KubeCon | CloudNativeCon

North America 2023

KubeCon | CloudNativeCon

North America 2023

eBPF + Wasm: Lightweight Observability on Steroids

Michael Yuan & Yusheng Zheng

Agenda

Introduction to eBPF and Wasm

How Wasm improves eBPF developer experience:

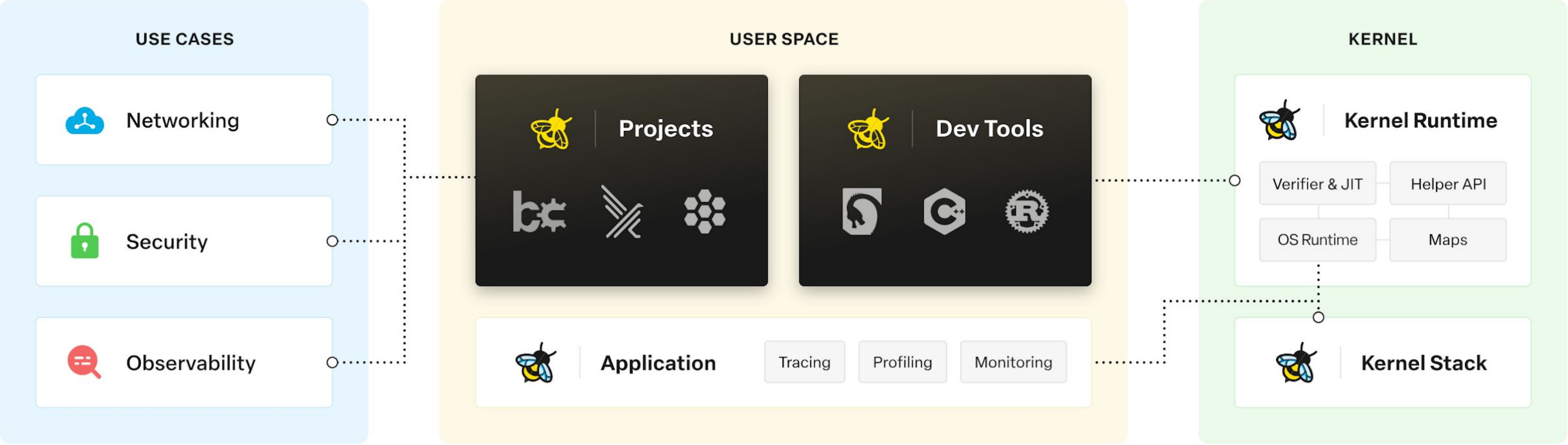
- Non-intrusive deployment into k8s pods
- Decoupled from application workloads
- Declarative security checks at deployment time
- Supports user-space eBPF
- Downstream analytics of eBPF data

How eBPF improves Wasm developer experience:

- Better security sandboxing for WASI
- Better observability for host calls
- Better debugging

eBPF

eBPF (Extended Berkeley Packet Filter): Dynamically and safely program the Linux kernel for efficient networking, observability, tracing, and security



WebAssembly

WebAssembly (*Wasm*):

- A binary format for user space security sandbox
- Support multiple programming languages
- Very lightweight and fast
- Access to host resources through capability-based security declarations
- Cross platform portability

WasmEdgeRuntime

https://github.com/WasmEdge/WasmEdge

KubeCon | CloudNativeCon

North America 2023

How Wasm improves eBPF DevEx

eBPF Deployment models

- Integrated control plane：bundled in the workload app
  cilium, pixie, tetragon, falco…

- Decoupled / sidecar：a eBPF daemon
  bumblebee, inspektor-gadget, bpfd…

Integrated eBPF deployment

Closely integrate model and deploy with containers in a k8s cluster

- Embedding in the agent to release the eBPF probe
- Allows direct control and efficient management

Challenges

- Security and Permissions: CAP_BPF
- Heavy weight for small eBPF programs
- More than one eBPF User: potential for conflicts in hooks

Integrated eBPF deployment

Closely integrate model and deploy with containers in a k8s cluster

Customs Protocols

HTTP/2

SSL/TLS

Challenges

- New eBPF probes needs to be released with the agent
- Hard to trace private Protocols or Application logic with only pre-defined probes
- Hard to perform user-defined complex data processing

Decoupled sidecar eBPF deployment

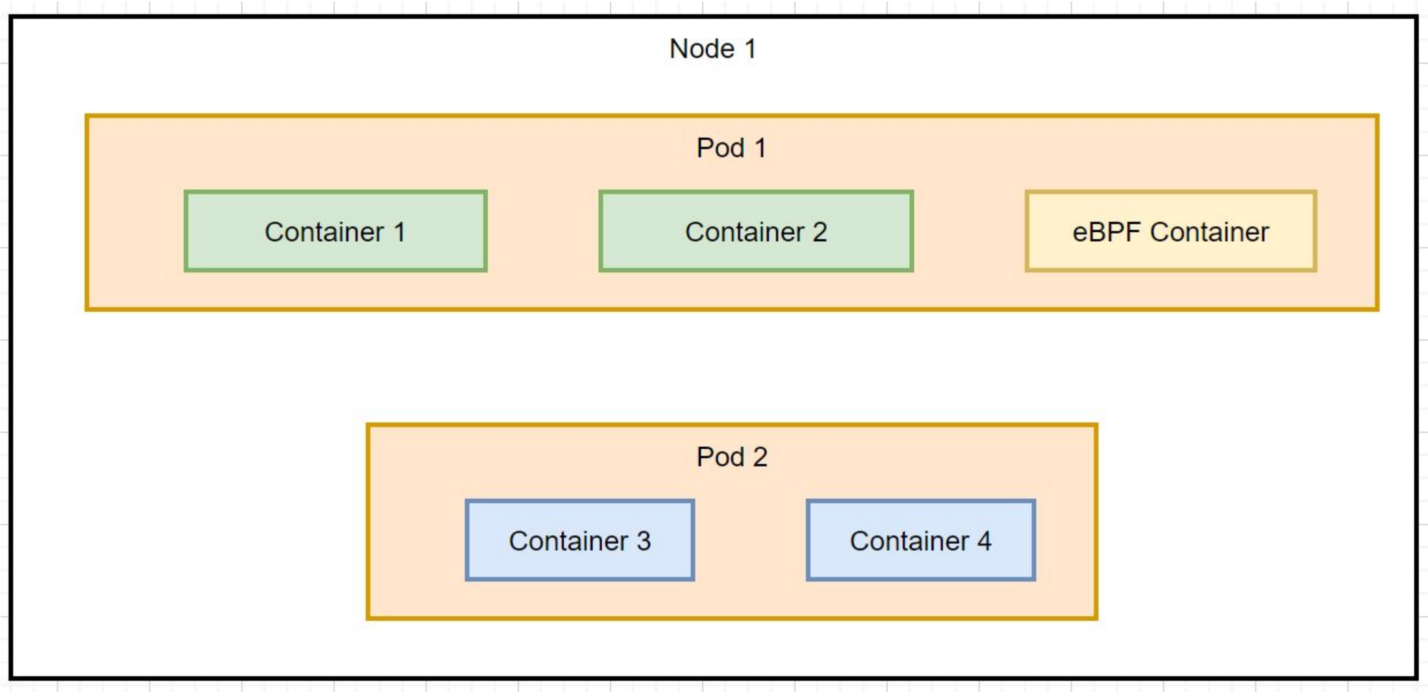
Daemon-Based Approach

- BPF daemon manages lifecycle and permissions via RPCs
- Decouples BPF functionality for modularity

Challenges

- Support and Maintenance
- Critical Component Risk
- Consistency in Updates
- Kernel Feature Integration

Wasm + eBPF: best of both worlds



Node 1

Pod 1

Container 1

Container 2

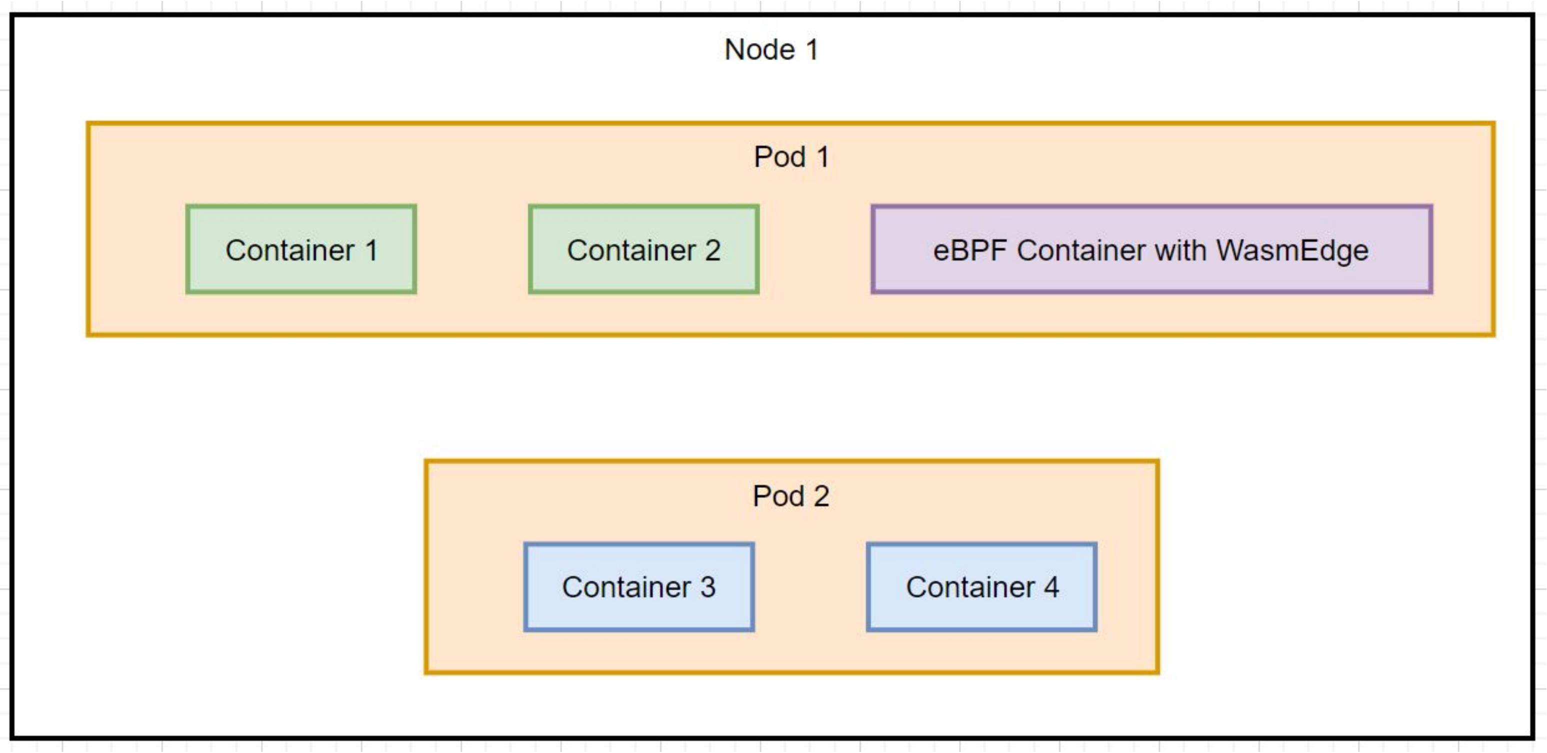
eBPF Container

Pod 2

Container 3

Container 4

Wasm + eBPF: best of both worlds



WasmEdge eBPF plugin: wasm-bpf

- Library and tool chains: https://github.com/eunomia-bpf/wasm-bpf
- Runtime support: https://github.com/WasmEdge/WasmEdge/tree/master/plugins/wasm_bpf

Faster, easier & safer eBPF deployment

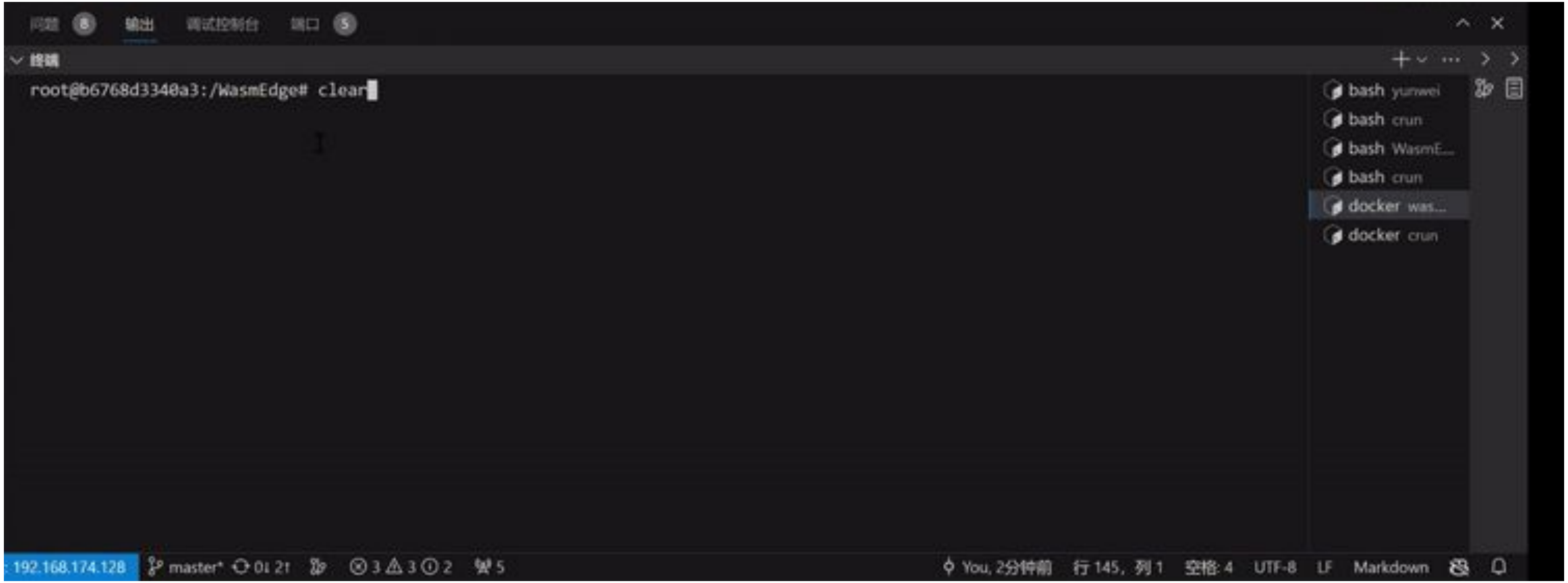
- [Faster] WasmEdge app is only 1/100 the size of containers
- [Easier] Portable across platforms and kernel versions with CO-RE
  - with user space eBPF runtime, no kernel eBPF support needed
- [Easier] Manage eBPF packages as OCI images in Wasm module
- [Safer] Configurable and limited eBPF WASI behavior
- [Safer] Improved security with access control to lower kernel resources
- [Safer] Sandbox the user space and enable eBPF plugin in tools

Trade off

Too good to be true? There is no free lunch.

- Libraries and toolchains need to be ported,
  - e.g. libbpf and libbpf-rs -> libbpf-wasm
- eBPF features are limited in Wasm
  - e.g., sockets

Use container tools to run Wasm + eBPF

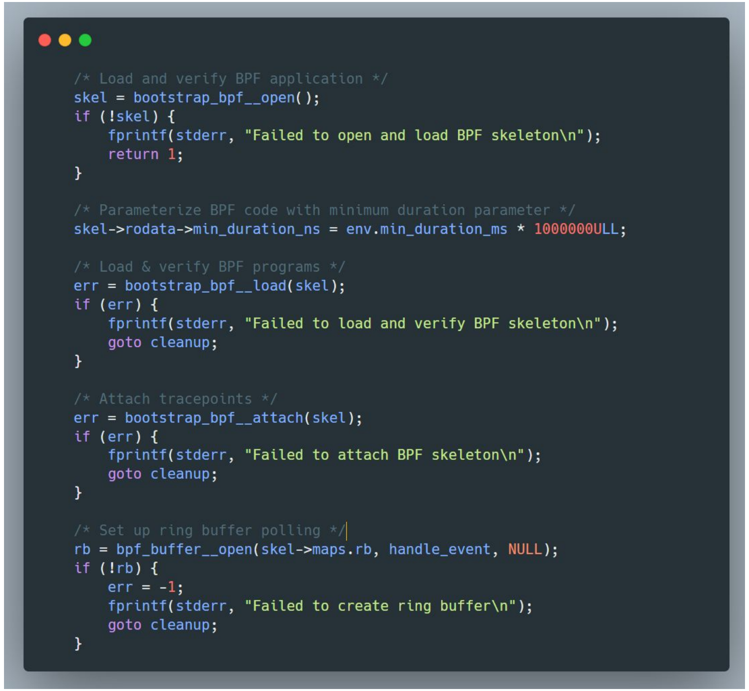


```
root@b6768d3340a3:/WasmEdge# clear
```

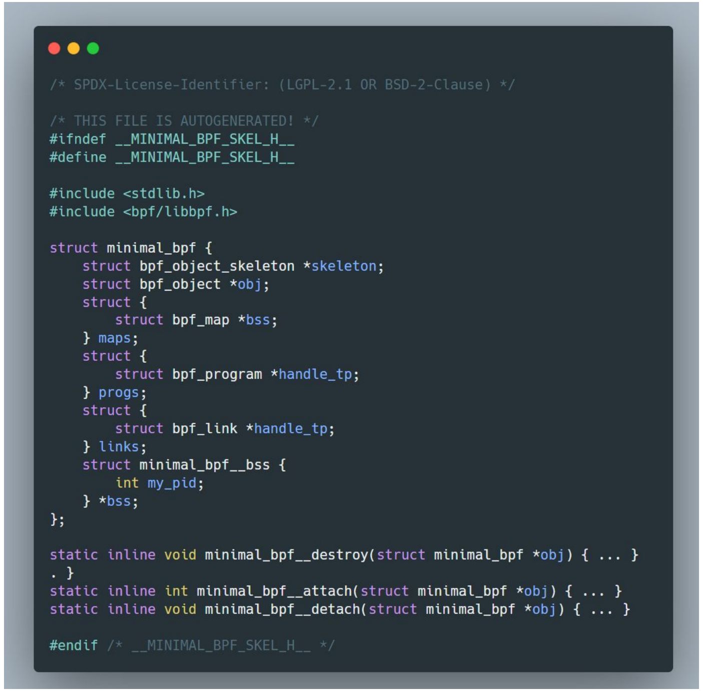
Use podman to deploy eBPF with WebAssembly

Developer experience

Similar developing experience as the libbpf-bootstrap



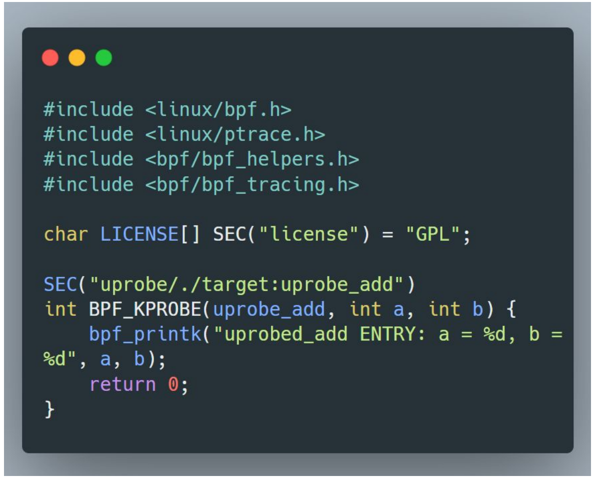
```
/* Load and verify BPF application */
skel = bootstrap_bpf__open();
if (!skel) {
    fprintf(stderr, "Failed to open and load BPF skeleton\n");
    return 1;
}

/* Parameterize BPF code with minimum duration parameter */
skel->rodata->min_duration_ns = env.min_duration_ms * 1000000ULL;

/* Load & verify BPF programs */
err = bootstrap_bpf__load(skel);
if (err) {
    fprintf(stderr, "Failed to load and verify BPF skeleton\n");
    goto cleanup;
}

/* Attach tracepoints */
err = bootstrap_bpf__attach(skel);
if (err) {
    fprintf(stderr, "Failed to attach BPF skeleton\n");
    goto cleanup;
}

/* Set up ring buffer polling */
rb = bpf_buffer__open(skel->maps.rb, handle_event, NULL);
if (!rb) {
    err = -1;
    fprintf(stderr, "Failed to create ring buffer\n");
    goto cleanup;
}
```



```
/* SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause) */

/* THIS FILE IS AUTOGENERATED! */
#ifndef __MINIMAL_BPF_SKEL_H__
#define __MINIMAL_BPF_SKEL_H__

#include <stdlib.h>
#include <bpf/libbpf.h>

struct minimal_bpf {
    struct bpf_object_skeleton *skeleton;
    struct bpf_object *obj;
    struct {
        struct bpf_map *bss;
    } maps;
    struct {
        struct bpf_program *handle_tp;
    } progs;
    struct {
        struct bpf_link *handle_tp;
    } links;
    struct minimal_bpf__bss {
        int my_pid;
    } *bss;
};

static inline void minimal_bpf__destroy(struct minimal_bpf *obj) { ... }
. }
static inline int minimal_bpf__attach(struct minimal_bpf *obj) { ... }
static inline void minimal_bpf__detach(struct minimal_bpf *obj) { ... }

#endif /* __MINIMAL_BPF_SKEL_H__ */
```

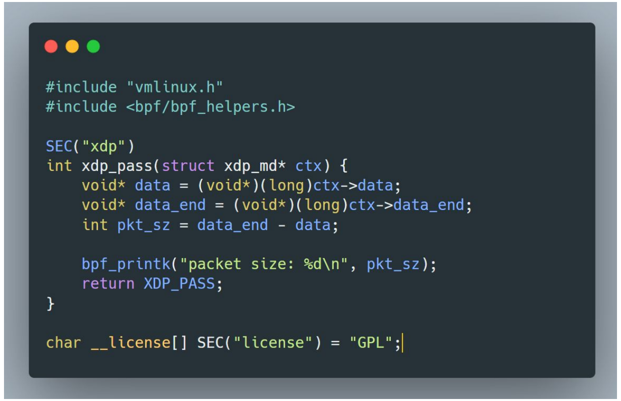
Examples

Support eBPF use cases from Observability, Networking to Security



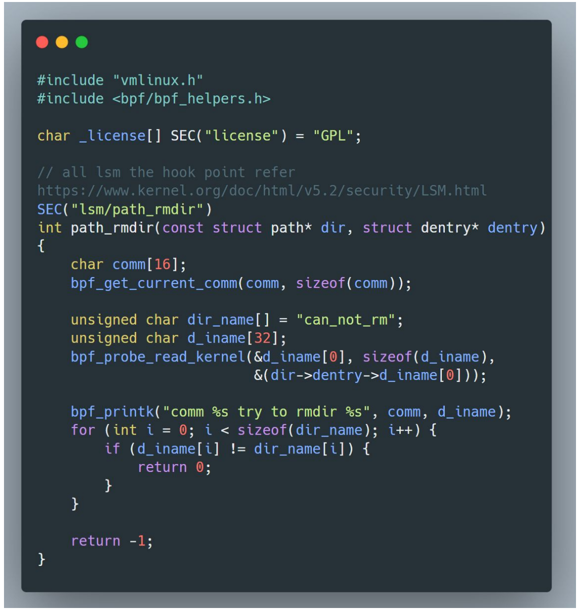
```c
#include <linux/bpf.h>
#include <linux/ptrace.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>

char LICENSE[] SEC("license") = "GPL";

SEC("uprobe/./target:uprobe_add")
int BPF_KPROBE(uprobe_add, int a, int b) {
    bpf_printk("uprobed_add ENTRY: a = %d, b =
%d", a, b);
    return 0;
}
```

Uprobe



```c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>

SEC("xdp")
int xdp_pass(struct xdp_md* ctx) {
    void* data = (void*)(long)ctx->data;
    void* data_end = (void*)(long)ctx->data_end;
    int pkt_sz = data_end - data;

    bpf_printk("packet size: %d\n", pkt_sz);
    return XDP_PASS;
}

char __license[] SEC("license") = "GPL";
```

XDP

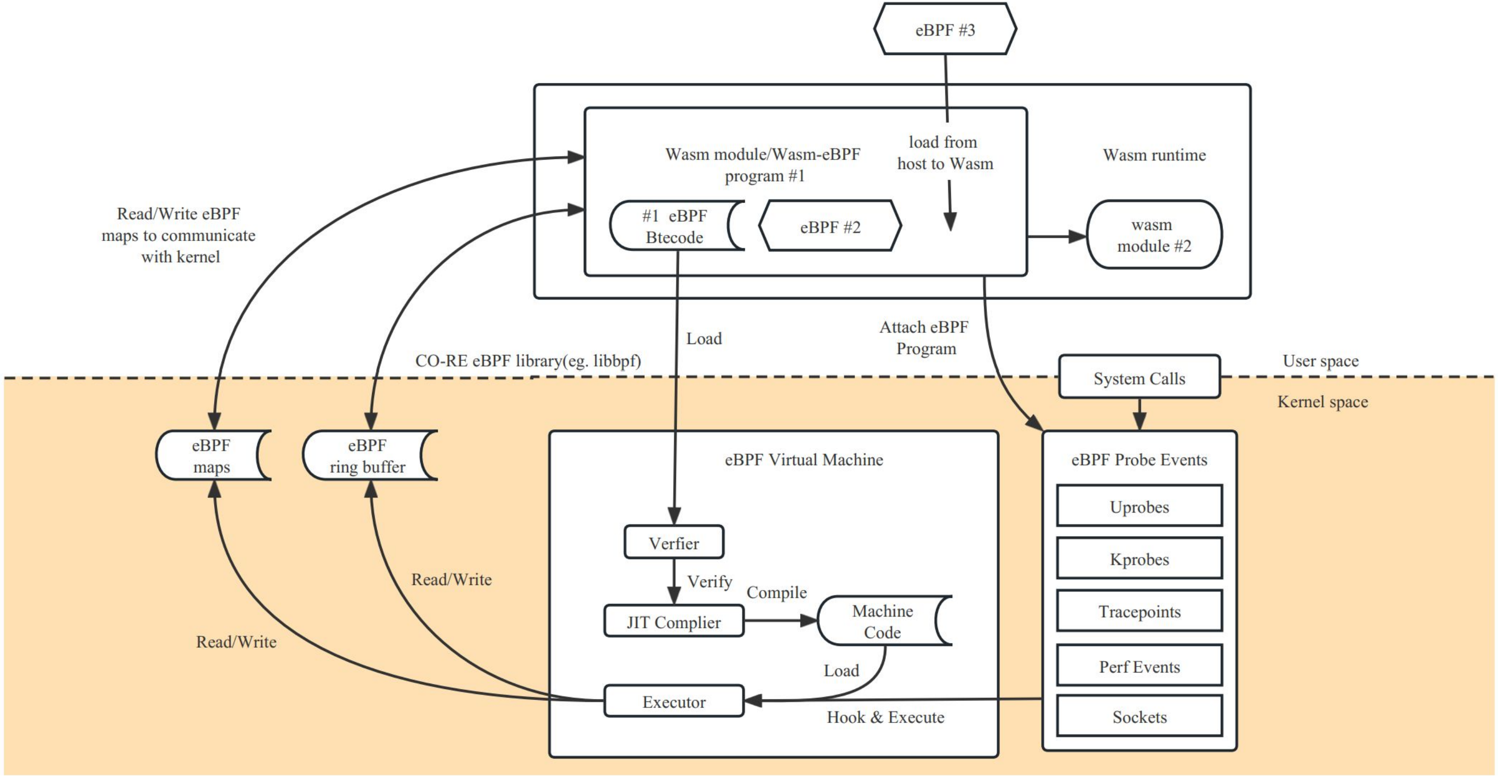


```c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>

char _license[] SEC("license") = "GPL";

// all lsm the hook point refer
https://www.kernel.org/doc/html/v5.2/security/LSM.html
SEC("lsm/path_rmdir")
int path_rmdir(const struct path* dir, struct dentry* dentry)
{
    char comm[16];
    bpf_get_current_comm(comm, sizeof(comm));

    unsigned char dir_name[] = "can_not_rm";
    unsigned char d_iname[32];
    bpf_probe_read_kernel(&d_iname[0], sizeof(d_iname),
                          &(dir->dentry->d_iname[0]));

    bpf_printk("comm %s try to rmdir %s", comm, d_iname);
    for (int i = 0; i < sizeof(dir_name); i++) {
        if (d_iname[i] != dir_name[i]) {
            return 0;
        }
    }

    return -1;
}
```

LSM

https://github.com/eunomia-bpf/wasm-bpf/blob/main/examples

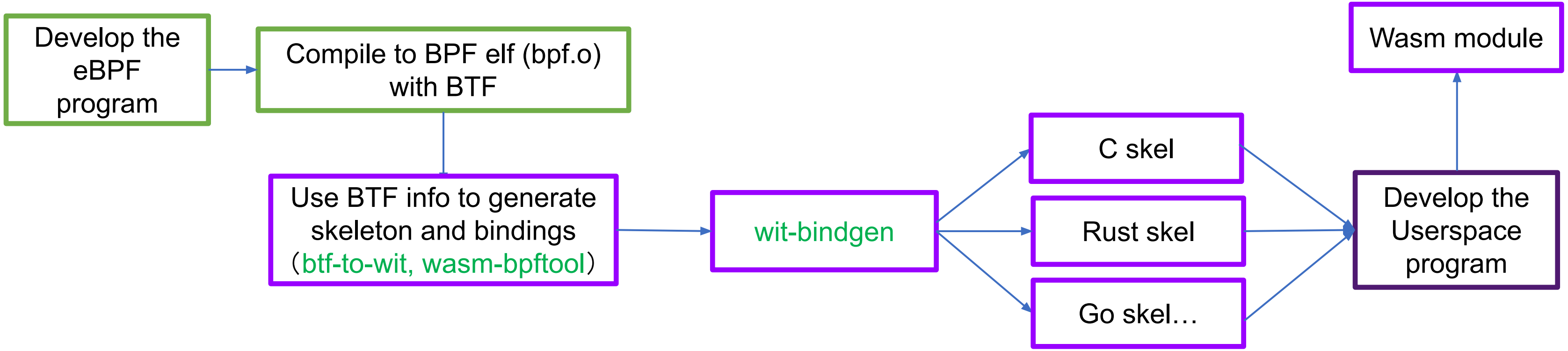
How it works


eBPF #3
Wasm module/Wasm-eBPF program #1
load from host to Wasm
Wasm runtime
#1 eBPF Btecode
eBPF #2
wasm module #2
Read/Write eBPF maps to communicate with kernel
Load
Attach eBPF Program
CO-RE eBPF library(eg. libbpf)
System Calls
User space
Kernel space
eBPF maps
eBPF ring buffer
eBPF Virtual Machine
eBPF Probe Events
Uprobes
Kprobes
Tracepoints
Perf Events
Sockets
Verfier
Verify
Compile
JIT Complier
Machine Code
Load
Executor
Hook & Execute
Read/Write
Read/Write

How it works

The toolchains:

- Auto generate the Wasm-eBPF skeleton headers and type definitions for bindings
- No serialization overhead for complex data type
- Share memory maps between Wasm and kernel


Develop the eBPF program
Compile to BPF elf (bpf.o) with BTF
Use BTF info to generate skeleton and bindings（btf-to-wit, wasm-bpftool）
wit-bindgen
C skel
Rust skel
Go skel…
Develop the Userspace program
Wasm module

Challenges

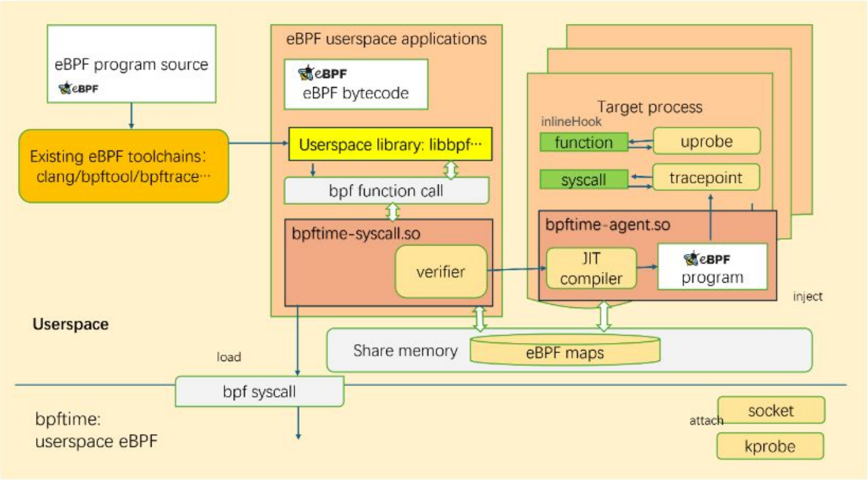
- Port libraries and prepare toolchains to Wasm:
  C/C++: libbpf
  Rust: libbpf-rs
  Go: cilium/go
- Wasm is 32bit and eBPF is 64 bit: data layout is different
  Use toolchains to generated bindings and avoid serialization
- eBPF may require kernel version support:
  Use CO-RE to enable portable between different kernel
  Use user space eBPF runtime for optional
  A compatible layer

Wasm with user space eBPF

Wasm with kernel eBPF enable more possibilities, but require kernel version support and privilege.

- Completely Userspace eBPF runtime

- Uprobe and Syscall hooks, Interprocess maps
- No manual integration or restart required
- Compatible with kernel eBPF toolchains
- Can run bcc tools and bpftrace in userspace
- Speed up 10x than kernel uprobe



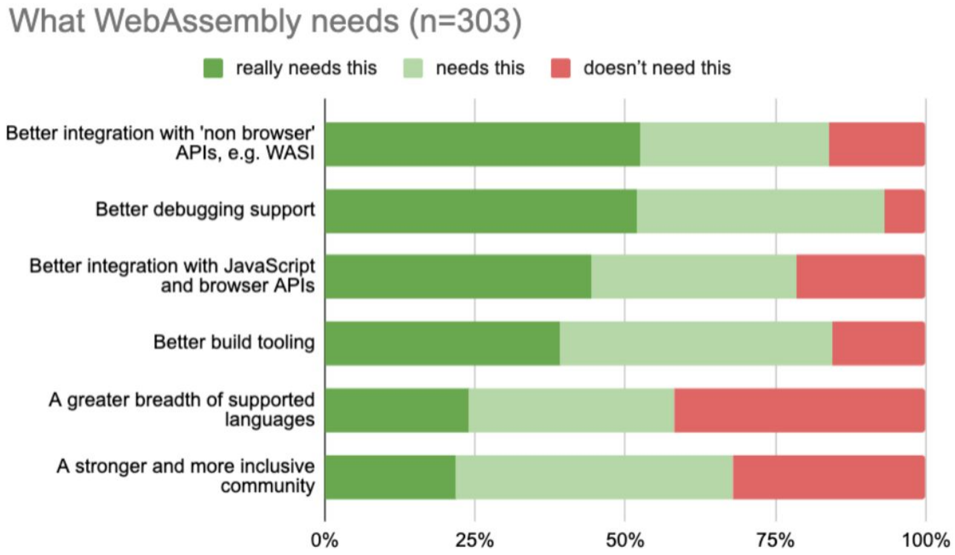
https://github.com/eunomia-bpf/bpftime

How eBPF improves Wasm DevEx

WASI and Debugging needs

WebAssembly needs:

- A better WASI (WebAssembly System Interface)
- A better debugging toolchain



eBPF makes WASI more secure

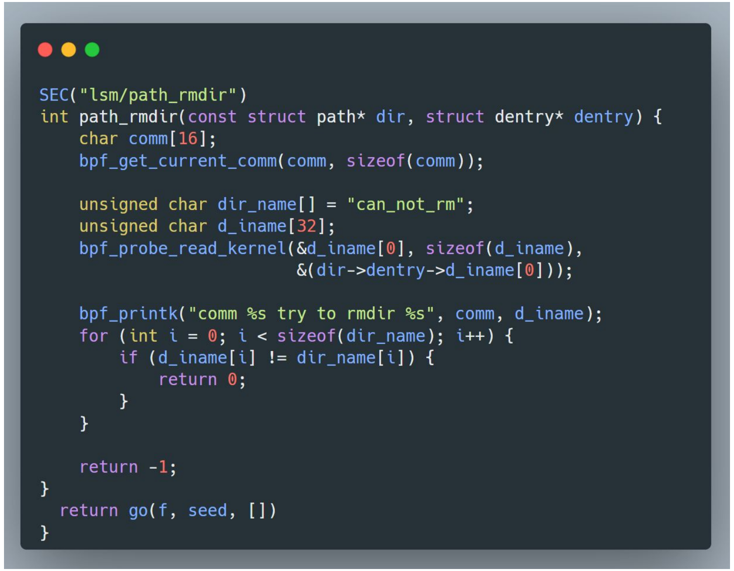
WASI checks if an accessed path is on an "allowed" list before granting access:

- Non-trivial task to implement this works
- Error prone: reliance on code-review
- Less configurable: directory-level granularity, not file-level.

Solution:

- Use eBPF LSM and seccomp to provide programmable access-control in Linux kernel for Wasi

Example: hook in dir remove and check the permission to remove a spec directory



```
SEC("lsm/path_rmdir")
int path_rmdir(const struct path* dir, struct dentry* dentry) {
    char comm[16];
    bpf_get_current_comm(comm, sizeof(comm));

    unsigned char dir_name[] = "can_not_rm";
    unsigned char d_iname[32];
    bpf_probe_read_kernel(&d_iname[0], sizeof(d_iname),
                          &(dir->dentry->d_iname[0]));

    bpf_printk("comm %s try to rmdir %s", comm, d_iname);
    for (int i = 0; i < sizeof(dir_name); i++) {
        if (d_iname[i] != dir_name[i]) {
            return 0;
        }
    }

    return -1;
}
  return go(f, seed, [])
}
```

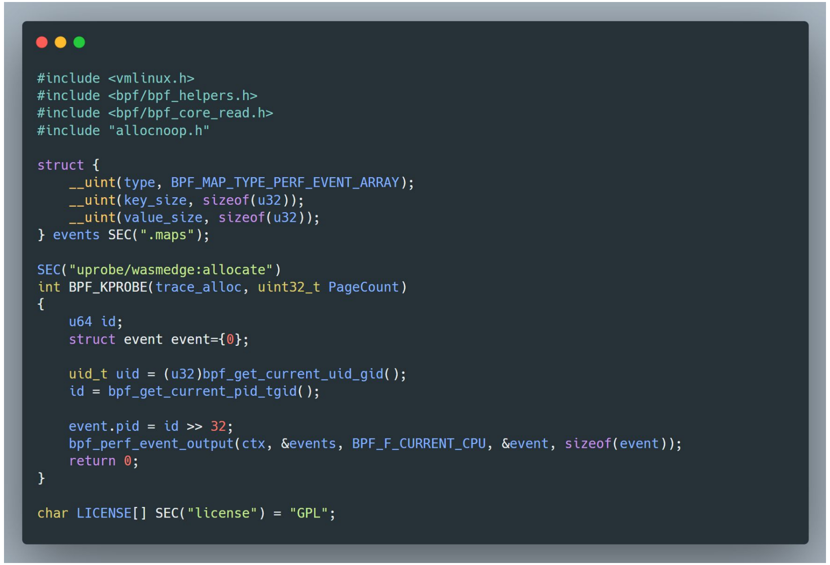
https://github.com/eunomia-bpf/wasm-bpf/tree/main/examples/lsm

Debug tracing for Wasm runtime

Enabling Detailed Tracing of WebAssembly with eBPF

- Wasm provides limited tracing methods
- eBPF's Uprobe can trace any user-space function
- No additional instrumentation needed.
- userspace eBPF runtime like bpftime, enable fast Uprobe without kernel support requirement and without root privilege.

Examples: trace memory allocation in WasmEdge



```c
#include <vmlinux.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_core_read.h>
#include "allocnoop.h"

struct {
    __uint(type, BPF_MAP_TYPE_PERF_EVENT_ARRAY);
    __uint(key_size, sizeof(u32));
    __uint(value_size, sizeof(u32));
} events SEC(".maps");

SEC("uprobe/wasmedge:allocate")
int BPF_KPROBE(trace_alloc, uint32_t PageCount)
{
    u64 id;
    struct event event={0};

    uid_t uid = (u32)bpf_get_current_uid_gid();
    id = bpf_get_current_pid_tgid();

    event.pid = id >> 32;
    bpf_perf_event_output(ctx, &events, BPF_F_CURRENT_CPU, &event, sizeof(event));
    return 0;
}

char LICENSE[] SEC("license") = "GPL";
```

Take away

1. A “Wasm container” can deploy eBPF on hosts of k8s pods — it has the benefits of both the tight integration and RPC approaches.

2. Wasm can run user space eBPF apps for tasks like data analytics.

3. eBPF could become part of a WASI implementation on Linux to enhance security and support debugging.

KubeCon | CloudNativeCon

North America 2023

Thanks

Session QR Codes will be
sent via email before the event

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

Agenda

Introduction to eBPF and Wasm

How can WebAssembly enhance eBPF:

- Current eBPF deployment model and challenges
- What Wasm can bring

How can eBPF enhance WebAssembly:

- Extend the WASI
- Enable better observability for Wasm Runtime

Compare eBPF and Wasm

WebAssembly (Wasm):

- Sandboxed
- Use run-time checks for any arbitrary code
- performance overhead
- Better language support and ecosystem

Extended Berkeley Packet Filter (eBPF):

- Static Verification
- minimize run-time checks, known before execution
- Performance-Oriented
- Small C programs

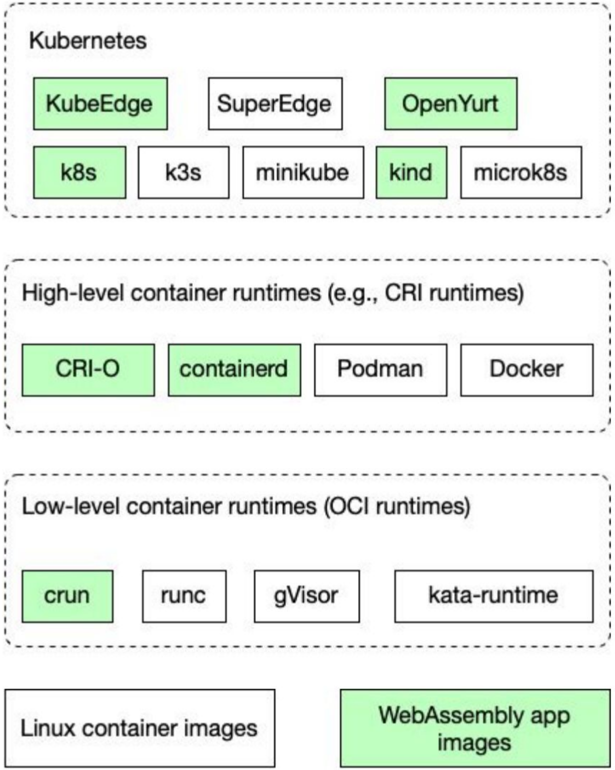
WasmEdge

WasmEdge: A lightweight, secure, high-performance and extensible WebAssembly Runtime

- Integrated support for WasmEdge in Docker CLI
- Transparently supports containers and Wasm apps from the same tools and in the same cluster

https://github.com/WasmEdge/WasmEdge

WasmEdgeRuntime



Kubernetes

KubeEdge SuperEdge OpenYurt

k8s k3s minikube kind microk8s

High-level container runtimes (e.g., CRI runtimes)

CRI-O containerd Podman Docker

Low-level container runtimes (OCI runtimes)

crun runc gVisor kata-runtime

Linux container images

WebAssembly app images

The container ecosystem

eBPF Deployment Challenges

In summary:

- Security Risks with Privileged Access
- Compatibility Issues with eBPF Hooks and Kernel Versions
- Complex Lifecycle and Deployment Management
- Challenges with Versioning and Plugability

What Wasm can bring

WebAssembly’s possible Role in Enhancing eBPF Deployments:

- Lightweight and portable
- Fine-Grained Permissions
- Help with Isolation
- Lifecycle Management
- Versioning and Update with plugin

WasmEdge eBPF plugin: wasm-bpf

WebAssembly library, toolchain and runtime for eBPF programs

- Create Wasm-based eBPF control plane applications
- Enable managing eBPF programs in k8s pods with lightweight Wasm container
- Enables Wasm plugin in eBPF core applications

https://github.com/WasmEdge/WasmEdge/tree/master/plugins/wasm_bpf
https://github.com/eunomia-bpf/wasm-bpf

Make eBPF deployment safer

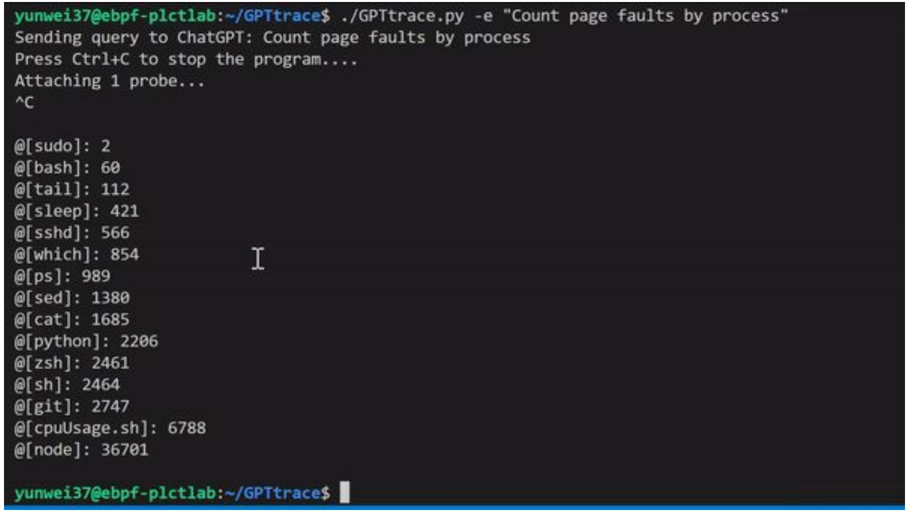
Runtime optimized for eBPF in Wasm lightweight container

- Configurable and limited eBPF WASI behavior
- Improved security with access control to lower kernel resources
- Sandbox the user space and enable eBPF plugin in tools

Wasm + eBPF + LLM

Generate eBPF with natural language and run in WebAssembly:

- The AI generated code cannot be trusted.
- The generated eBPF program needs a isolation runtime to deploy.
- WebAssembly can be lightweight and cheaper than Docker or vm.



```
yunwei37@ebpf-plctlab:~/GPTtrace$ ./GPTtrace.py -e "Count page faults by process"
Sending query to ChatGPT: Count page faults by process
Press Ctrl+C to stop the program....
Attaching 1 probe...
^C

@[sudo]: 2
@[bash]: 60
@[tail]: 112
@[sleep]: 421
@[sshd]: 566
@[which]: 854
@[ps]: 989
@[sed]: 1380
@[cat]: 1685
@[python]: 2206
@[zsh]: 2461
@[sh]: 2464
@[git]: 2747
@[cpuUsage.sh]: 6788
@[node]: 36701

yunwei37@ebpf-plctlab:~/GPTtrace$
```

With our agent and GPT4, can have 80% rate to generate simple eBPF program successfully

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