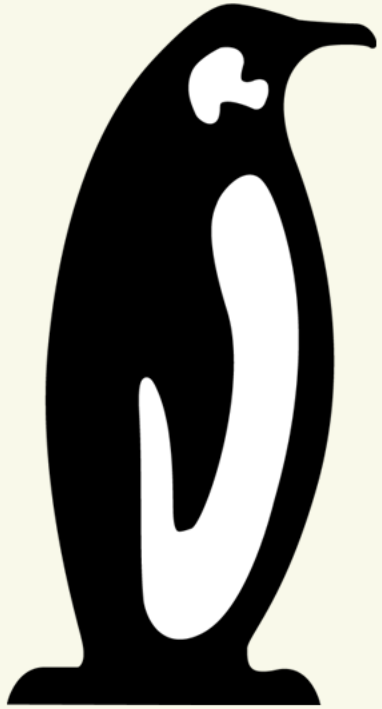


Linux Plumbers Conference

Richmond, Virginia | November 13-15, 2023

Bpftime: Userspace eBPF runtime

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

Yusheng Zheng

yunwei356@gmail.com

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?

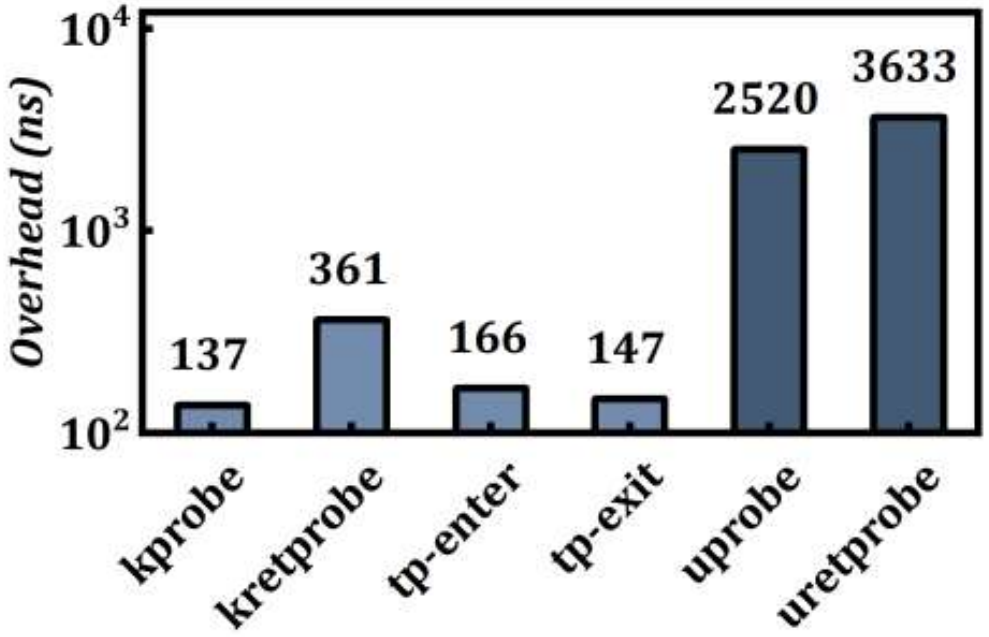
Uprobe: User-level dynamic tracing

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.

Kernel eBPF limited configurable

- Verifier has limited the operation of eBPF, config eBPF or make it Turing-complete requires kernel change
- Add new helper or new feature also requires kernel change

Why bpftime?

3. Current userspace eBPF runtime Limitations

Possible user space eBPF usecases:

- User space observability
- User space network
- User space Configuration, plugins and filters

Cannot run **workloads** in current eBPF ecosystem with existing userspace eBPF

Existing userspace eBPF

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

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 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 compiler for eBPF

Current support features

Userspace eBPF shared memory map types:

- BPF_MAP_TYPE_HASH
- BPF_MAP_TYPE_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:*

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

Support **22 kernel helper functions**

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

Test JIT with **bpf_conformance**

Uprobe and kprobe mix: 2 modes

- **Run eBPF in userspace only (mode 1)**
  - 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 (mode 2)**
  - 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

Examples

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

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

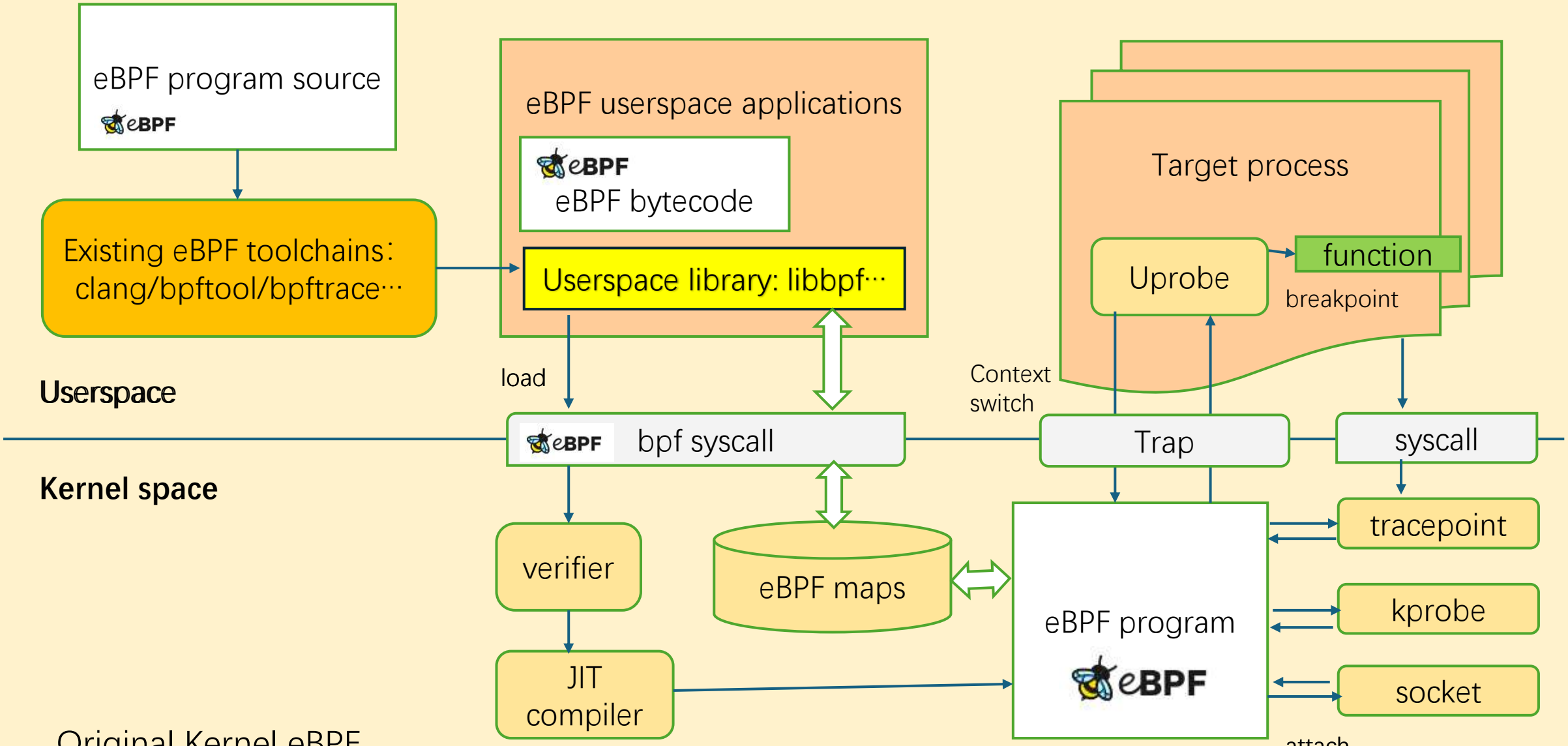
```
$ sudo example/malloc/victim
malloc called from pid 12314
continue malloc...
```

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

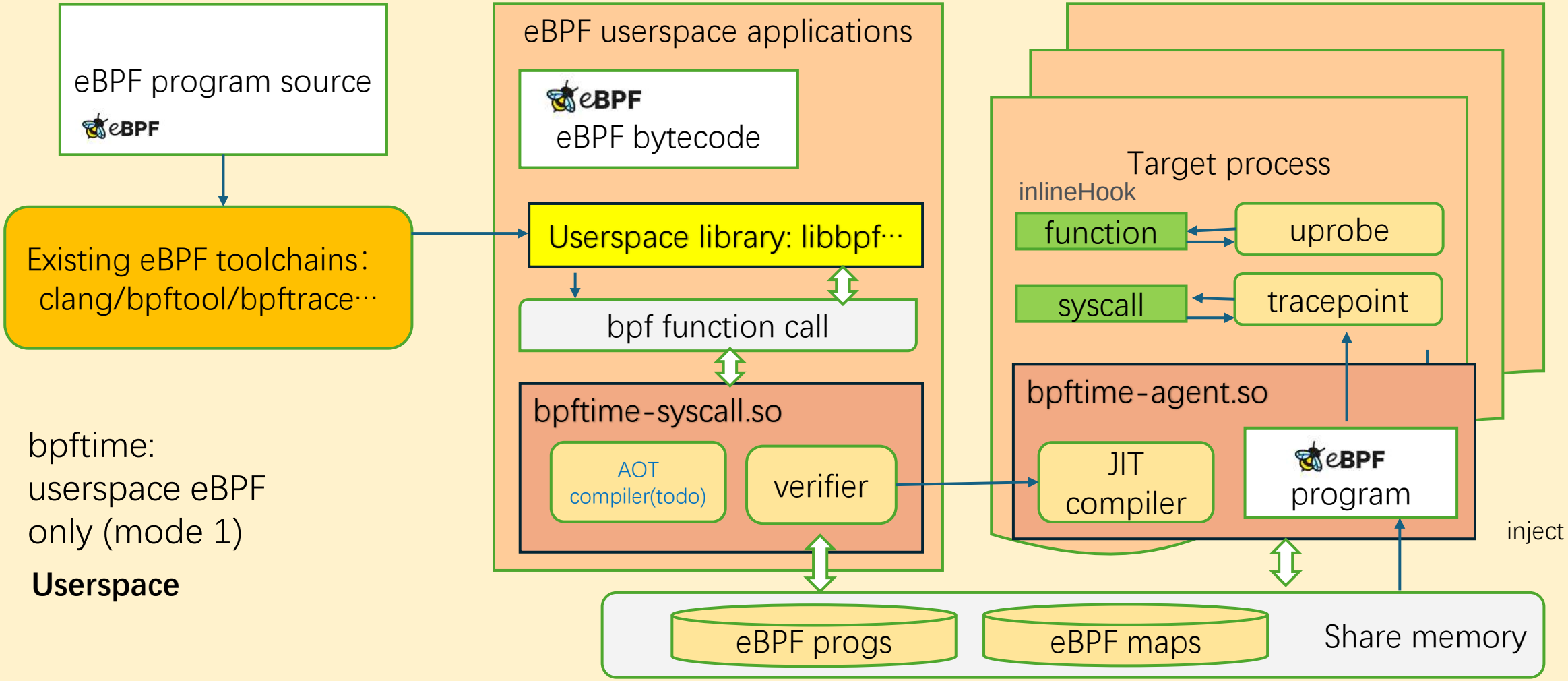
```
20:43:22
        pid=113413      malloc calls: 9
```

Mode 1: Run eBPF in userspace only

- Can run tools like bcc and bpftrace without modification



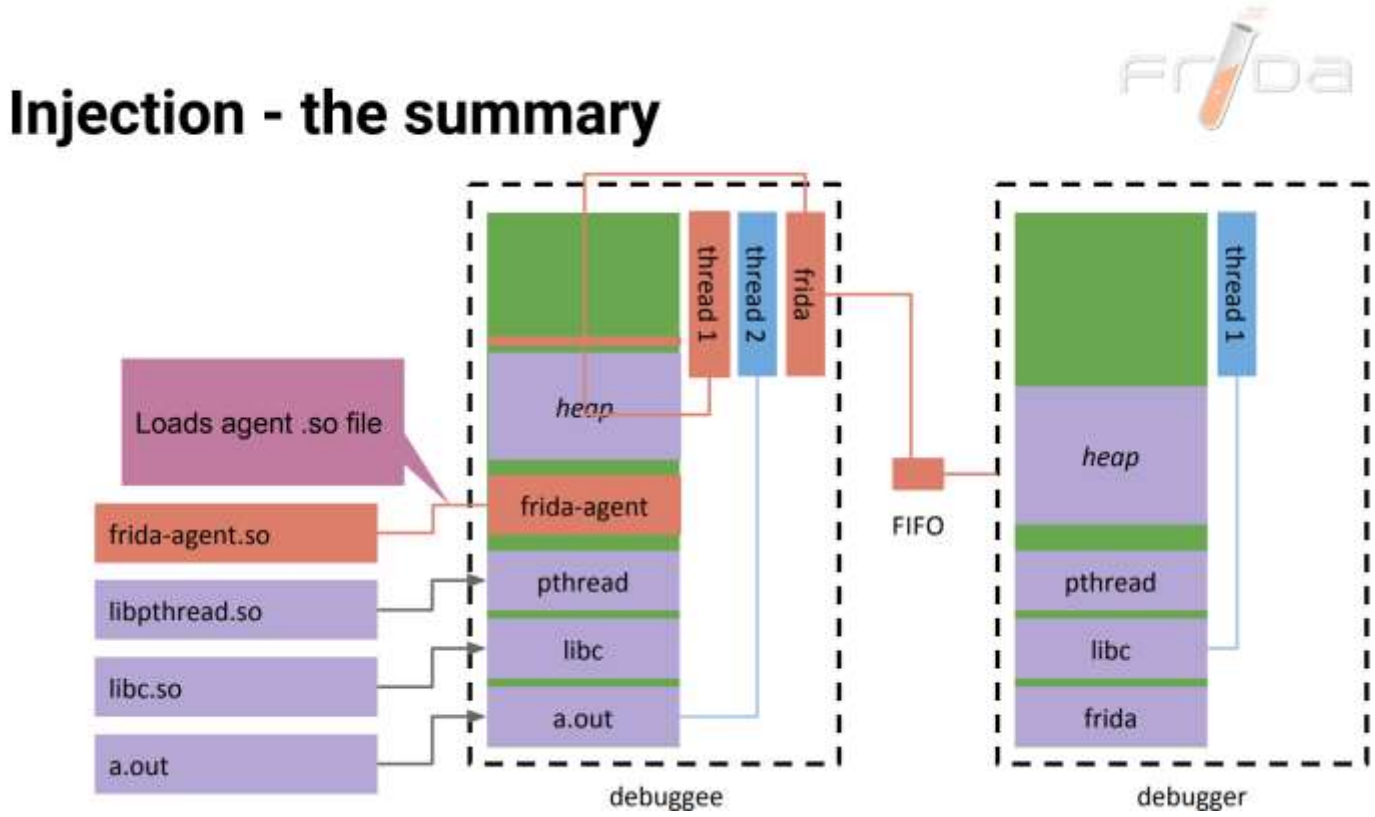
Original Kernel eBPF design: for reference


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(todo)
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: trampoline

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

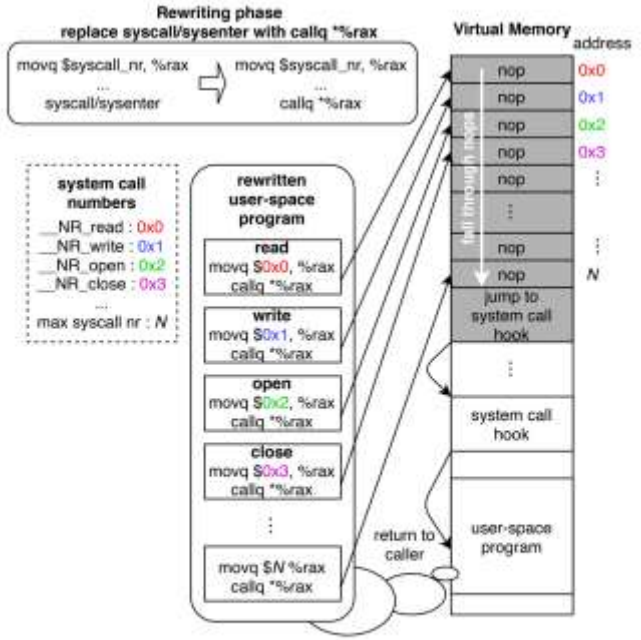
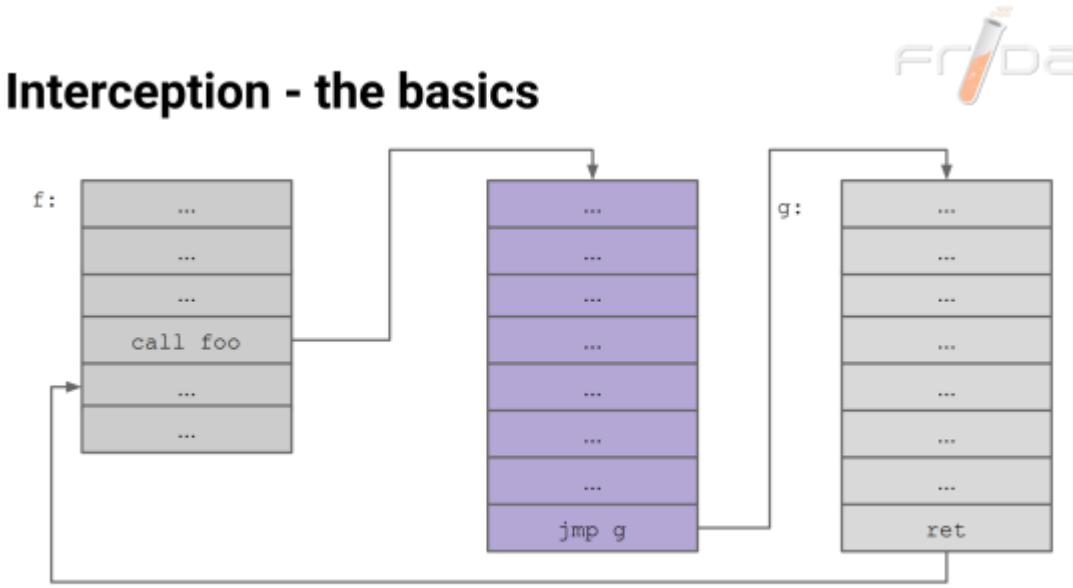
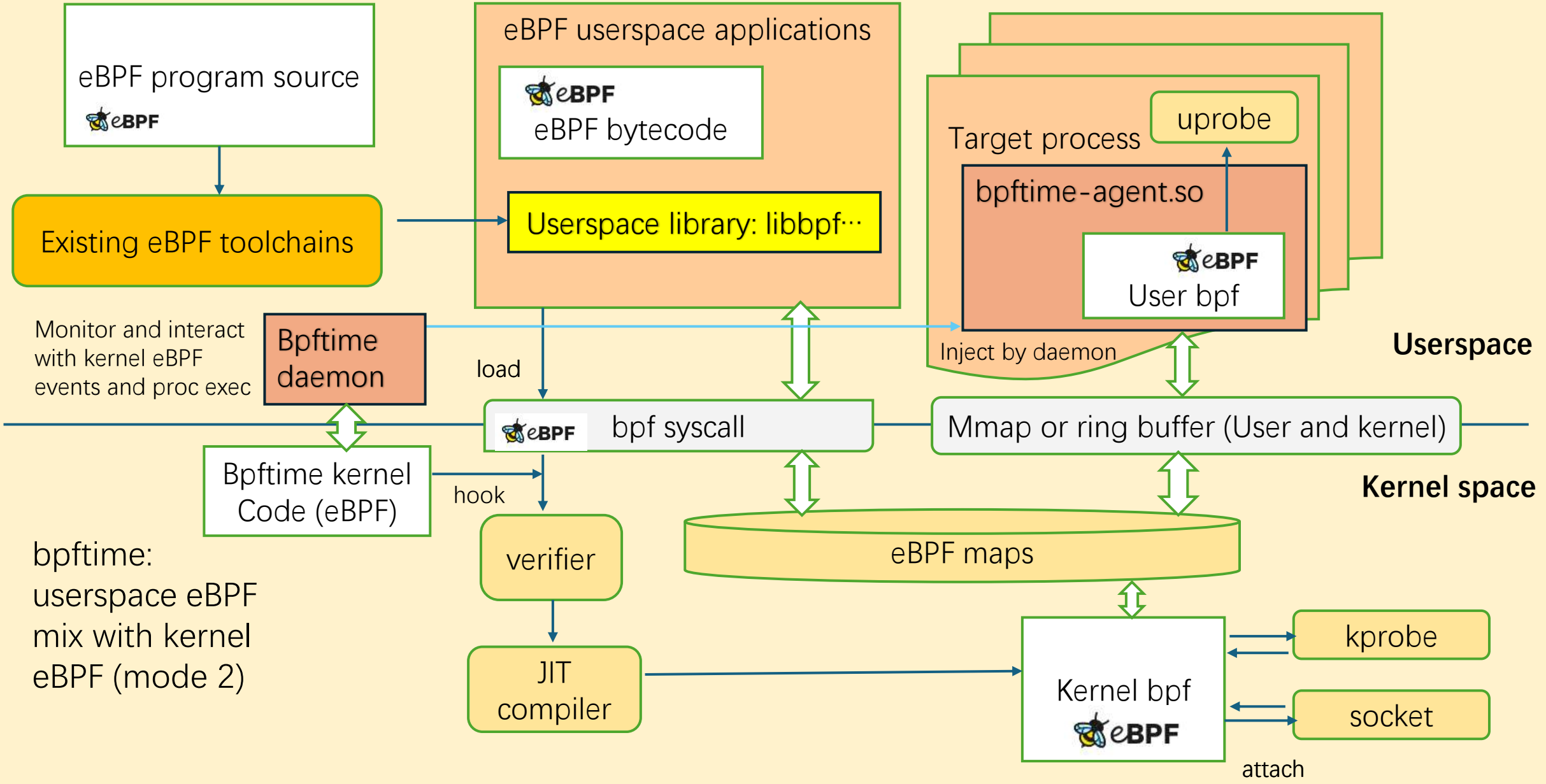


Figure 1: zpoline overview. The trampoline code is shaded.



Mode 2: eBPF in userspace work with kernel

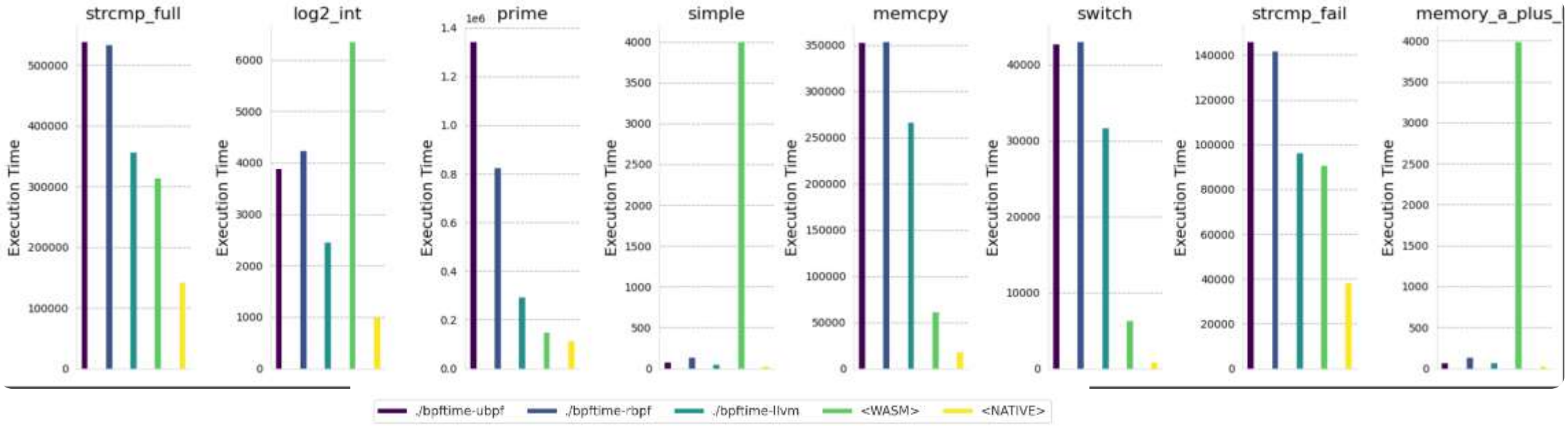
- Can run complex observability agents like deepflow
- Transparently work with kernel eBPF
- Using kernel eBPF maps
- “Offload” eBPF to userspace


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)

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 on the way

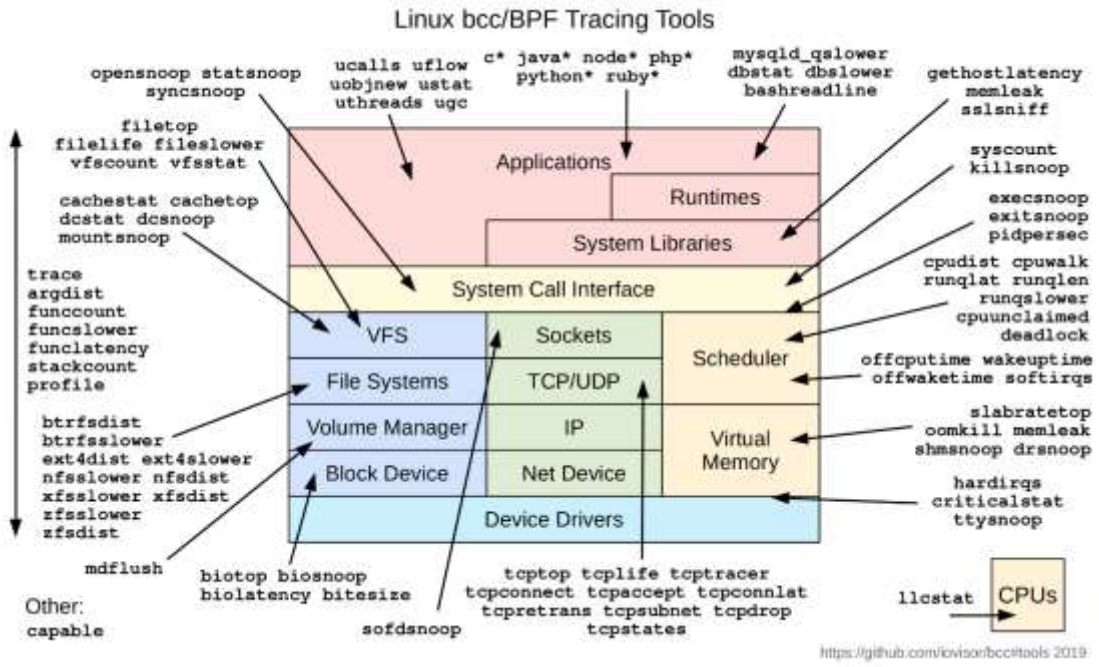
Evaluation & Cases

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

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**
- **Prometheus ebpf_exporter** is working as well




```
INFO: Global shm destructed
root@mnfe-pve:~/bpftime# bpftime load -- /root/bpftrace/build/src/bpftrace -e 'tracepoint:raw_sysc
alls:sys_enter { @[comm] = count(); }'
[2023-10-14 23:31:46.903] [info] manager constructed
[2023-10-14 23:31:46.995] [info] Initialize syscall server
[2023-10-14 23:31:46][info][1761762] Global shm constructed. global_shm_open_type 0 for bpftime_ma
ps_shm
[2023-10-14 23:31:47][info][1761762] Enabling helper groups ffi, kernel, shm_map by default
[2023-10-14 23:31:47][info][1761762] Create map with type 27
Attaching 1 probe...
[2023-10-14 23:31:47][info][1761762] Create map with type 5
[2023-10-14 23:31:47][info][1761762] Create map with type 27
[2023-10-14 23:31:47][info][1761762] Create map with type 2
^C

@[pwd]: 5
@[ls]: 19
@[whoami]: 24
INFO: Global shm destructed
root@mnfe-pve:~/bpftime#
```

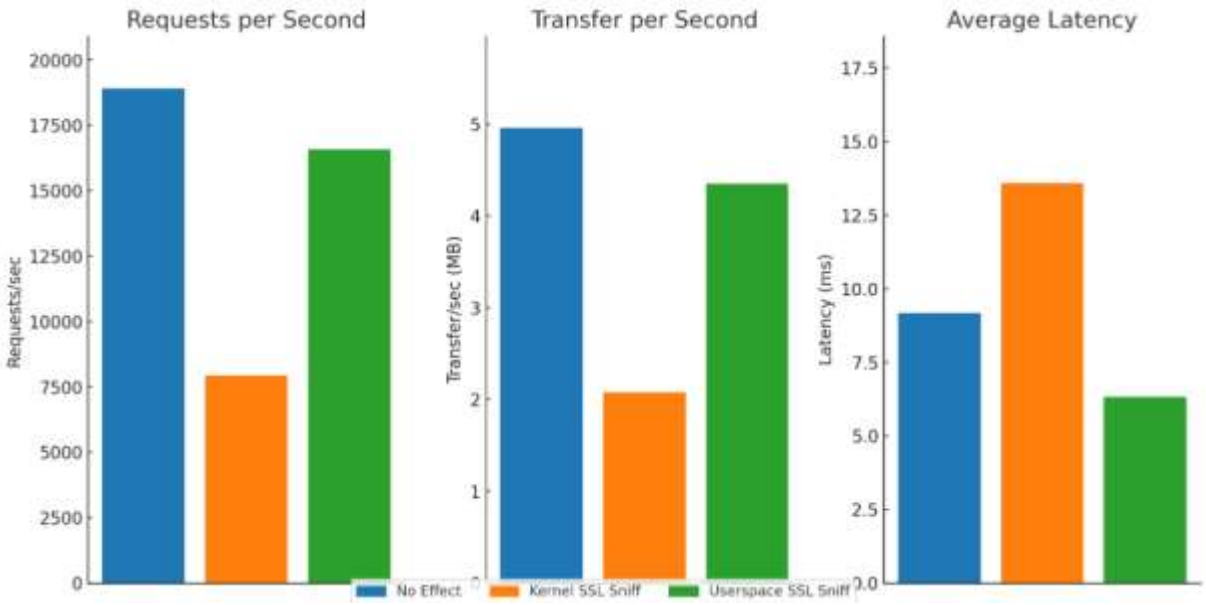
https://github.com/eunomia-bpf/bpftime/tree/master/example/bpftrace

Kernel vs. User SSLSniff on Nginx

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%**



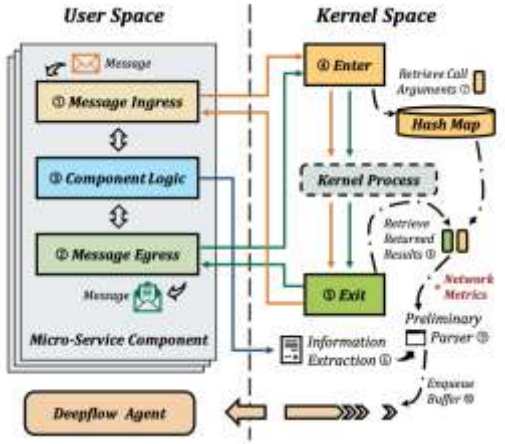
wrk https://127.0.0.1:4043/index.html -c 100 -d 10

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

Deepflow: a complex workload

- Application Observability using eBPF
- 5k+ LOC of kernel eBPF code, uprobe, kprobe, socket, and tracepoints work together
- Deployed in production and published in SIGCOMM 23
- Uprobe in L7 observability may be slow:
  - Userspace Uprobe:
    - Reduces requests/sec by 15.93%
    - Reduces transfer/sec by about 15.88%
  - Kernel Uprobe:
    - Reduces requests/sec by approximately 21.99%.
    - Reduces transfer/sec by about 21.96%.

*Test with all features enabled, golang http server with goroutine tracing



Roadmaps

Possible new usecases:

- Network related eBPF in userspace
  - Currently userspace eBPF can be used in DPDK, but No Control Plane for it
  - Programable userspace network stack, with existing eBPF Applications
- Use userspace eBPF to speed up fuse
  - Android or fuse for Cloud Storage
  - Filter in userspace
- Hotpatch userspace functions

Any new ideas?

Roadmaps

Improvements:

- More benchmarks and evaluations
- Make it works better with kernel eBPF
  - Improve compatibility: more maps and helpers support
- Performance optimize for LLVM JIT and runtime
- LLVM AOT compile eBPF for resource constrains environments
- Make sure the eBPF is not attacked
- More tests, CI and cleaner code

Open problems

- BPF_F_MMAP currently only for arrays, how to make a better-performance hash map shared between kernel and user space?
  - Introduce new hash map types?
  - Implement a basic hash map on top of array map?
  - Let kernel eBPF prog access userspace maps?
  - Use cache and sync them with syscall?
- Error propagation: can kernel eBPF wait for userspace operations?
- Unprivileged eBPF type?
- Security models?
- …

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

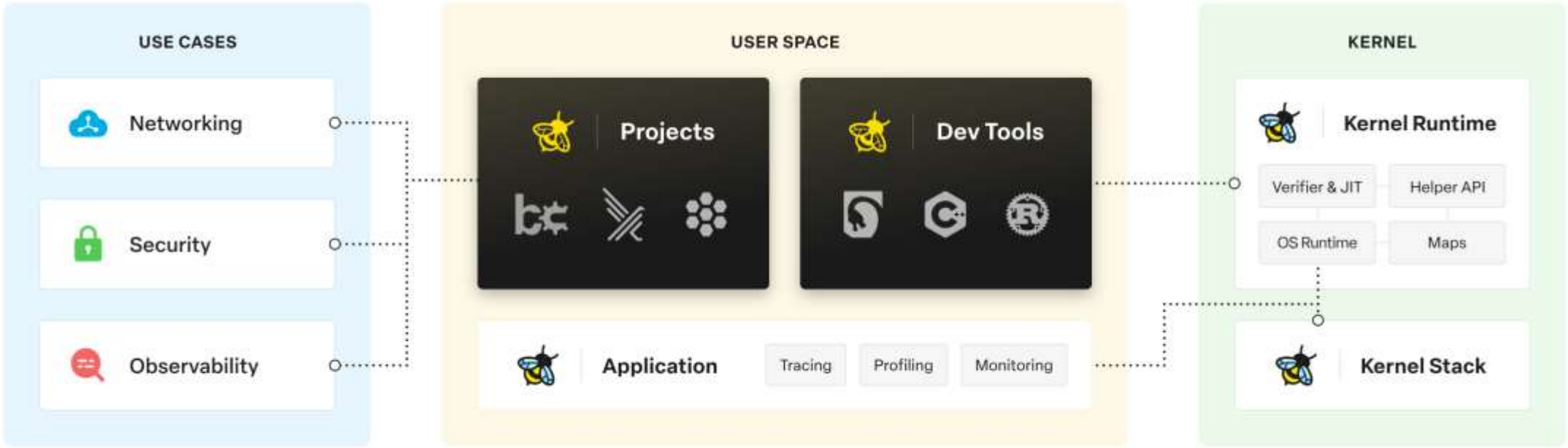
yunwei356@gmail.com

Thanks a lot!

Backup

eBPF

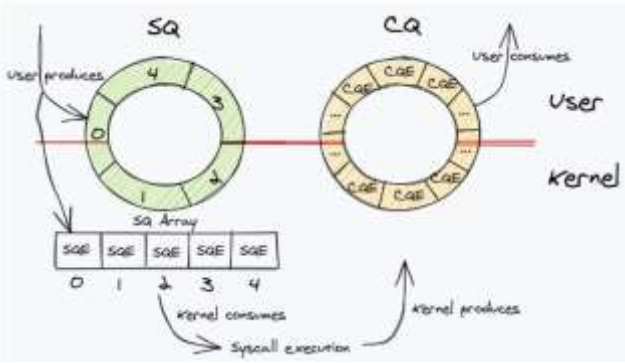
Dynamically and safely program the kernel for efficient networking, observability, tracing, and security



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.
- Includes cross-platform **interpreter** & Near native speed **LLVM JIT compiler,** support using ubpf JIT alternative
- **Inject** eBPF runtime to Any running Process without restart or manually recompile
- Working **together** with kernel eBPF maps, support “offload” and run from kernel

Motivation



4. Syscall may be slow, can we change how the kernel-user interaction works by user and kernel eBPF?

eBPF maps can work cross boundary and bridge the userspace and kernel, without syscall overhead:

- BPF_F_MMAP for share memory between kernel and userspace
- eBPF ring buffer and user ring buffer: similar to iouring

eBPF programs can patch kernel and userspace dynamically

Why not Wasm?

Why not Wasm? Different usecases

eBPF: performance first, use verifier for security

Wasm: security first, use SFI for security

- Wasi or eBPF Relies on underlying libraries for complex operations, e.g., Wasi-nn.
- Wasi for Wasm require additional validation and runtime checks, leading to high performance costs.
- Manual integration needed, making it less adaptable to API version changes.

Why not DBI tools?

There exists a lot of DBI tools, Frida, pin, etc…

- Traditional DBI tools use sandbox for isolation, eBPF use **verifier**
- **eBPF** can access deep data structs with pointers in the applications, without runtime checks
- **eBPF** can relocation between difference userspace application versions (CO-RE)
- **eBPF** can summarize data from multiple processes, both user and kernel at runtime
- A **large community** and growing **ecosystem**

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...
```

Design goals

1. **Enhanced Performance and Flexibility**:
   Enable faster and more flexible execution of eBPF programs within userspace.
2. **Toolchain Compatibility**:
   Ensure seamless integration with existing eBPF toolchains like clang and libbpf.
3. **Transparent Execution of Complex Workloads**:
   Support efficient execution of real-world complex eBPF workloads using userspace uprobe, support running userspace eBPF together with kernel eBPF
4. **Safety and Security**:
   Use kernel or userspace verifier to make sure the eBPF will not break userspace App.
5. **Non-intrusive Integration**:
   Enable integration without kernel changes, or manual intervention on the userspace side.

Challenges

- Userspace **libraries** and **toolchain** of eBPF has complex operations
  - Invoke syscall bpf, perf event, epoll, mmap, etc…
  - Data section and maps need relocation
  - CO-RE or LLVM for different kernel versions
  - Complex operations on maps for control and communications
- eBPF needs to be attached to **events**
- Real world eBPF applications has a mix of kernel **kprobe** and **uprobe**

Challenges

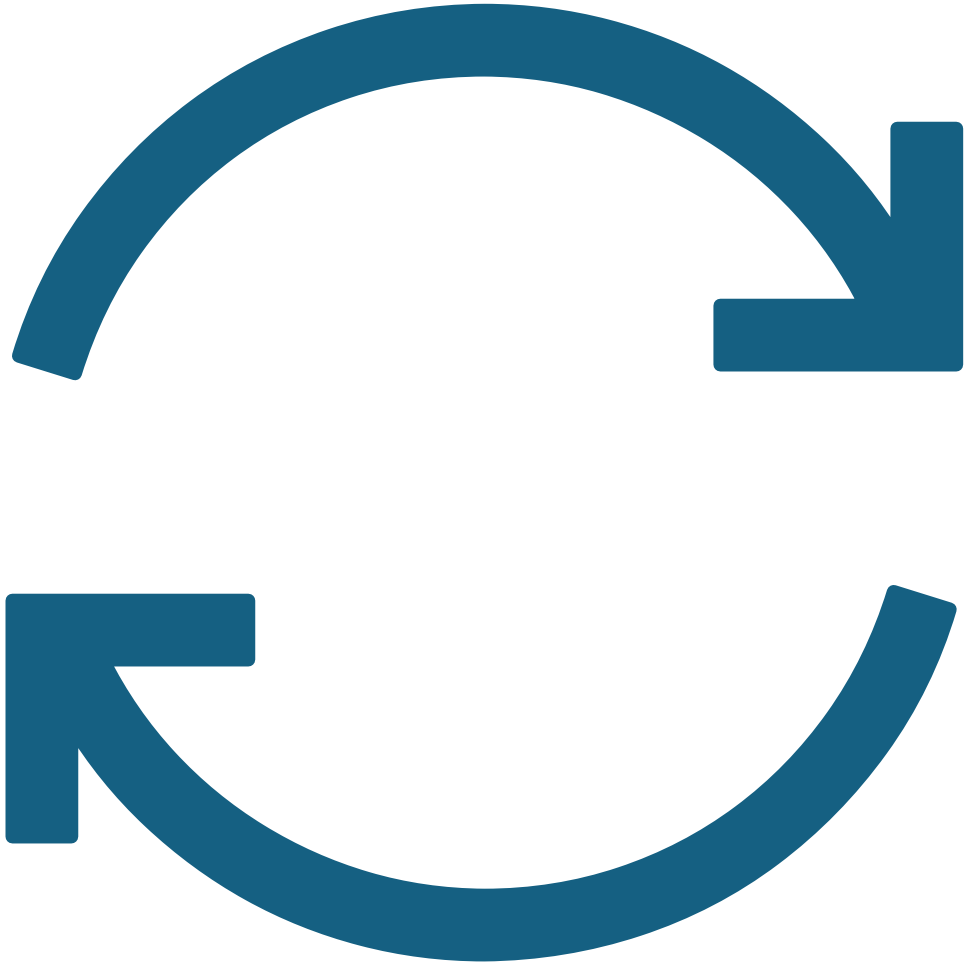
- Userspace **libraries** and **toolchain** of eBPF has complex operations
- eBPF needs to be attached to **events and helpers**
  - A subset of Kernel helpers can be enabled in userspace
  - What kind of events can be captured in userspace: Uprobe and syscall
  - How to find a similar but faster approach to attach to userspace
    - Uprobe can be attach when a process starts, or dynamically inject at run time
    - How to capture all syscall in userspace
- Real world eBPF applications has a mix of kernel **kprobe** and **uprobe**

Challenges

- Userspace **libraries** and **toolchain** of eBPF has complex operations
- eBPF needs to be attached to **events and helpers**
- Real world eBPF applications has **a mix of kernel kprobe and uprobe**
  - How to make userspace eBPF progs using kernel maps

Security

- Verifier-Ensured Safety
- Runtime Memory Protection
- Enable unprivileged kernel map access by pin map
- Split the share memory to multiple sections:
  - The agent eBPF runtime can only read the bpf programs and metadata section
  - Cannot modify or delete any section.
  - can read or write the map data section



Uprobe and Kprobe mix design: prog

Observation 1:

- Only the uprobe attach and reletaed bpf program needs to be changed
- bpf_probe_write_user is enabled by default, and can change behavior of syscall by modify userspace attr before it's copied into kernel.

➔We can make eBPF prog load and attach in userspace without even changing the kernel

- Trace the bpf syscall, record & replay (No always working)
- When the uprobe is attached, find the related prog and maps from kernel (Works)
- Can use kernel verifier or userspace verifier

Uprobe and Kprobe mix design: map

Observation 2:

- Some maps is only used for collecting function args, use by only uprobe or kprobe
- Only few maps need to be used by both kernel eBPF or uprobe eBPF: most of them are related to thread, goroutines, process info, not update frequently

Solutions: **No system call in helpers, using kernel maps with share memory and async**

- **ARRAY_MAPS**: BPF_F_MMAP (mmap support)
- **HASH_MAPS**: Let kernel eBPF access userspace maps, or use Cache & Syscall? Open problems.
- **Ring buffer/perf event**: use bpf user ring buffer to submit back to kernel

Uprobe and Kprobe mix design: data

Observation 3:

- Some Uprobe programs need to access deep kernel data structs (Rare cases)
  - For example, in deepflow project, SSL/TLS hook will get tcp seq to link L4 to L7 traffic for integrated analysis
  - Need access to socket data structs, task structs
  - This cannot be easily achieved in userspace
- However, access to kernel data structs needs a serials of helper call and checks, it's time consuming
  - The Uprobe overhead itself is only 20%-30% or less in deepflow, put it in userspace may not have too much benefits

Potential Solutions: **Configurable Uprobe in userspace**

- **Only** necessary Uprobe eBPF programs in userspace, some Uprobe can also run in kernel
- **Automatically** put some in userspace, some in kernel based on profile (Similar to OSDI'23 UB)