Bpftime: Userspace eBPF runtime

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

Yusheng Zheng

yunwei356@gmail.com

Bpftime: Userspace eBPF runtime

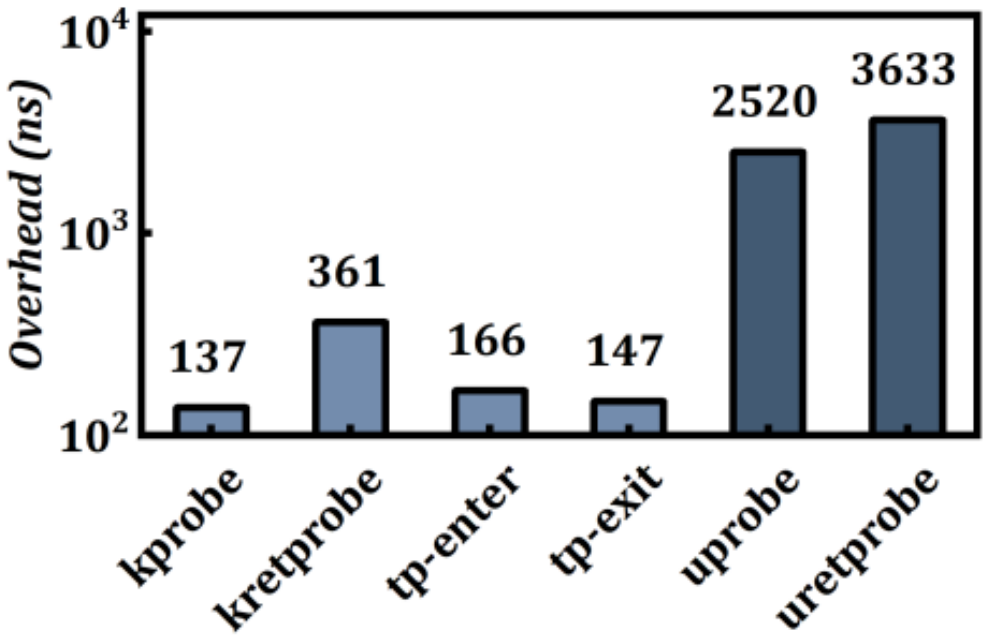
bpftime, a **full-featured**, **high-performance** eBPF runtime designed to operate in userspace：

- Fast Uprobe and Syscall hook capabilities
- Userspace uprobe can be 10x faster than kernel uprobe
  - Trace the functions or modify the function calls
- Programmatically hook all syscalls of a process safely and efficiently.
  - Monitor or Modify the syscall behavior
- Compare to:
  - Wasm in userspace
  - eBPF in kernel space
  - Other userspace eBPF runtime and toolchains
  - Other plugin systems or runtimes

Motivation

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.



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...

Motivation

2. Kernel eBPF Security Concerns and limited configurable

- 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.
- Verifier has limited the operation of eBPF, config requires kernel change

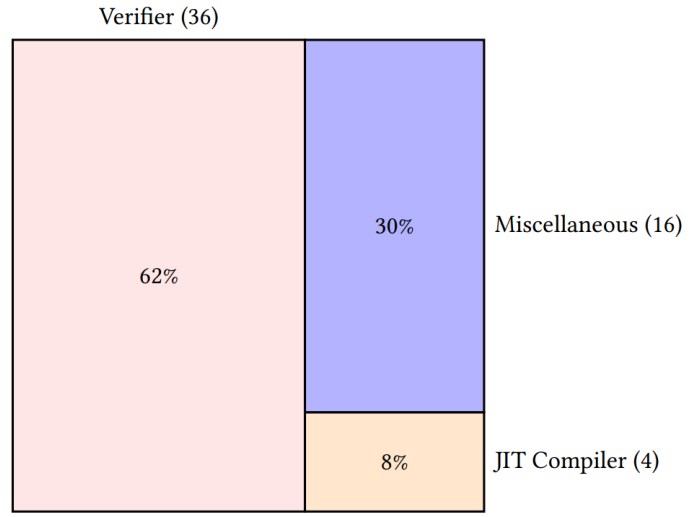


Figure 1: A tally of eBPF-related CVEs from 2010 to 2023. There are a total of 56 CVEs, the majority of which were discovered in the verifier.

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

Motivation

3. Wasm Runtime or other plugin systems Limitations

- Manual integration needed, making it less adaptable to API version changes.
- Relies on underlying libraries for complex operations, e.g., Wasi-nn.
- Security concerns with external APIs like Wasi require additional validation and runtime checks, leading to high performance costs.

eBPF: performance first and security second, use verifier for security

Wasm: puts more emphasis on security at the cost of some runtime overheads, use SFI for security

Current eBPF userspace Runtimes

- **Ubpf**: ELF parsing, simple hash map, arm64, x86 JIT.
- **Rbpf**: Helper mechanism, x86 JIT, VM. GitHub.
- **Drawbacks**:
  - Complex integration and usage, cannot use kernel eBPF library and toolchains, e.g. libbpf/bpftrace/clang
  - No attach support. No interprocess maps.
  - Limited functionality in userspace.
  - JIT supports for only arm64 or x86

Existing Non-linux 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 + hotpatch + **Windows**

Features of bpftime

- Run **eBPF in userspace** just like in the kernel
- Achieve **10x speedup** vs. kernel uprobes.
- Use **shared eBPF maps** for data & control.
- **Compatible** with clang, libbpf, and existing eBPF toolchains; supports CO-RE & BTF.
- Support **external functions(ffi)** and pointers
- Includes **cross-platform interpreter** & fast **LLVM JIT compiler** & handcraft **x86 JIT in C** for limited resources, Near to native speed
- **Inject** eBPF runtime to Any running Process without restart or manually recompile
- Runing **not only in Linux**: all unix system and windows, even Iot devices

Examples

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

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

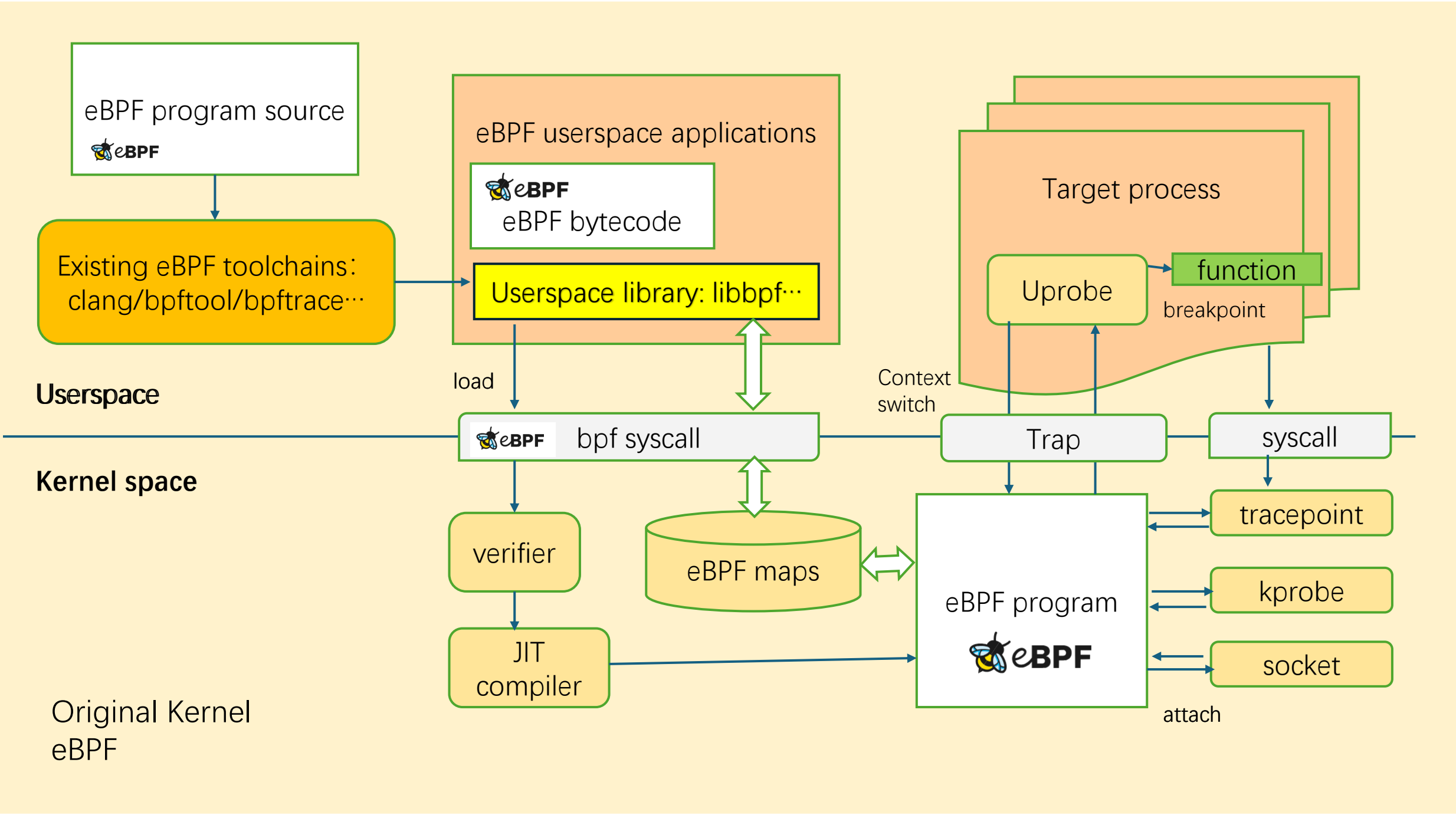
Use syscall tracepoint to monitor open and close syscall, with ring buffer for output

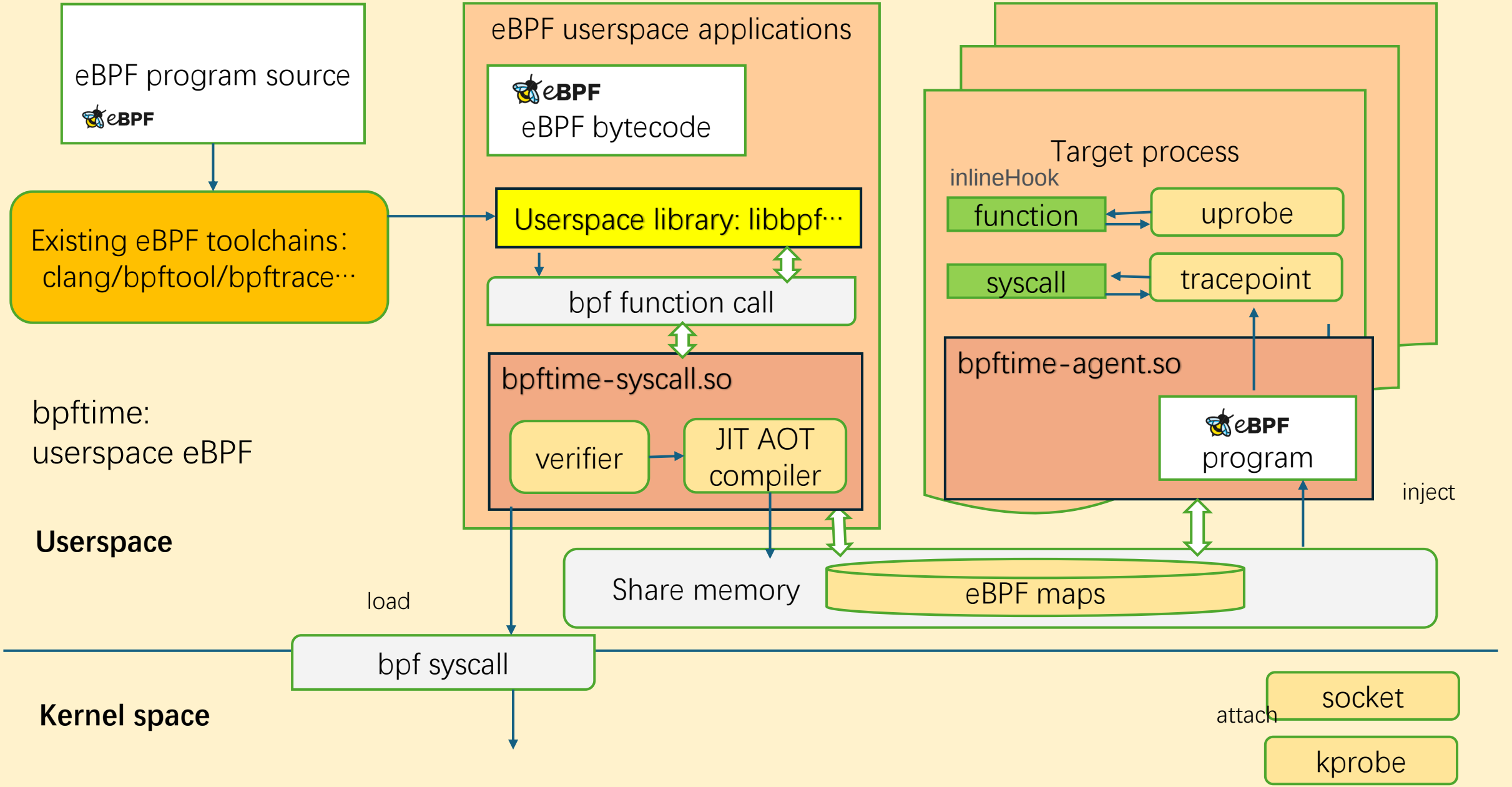
Usage

```
$ 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...
```


eBPF program source
eBPF
Existing eBPF toolchains:
clang/bpftool/bpftrace…
eBPF userspace applications
eBPF
eBPF bytecode
Userspace library: libbpf…
load
Userspace
Kernel space
eBPF
bpf syscall
verifier
JIT
compiler
eBPF maps
Target process
Uprobe
function
breakpoint
Context
switch
Trap
syscall
tracepoint
kprobe
socket
eBPF program
eBPF
attach
Original Kernel
eBPF


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
verifier
JIT AOT
compiler
Target process
inlineHook
function
uprobe
syscall
tracepoint
bpftime-agent.so
eBPF
program
inject
bpftime:
userspace eBPF
Userspace
load
Share memory
eBPF maps
bpf syscall
Kernel space
attach
socket
kprobe

Benchmark

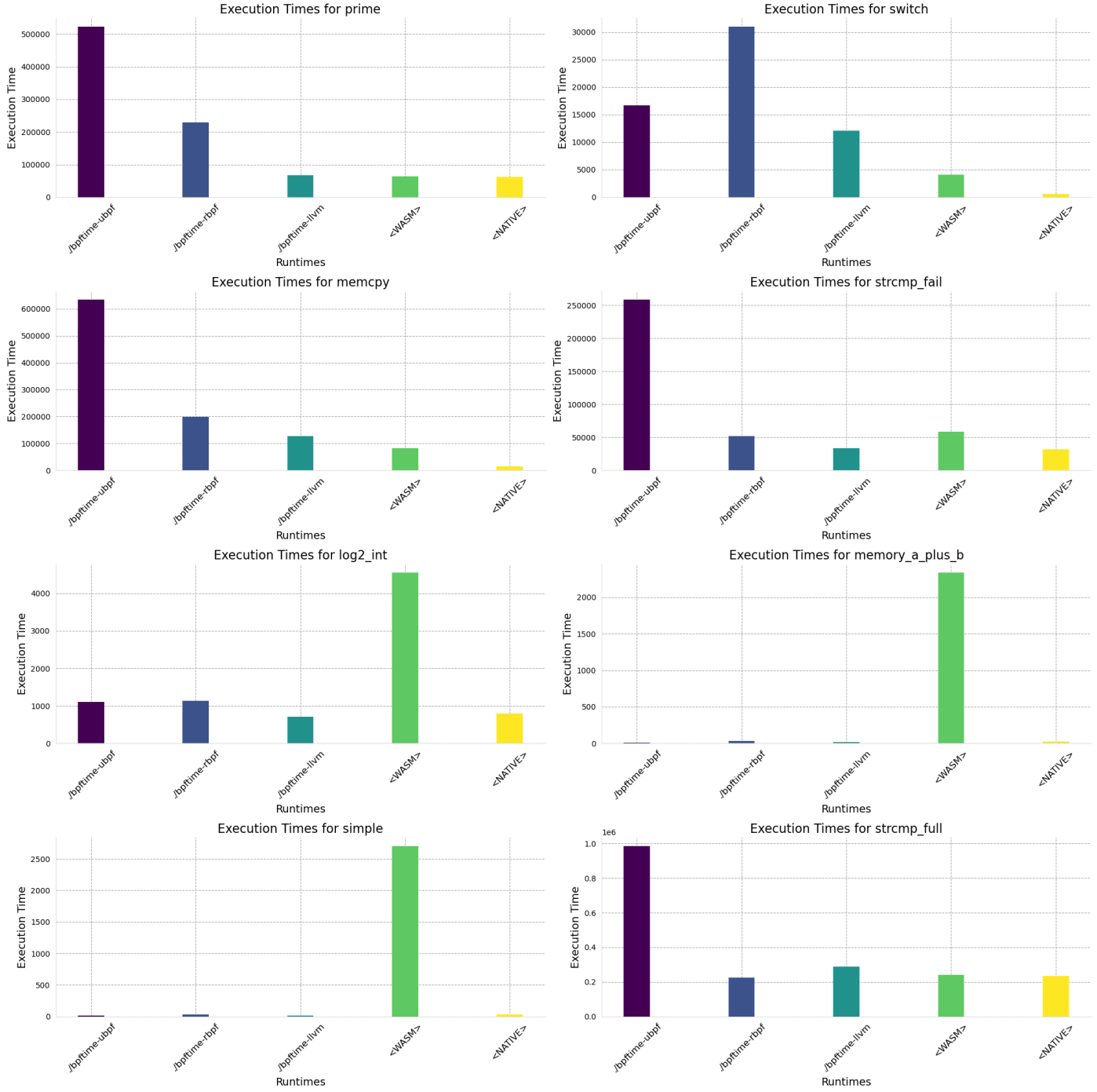
How is the performance of `userspace uprobe` compared to `kernel uprobes`?

Probe/Tracepoint Types	Kernel (ns)	Userspace (ns)	Insn Count
Uprobe	3224.172760	314.569110	4
Uretprobe	3996.799580	381.270270	2
Syscall Tracepoint	151.82801	232.57691	4
Embedding runtime	Not avaliable	110.008430	4

Benchmark

Exec time: Left to right: ubpf jit, rbpf jit, llvm jit/aot, wasm, native

- LLVM jit can be the fastest
- LLVM is heavy? AOT is on the way



More is coming…

- Figure out how to run it together transparently with kernel probe
- An AOT compiler for eBPF can be easily added based on the LLVM IR.
- More examples and usecases.
- …

This project is open-sourced by https://github.com/eunomia-bpf/bpftime

bpf-developer-tutorial

- https://github.com/eunomia-bpf/bpf-developer-tutorial
- 1.1k+ stars in Github
- 10W+ readings on various Platform: zhihu, wechat, juejin…

AI for eBPF code Generation

- https://gpt-2-bpftrace.vercel.app/
- https://github.com/eunomia-bpf/GPTtrace
- KEN project （Not yet open-sourced）
- GPT4 can generate eBPF code up to 80% correct with our AI agents

Table 2. Comparison of different with fine-tuned vectorDB

Type	TP %	FP %	FN %
GPT3.5-Base	42.5%	12.5%	45%
GPT3.5-3Trails	57.5%	10%	32.5%
GPT4-Base	40%	5%	55%
GPT4-3Trails	65%	5%	30%
GPT4-HumanInfo3Trials	72.5%	2.5%	25%
GPT4-SMT3Trials	80%	2.5%	17.5%
CodeLLama-Base	22.5%	7.5%	65%
CodeLLama-3Trails	42.5%	7.5%	50%
CodeLLama-HumanInfo3Trials	45%	2.5%	52.5%
CodeLLama-SMT3Trials	40%	2.5%	57.5%