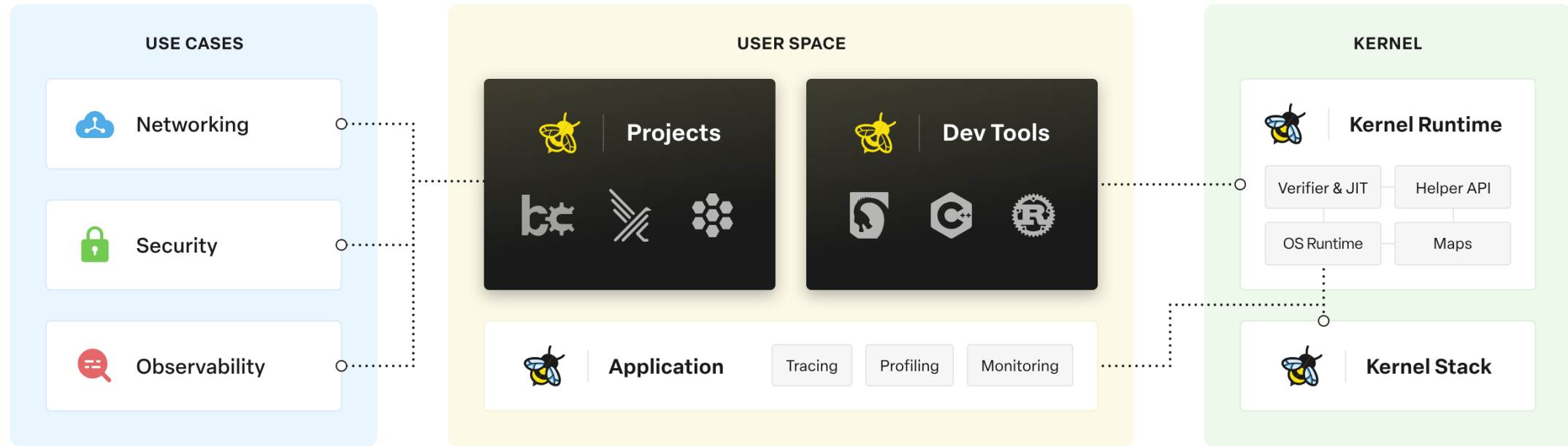

eBPF & eunomia-bpf

郑昱笙

<https://github.com/yunwei37>

eBPF: Runtime from Linux Kernel



✓ Programs are verified to safely execute

✓ Hook anywhere in the kernel to modify functionality

✓ JIT compiler for near native execution speed

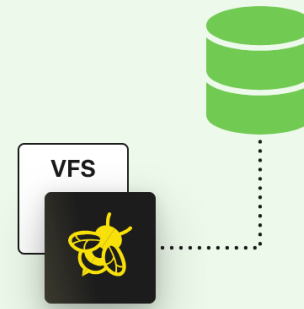
✓ Add OS capabilities at runtime

What's possible with eBPF?



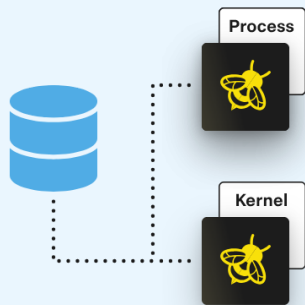
Networking

Speed packet processing without leaving kernel space. Add additional protocol parsers and easily program any forwarding logic to meet changing requirements.



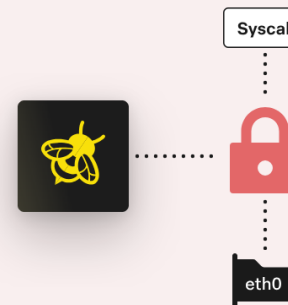
Observability

Collection and in-kernel aggregation of custom metrics with generation of visibility events and data structures from a wide range of possible sources without having to export samples.



Tracing & Profiling

Attach eBPF programs to trace points as well as kernel and user application probe points giving powerful introspection abilities and unique insights to troubleshoot system performance problems.



Security

Combine seeing and understanding all system calls with a packet and socket-level view of all networking to create security systems operating on more context with a better level of control.

Why eBPF?

The original ways to change kernel:

Native Support

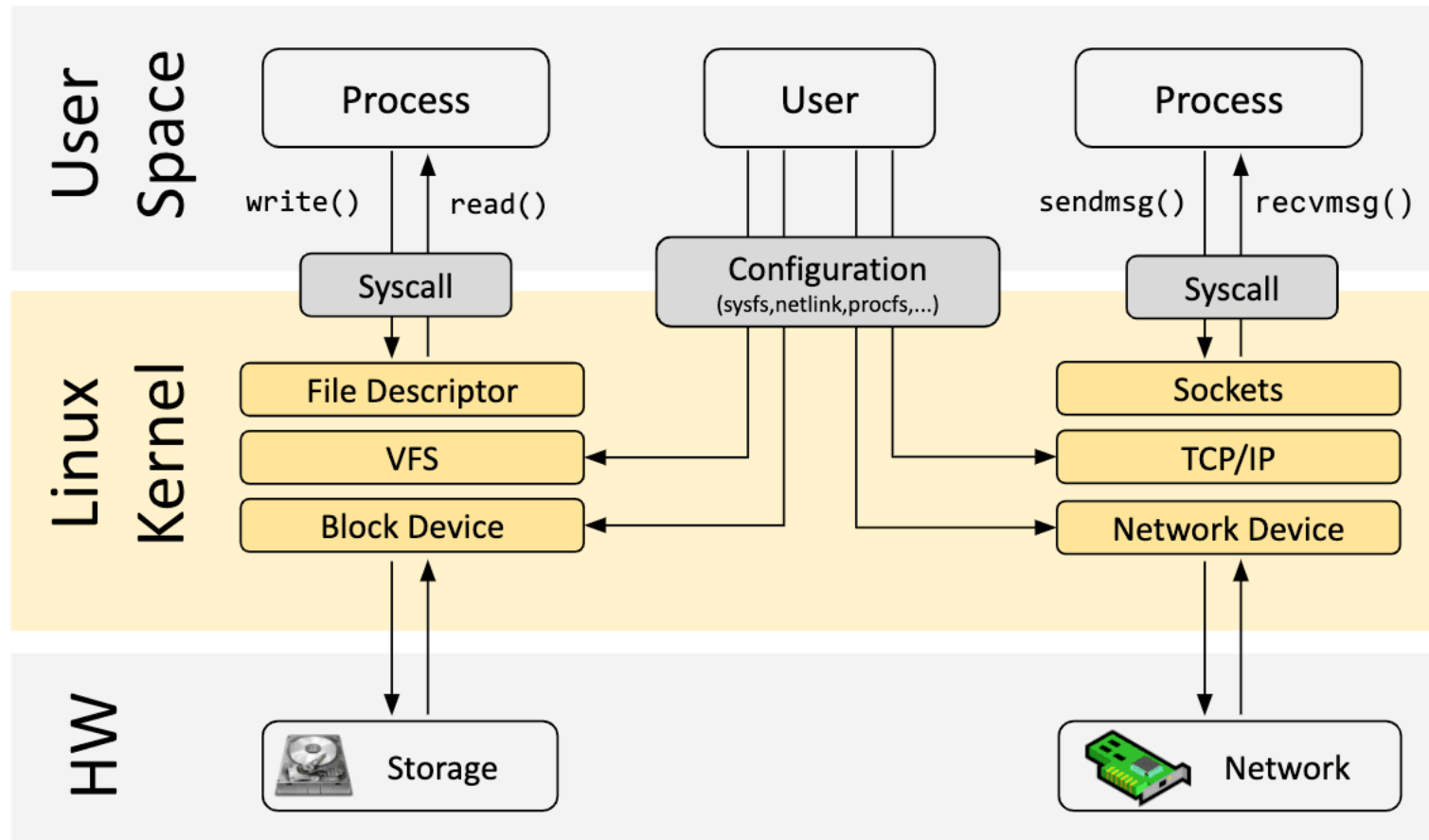
1. Change kernel source code and convince the Linux kernel community that the change is required.
2. Wait several years for the new kernel version to become a commodity.

Kernel Module

1. Write a kernel module
2. Fix it up regularly, as every kernel release may break it
3. Risk corrupting your Linux kernel due to lack of security boundaries

eBPF: more stable API, across kernel version, and verify for security

eBPF for Linux kernel



Current & future of eBPF

The design goal of eBPF is to speed up the Innovation in modern systems.

Current:

- eBPF is already Turing complete, meaning it can perform any computation given enough resources.
 - You can build complex data structs and even put part of redis/memcached in it.
- More usecases: scheduler, driver, etc...
- eBPF for Windows extends eBPF's capabilities beyond Linux

Future:

- Expanding eBPF to More Platforms & Standardize
 - Optimizations & more powerful runtime features
-

eunomia-bpf

eBPF is still a relatively new ecosystem.
What we are doing do:

- Make eBPF more easier to learn and use
- Extend eBPF to userspace

<https://github.com/eunomia-bpf> and <https://eunomia.dev/>

eunomia-bpf

Pinned

[Customize pins](#)**bpftime**Public

Userspace eBPF runtime for Observability, Network & General Extensions Framework



C++



814



75

**bpf-developer-tutorial**Public

eBPF Developer Tutorial: Learning eBPF Step by Step with Examples



C



2.6k



362

**wasm-bpf**Public

WebAssembly library, toolchain and runtime for eBPF programs



Rust



382



26

**eunomia-bpf**Public

A Toolchain to make Build and Run eBPF programs easier



Rust



671



60

**GPTtrace**Public

Generate eBPF programs and tracing with ChatGPT



Python



222



21

**llvmbpf**Public

Userspace eBPF VM with llvm JIT/AOT compiler



LLVM



52



4

bpftime

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

bpftime: Userspace eBPF runtime

A userspace eBPF runtime compatible with kernel

- Support Uprobe/USDT/syscall tracepoints/XDP in userspace
- 10+ Maps types and 30+ helpers support
- Run together with kernel eBPF

The VM inside:

- ubpf or llvmbpf

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

bpftime: Userspace eBPF runtime

A userspace eBPF runtime compatible with kernel

- Support various events
- 10+ Maps types and 30+ helpers support
- Run together with kernel eBPF

The VM inside:

- ubpf or llvmbpf

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

Prog types can attached in userspace:

- tracepoint:raw_syscalls:sys_enter
- tracepoint:syscalls:sys_enter/exit_*
- uretprobe:*
- uprobe:*
- usdt:*
- xdp

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**

bpftime for observability

Why userspace tracing? Faster and More Flexible

- **~1000ns** in kernel uprobe **vs** **~100ns** in userspace uprobe
- **~4ns** for userspace memory access **vs** **~40ns** in kernel
- Tracing Overhead on Untraced Processes for syscall tracepoint

What can we run in userspace?

- Tools like bcc and bpftrace
- Complex Observability Agents with kprobe and uprobe together

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

bpftime for userspace network

Why userspace eBPF?

- kernel bypass is faster: DPDK and AF_XDP
- eBPF ecosystem and tools

Why bpftime? We already have eBPF-in-DPDK

- Not eBPF maps and helpers support
- Not control plane application (Cannot keep compatible)

Faster and more optimize with LLVM

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



Get started

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

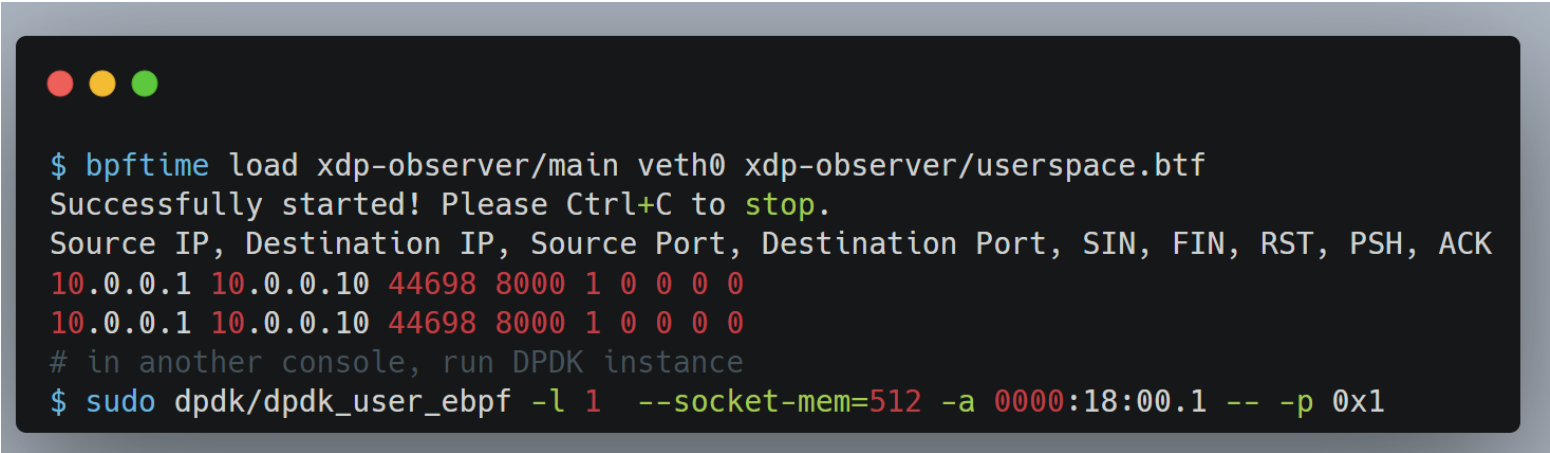
```
$ sudo bpftime attach 101771 # You may need to run make install in root
Inject: "/root/.bpftime/libbpftime-agent.so"
Successfully injected. ID: 1
```

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
```

bpftime with userspace network

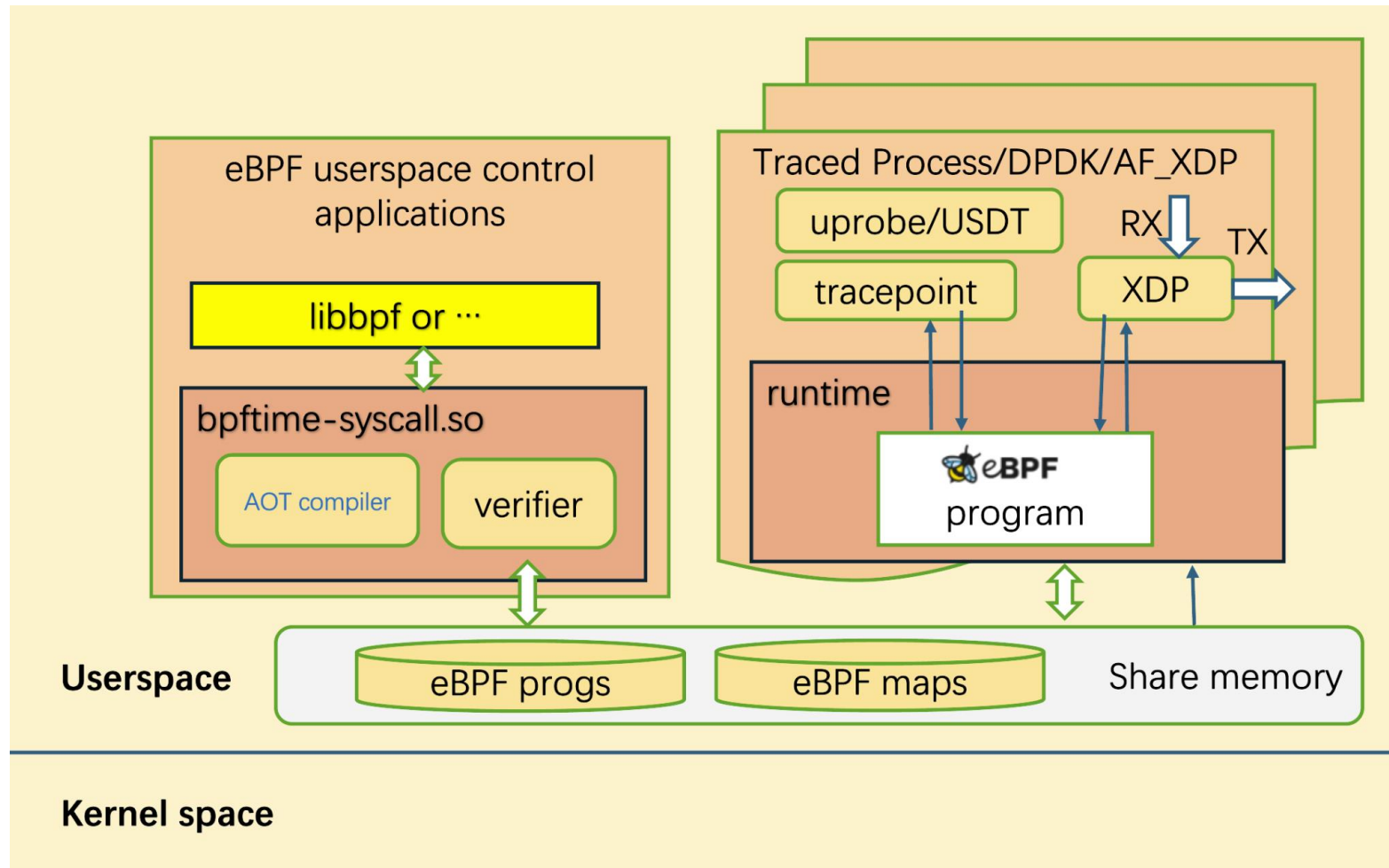
We can now seamlessly integrate the eBPF **XDP** ecosystem into kernel-bypass applications, enabling solutions like **Katran**!

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The terminal displays the output of the 'bpftime load' command, showing a successful start and two lines of network traffic data. The data is formatted as a table with columns for Source IP, Destination IP, Source Port, Destination Port, and flags (SIN, FIN, RST, PSH, ACK).

```
$ bpftime load xdp-observer/main veth0 xdp-observer/userspace.btf
Successfully started! Please Ctrl+C to stop.
Source IP, Destination IP, Source Port, Destination Port, SIN, FIN, RST, PSH, ACK
10.0.0.1 10.0.0.10 44698 8000 1 0 0 0 0
10.0.0.1 10.0.0.10 44698 8000 1 0 0 0 0
# in another console, run DPDK instance
$ sudo dpdk/dpdk_user_ebpf -l 1 --socket-mem=512 -a 0000:18:00.1 -- -p 0x1
```

(limited to XDP_TX and XDP_DROP)

Control plane support



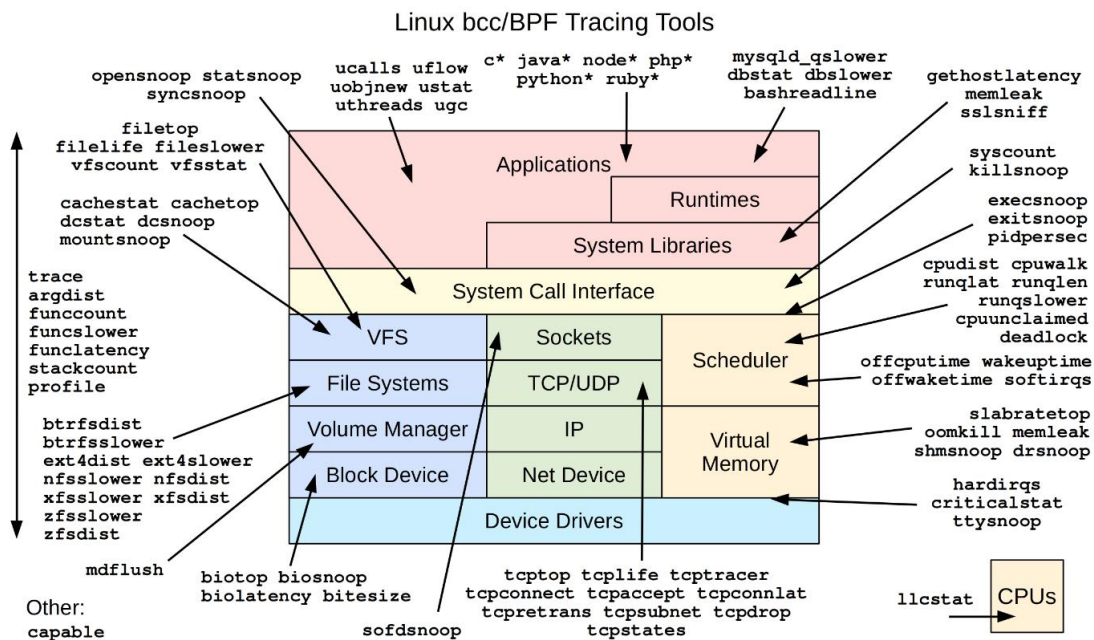
Bpftrace and BCC

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_syscalls: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>

llvmbpf: eBPF VM with llvm JIT/AOT

A standalone eBPF VM and compiler tool

- Separated from bpftime repo
- Easy-to-use
- Tested with [bpf_conformance](#)
- Performance Optimization
 - Inline helpers

<https://github.com/eunomia-bpf/llvmbpf>

A terminal window with a dark background and light blue text. It shows the initialization and execution of an llvmbpf VM. The code includes loading code, registering an external function for printing, and executing the BPF program.

```
llvmbpf_vm vm;  
  
vm.load_code(code, code_len);  
vm.register_external_function(2, "print", (void *)print_func);  
vm.exec(&bpf_mem, sizeof(bpf_mem), res);
```

Other projects

<https://github.com/eunomia-bpf/>

bpf-developer-tutorial

Learning eBPF Step by Step with Examples, cover all topics

- Observability
- Security
- Network
- Scheduler

入门示例

这一部分包含简单的 eBPF 程序示例和介绍。主要利用 `eunomia-bpf` 框架简化开发，介绍 eBPF 的基本用法和开发流程。

- [lesson 0-introduce](#) eBPF 示例教程 0: 核心概念与工具简介
- [lesson 1-helloworld](#) eBPF 入门开发实践教程一: Hello World, 基本框架和开发流程
- [lesson 2-kprobe-unlink](#) eBPF 入门开发实践教程二: 在 eBPF 中使用 kprobe 监测捕获 unlink 系统调用
- [lesson 3-fentry-unlink](#) eBPF 入门开发实践教程三: 在 eBPF 中使用 fentry 监测捕获 unlink 系统调用
- [lesson 4-opensnoop](#) eBPF 入门开发实践教程四: 在 eBPF 中捕获进程打开文件的系统调用集合, 使用全局变量过滤进程 pid
- [lesson 5-uprobe-bashreadline](#) eBPF 入门开发实践教程五: 在 eBPF 中使用 uprobe 捕获 bash 的 readline 函数调用
- [lesson 6-sigsnoop](#) eBPF 入门开发实践教程六: 捕获进程发送信号的系统调用集合, 使用 hash map 保存状态
- [lesson 7-execsnoop](#) eBPF 入门实践教程七: 捕获进程执行事件, 通过 perf event array 向用户态打印输出
- [lesson 8-exitsnoop](#) eBPF 入门开发实践教程八: 在 eBPF 中使用 exitsnoop 监控进程退出事件, 使用 ring buffer 向用户态打印输出
- [lesson 9-runqlat](#) eBPF 入门开发实践教程九: 捕获进程调度延迟, 以直方图方式记录
- [lesson 10-hardirqs](#) eBPF 入门开发实践教程十: 在 eBPF 中使用 hardirqs 或 softirqs 捕获中断事件

高级文档和示例

我们开始构建完整的 eBPF 项目, 主要基于 `libbpf`, 并将其与各种应用场景结合起来, 以便实际使用。

- [lesson 11-bootstrap](#) eBPF 入门开发实践教程十一: 在 eBPF 中使用 libbpf 开发用户态程序并跟踪 exec() 和 exit() 系统调用
- [lesson 12-profile](#) eBPF 入门实践教程十二: 使用 eBPF 程序 profile 进行性能分析
- [lesson 13-tcpconlat](#) eBPF入门开发实践教程十三: 统计 TCP 连接延时, 并使用 libbpf 在用户态处理数据
- [lesson 14-tcpstates](#) eBPF入门实践教程十四: 记录 TCP 连接状态与 TCP RTT
- [lesson 15-javagc](#) eBPF 入门实践教程十五: 使用 USDT 捕获用户态 Java GC 事件耗时
- [lesson 16-memleak](#) eBPF 入门实践教程十六: 编写 eBPF 程序 Memleak 监控内存泄漏
- [lesson 17-biopattern](#) eBPF 入门实践教程十七: 编写 eBPF 程序统计随机/顺序磁盘 I/O
- [lesson 18-further-reading](#) 更多的参考资料: 论文、项目等等
- [lesson 19-lsm-connect](#) eBPF 入门实践教程: 使用 LSM 进行安全检测防御
- [lesson 20-tc](#) eBPF 入门实践教程二十: 使用 eBPF 进行 tc 流量控制
- [lesson 21-xdp](#) eBPF 入门实践教程二十一: 使用 XDP 进行可编程数据包处理

GPTtrace

An experiment for generating eBPF programs with natural language

- start tracing with natural language
- let AI explain the result to you

```
yunwei@yunwei-virtual-machine:~/GPTtrace$ python3 -m gpttrace "tracing with Count page faults by process"
The bpf program to run is: sudo bpftrace -e software:faul...:1 { @[comm] = count(); }
timeout: 300
Enter 'y' to proceed: y
^C

@[sudo]: 5
@[which]: 130
@[sleep]: 146
@[ps]: 330
@[sh]: 377
@[sed]: 451
@[cat]: 1439
@[node]: 2983
@[cpuUsage.sh]: 3702

AI explanation:
The output of the previous bpftrace result is a list of processes along with the number of occurrences of each process in the trace. Each process is represented by its name in square brackets ([]), followed by the count of page faults generated by that process.

For example, the process with the name `sudo` generated 5 page faults, the process `which` generated 130 page faults, and so on.

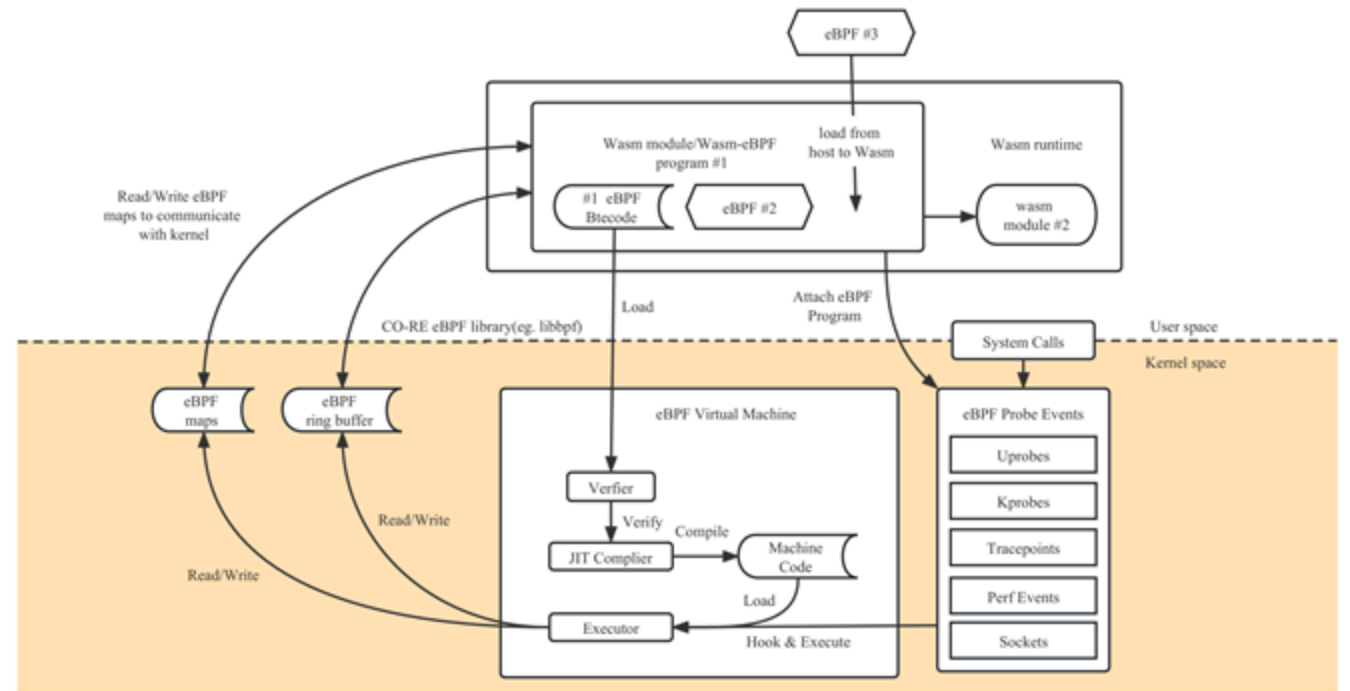
The list is sorted in ascending order based on the count of page faults. Therefore, the process with the lowest count is displayed first, and the process with the highest count is displayed last.

This output provides information about the frequency of page faults generated by different processes in the system, helping to identify potential performance or resource utilization issues.
yunwei@yunwei-virtual-machine:~/GPTtrace$
```

Wasm-bpf

Wasm-bpf: Wasm library and
toolchain for eBPF

Make deploy eBPF easier



<https://github.com/eunomia-bpf/wasm-bpf>

eunomia-bpf tools

A Toolchain to make Build and Run eBPF programs easier

Compile the program using ecc:

```
$ ./ecc minimal.bpf.c  
Compiling bpf object...  
Packing ebpf object and config into package.json...
```

Or compile using a docker image:

```
docker run -it -v `pwd`:/:src/ ghcr.io/eunomia-bpf/ecc-`uname -m`:latest
```

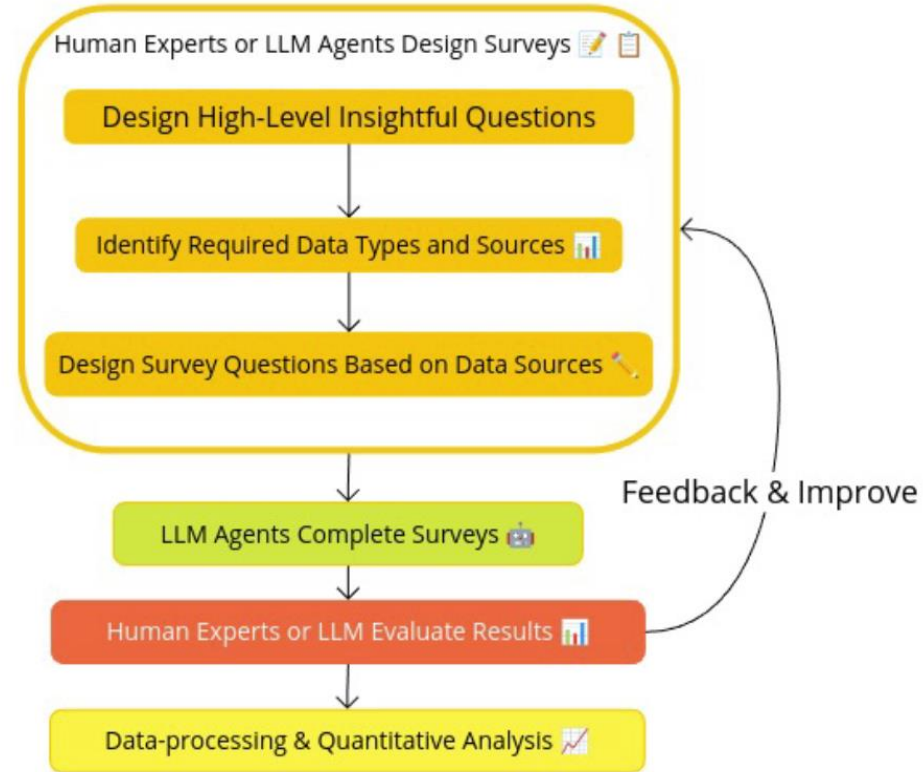
Then run the compiled program using ecli:

```
$ sudo ./ecli run package.json  
Running eBPF program...
```

<https://github.com/eunomia-bpf/eunomia-bpf>

Code-survey

Using LLM to understand Linux kernel or other large codebases



<https://github.com/eunomia-bpf/code-survey>
<https://arxiv.org/abs/2410.01837>

What's next?

<https://github.com/eunomia-bpf>

PLCT intern

BJ72 eunomia-bpf intern (20240101 开放 2 名)

(本岗位为外部社区联合贡献岗位, mentor来自 eunomia-bpf 社区。)

eunomia-bpf 社区是一个致力于探索、改进、扩展 eBPF 相关开发工具链和运行时的开源社区, 我们维护了一系列的开源项目、文档教程, 并且积极探索将 eBPF 从内核态扩展到用户态、简化 eBPF 的开发和移植、发布流程、使用 LLM 自动生成 eBPF 代码、将 Wasm 和 eBPF 结合起来等等相关解决方案。希望我们能有机会一起扩展整个 eBPF 生态的边界, 并且做出能持续长久、有影响力的开源项目。

- 主页: <https://eunomia.dev> 和 <https://github.com/eunomia-bpf>
- 教程: <https://eunomia.dev/zh/tutorials/> 和博客: <https://eunomia.dev/zh/blogs/>

工作内容:

Also GSOC and OSPP

<https://github.com/plctlab/weloveinterns/blob/master/open-internships.md>

Questions?

<https://github.com/eunomia-bpf>

Thanks
