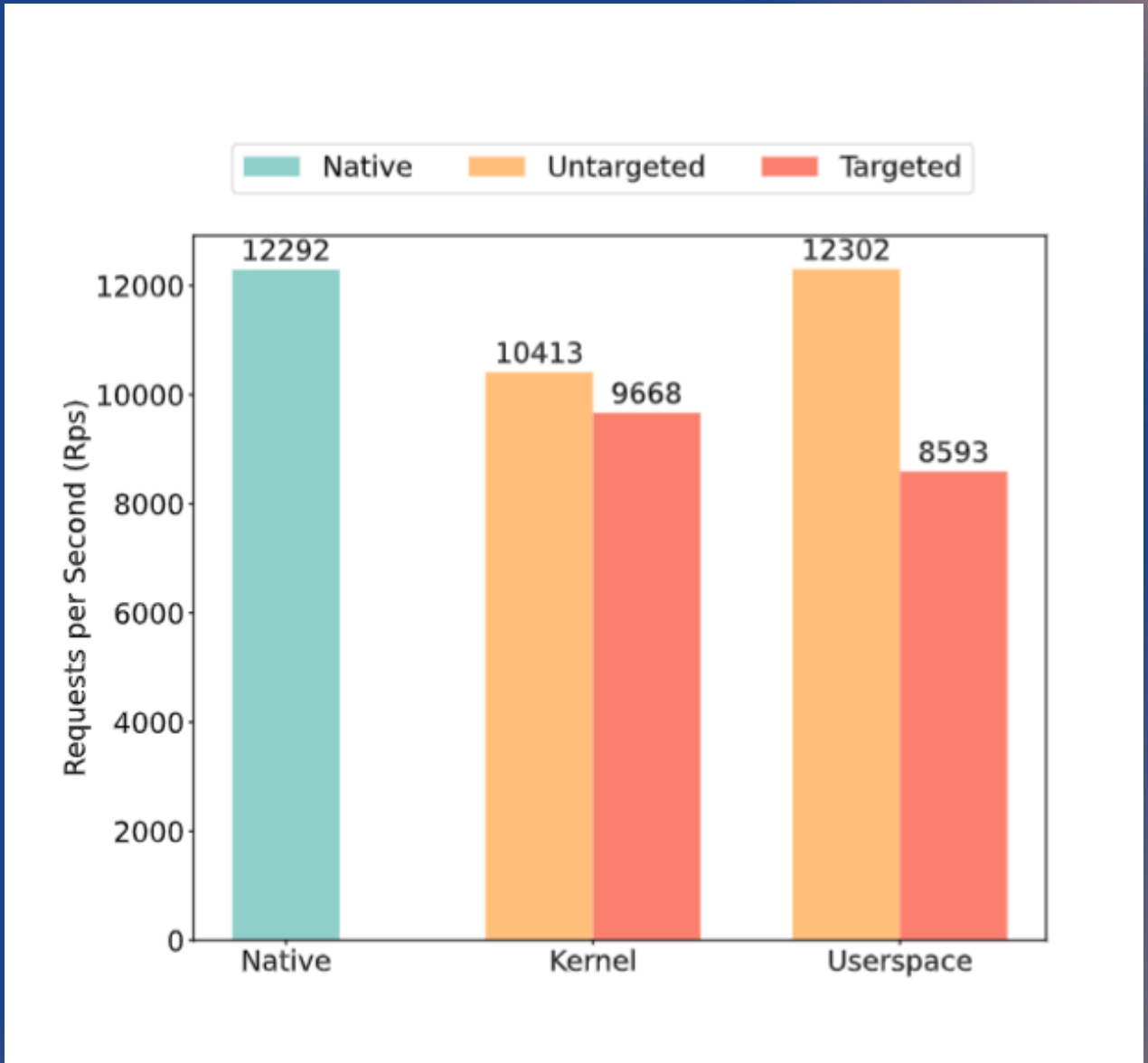


wrk https://127.0.0.1:4043/index.html -c 100 -d 10

Test Environment: Linux version 6.2.0, Nginx version 1.22.0, and wrk version 4.2.0.

Syscount

- syscount counting the systemcalls of the Nginx process, sort them and measure the latency
- https://github.com/iovisor/bcc/blob/master/tools/syscount_example.txt



Error injection or hotpatch

- Support error injection or override userspace function and syscall
- bpf_override_return

Benchmark:

- Ptrace stop application: 48ms
- LD_PRELOAD: 30ms

```
5     */
6    #define BPF_NO_GLOBAL_DATA
7    #include <vmlinux.h>
8    #include <bpf/bpf_helpers.h>
9    #include <bpf/bpf_tracing.h>
10
11   SEC("uprobe")
12   int do_error_inject_patch(struct pt_regs *ctx)
13   {
14           int rand = bpf_get_prandom_u32();
15           if (rand % 2 == 0) {
16                   bpf_printk("bpf: Inject error. Target func will not exec.\n");
17                   bpf_override_return(ctx, -1);
18                   return 0;
19           }
20           bpf_printk("bpf: Continue.\n");
21           return 0;
22   }
23
24   char LICENSE[] SEC("license") = "GPL";
```

Nginx eBPF module

- Use userspace eBPF as nginx module
- User verifier instead of sandbox, without boundary check require
- Allow eBPF to access data structs

Configuration	Througput	Overhead
Native	147000	-
eBPF security	135000	8 %
Lua security	114000	22 %
Wasm security	106000	28 %

Take away & QA

- Userspace uprobe can be **10x** faster than kernel uprobe
- Shm maps and dynamically inject into running process
- Compatible with existing eBPF toolchains, libraries, applications
- Work together with kernel eBPF

Questions? Comments? Possible new use cases?
Please tell us…

https://github.com/eunomia-bpf/bpftime

yunwei356@gmail.com

Thanks a lot!