The journey of BPF from restricted C language towards extended and safe C.

Alexei Starovoitov

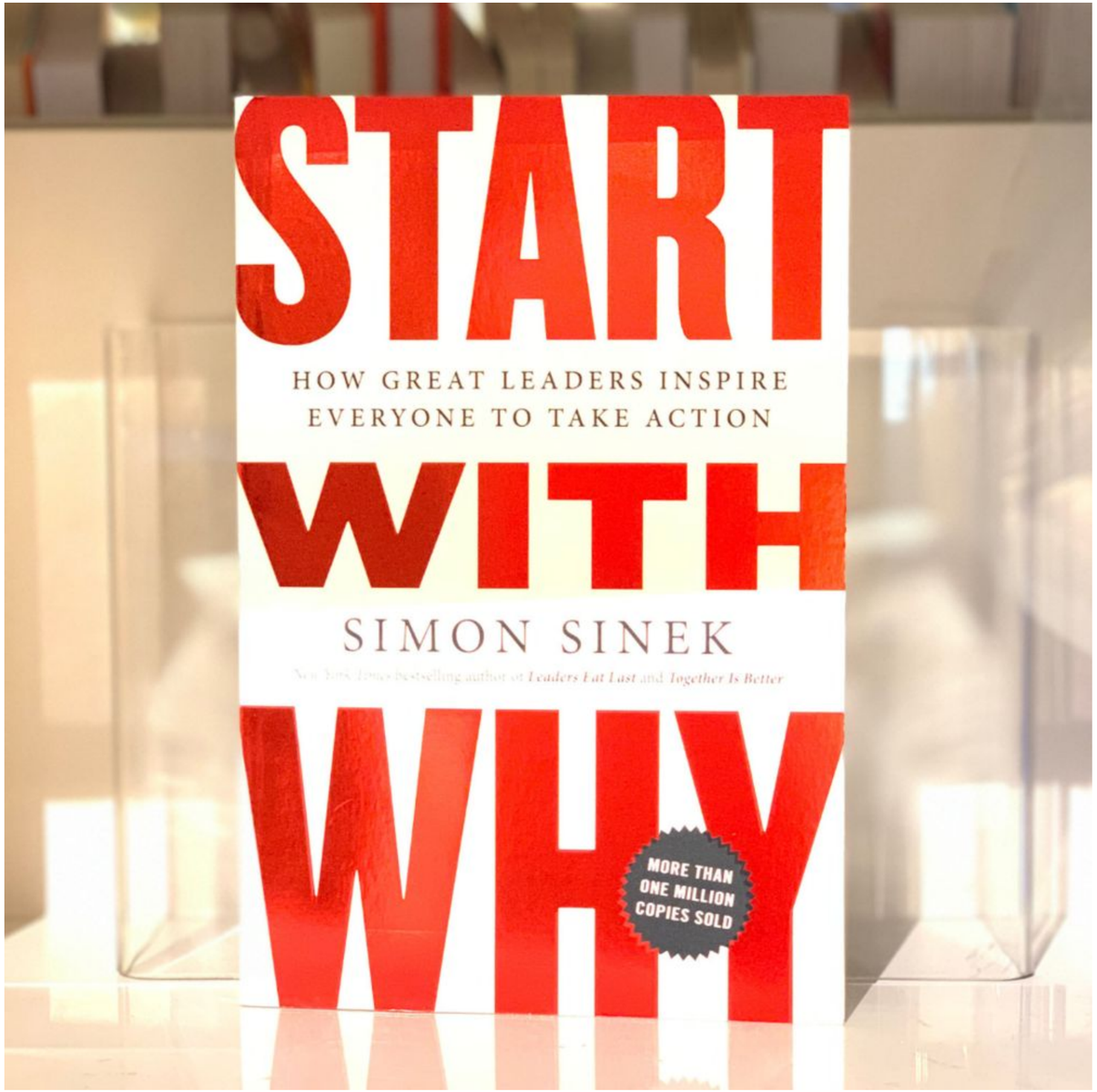
Agenda

01 Why are we working on BPF ?

02 What's wrong with C language?

03 What's wrong with kernel modules?

Why do I still work on BPF ?



The clarity of personal mission is #1 reason to enjoy the work.

What is a mission statement ?

- Tesla mission

  To accelerate the world’s transition to sustainable energy.

- Google mission

  To organize the world's information and make it universally accessible and useful.

The mission is a precise description of WHY companies do what they do.

Reasons to have a mission statement

It helps company leaders to stay focused on a mission.

It helps employees to understand the value of their work.

It helps new recruits to see whether they want to join such a company.

It is a self select mechanism for individuals and for the teams.

s/company/any open source project/

Personal mission statement

An individual needs to have clarity on his own mission to stay in the flow.

My mission statement

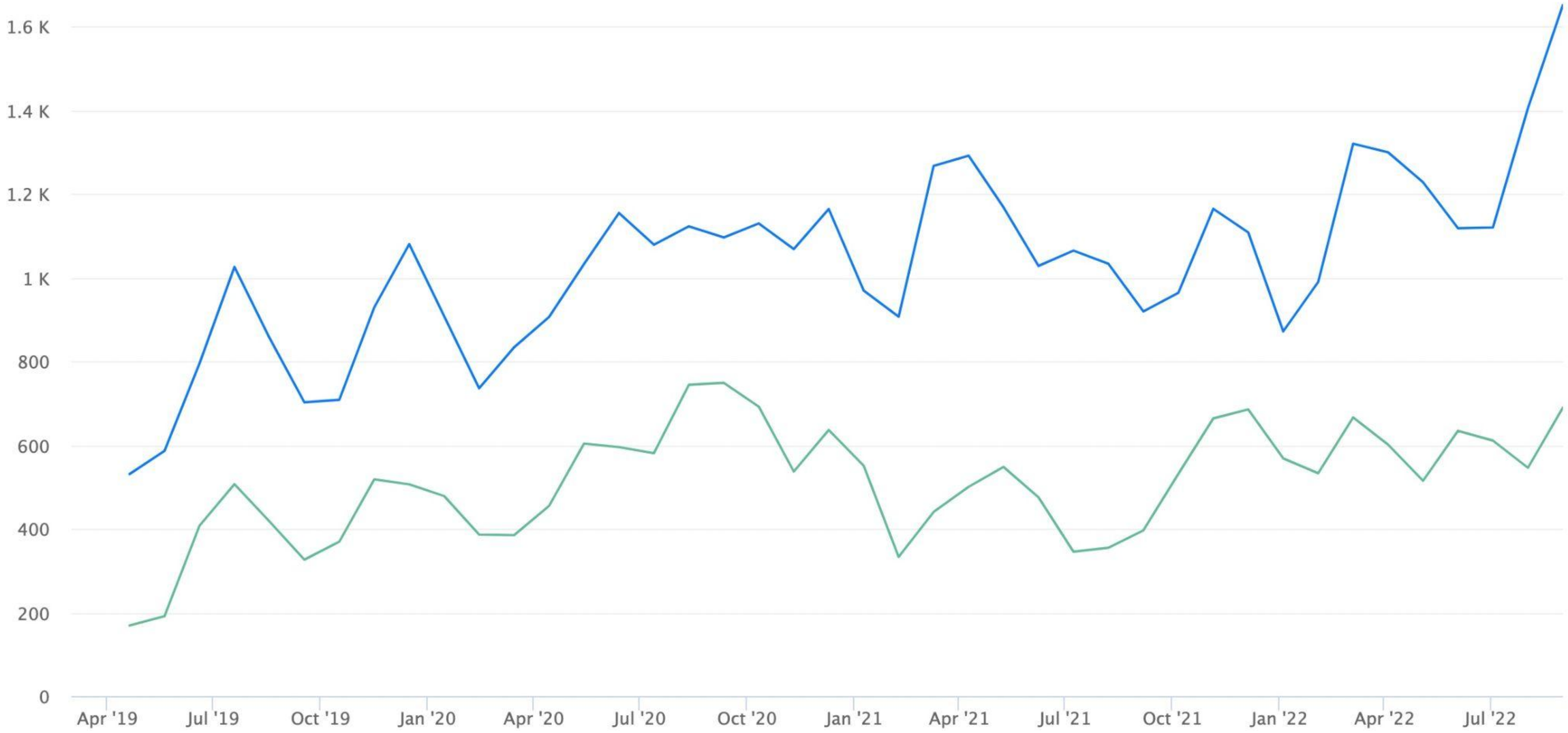
To innovate and enable others to innovate

BPF enables innovation

- BPF satisfies my own thirst for innovation
- BPF enables others to innovate
  - within BPF infra
  - in other kernel subsystems

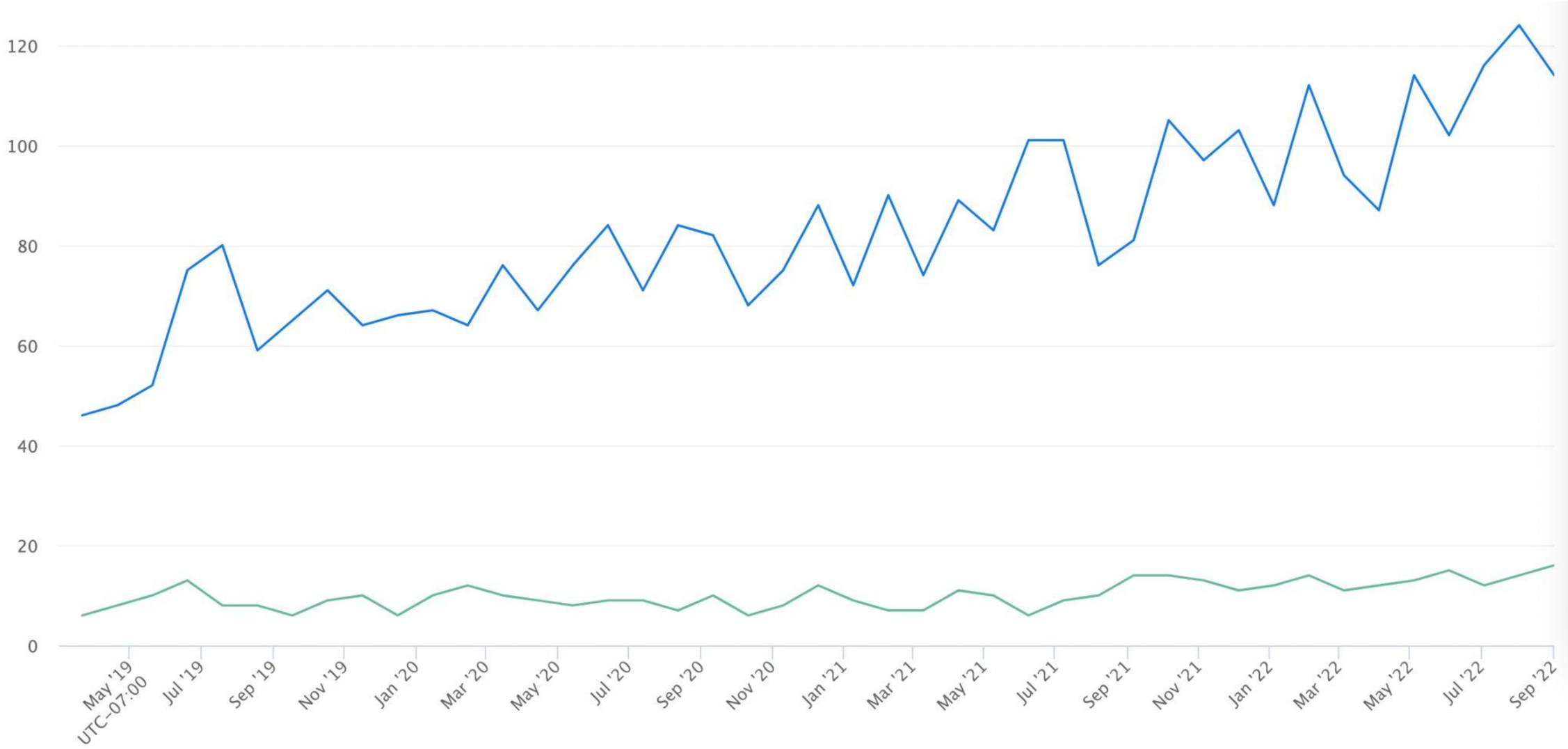
- That's why I still work on BPF !

Rate of innovation in the kernel BPF subsystem



Number of emails per month in bpf@vger (green - Meta BPF team, blue - the rest of BPF community)
~2k emails a month. 50-70 emails a day!

Rate of innovation in the kernel BPF subsystem



Number of BPF developers per month (green - Meta BPF team, blue - the rest of BPF community)

BPF enabled innovation in applications

https://ebpf.io/applications 50%

Major Applications

bcc
Toolkit and library for efficient BPF-based kernel tracing

BCC is a toolkit for creating efficient kernel tracing and manipulation programs built upon eBPF, and includes several useful command-line tools and examples. BCC eases writing of eBPF programs for kernel instrumentation in C, includes a wrapper around LLVM, and front-ends in Python and Lua. It also provides a high-level library for direct integration into applications.

GITHUB

Cilium
eBPF-based Networking, Security, and Observability

Cilium is an open source project that provides eBPF-powered networking, security and observability. It has been specifically designed from the ground up to bring the advantages of eBPF to the world of Kubernetes and to address the new scalability, security and visibility requirements of container workloads.

GITHUB WEBSITE

bpftrace
High-level tracing language for Linux eBPF

bpftrace is a high-level tracing language for Linux eBPF. Its language is inspired by awk and C, and predecessor tracers such as DTrace and SystemTap. bpftrace uses LLVM as a backend to compile scripts to eBPF bytecode and makes use of BCC as a library for interacting with the Linux eBPF subsystem as well as existing Linux tracing capabilities and attachment points.

GITHUB WEBSITE

Falco
Cloud Native Runtime Security

Falco is a behavioral activity monitor designed to detect anomalous activity in applications. Falco audits a system at the Linux kernel layer with the help of eBPF. It enriches gathered data with other input streams such as container runtime metrics and Kubernetes metrics, and allows to continuously monitor and detect container, application, host, and network activity.

GITHUB WEBSITE

Katran
A high performance layer 4 load balancer

Katran

Katran is a C++ library and eBPF program to build a high-performance layer 4 load balancing forwarding plane. Katran leverages the XDP infrastructure from the Linux kernel to provide an in-kernel facility for fast packet processing. Its performance scales linearly with the number of NIC's receive queues and it uses RSS friendly encapsulation for forwarding to L7 load balancers.

GITHUB WEBSITE

Hubble
Network, Service & Security Observability for Kubernetes using eBPF

Hubble is a fully distributed networking and security observability platform for cloud native workloads. It is built on top of Cilium and eBPF to enable deep visibility into the communication and behavior of services as well as the networking infrastructure in a completely transparent manner.

GITHUB WEBSITE

Tracee
Linux Runtime Security and Forensics using eBPF

Tracee uses eBPF technology to detect and filter operating system events, helping you expose security insights, detect suspicious behavior, and capture forensic indicators.

GITHUB

Tetragon
eBPF-based Security Observability & Runtime Enforcement

Tetragon provides eBPF-based transparent security observability combined with real-time runtime enforcement. The deep visibility is achieved without requiring application changes and is provided at low overhead thanks to smart Linux in-kernel filtering and aggregation logic built directly into the eBPF-based kernel-level collector. The embedded runtime enforcement layer is capable of performing access control on kernel functions, system calls and at other enforcement levels.

GITHUB

kubectl trace
Schedule bpftrace programs on your Kubernetes cluster

kubectl-trace is a kubectl plugin that allows for scheduling the execution of bpftrace(8) programs in Kubernetes clusters. kubectl-trace does not require installation of any components directly onto a Kubernetes cluster in order to execute bpftrace programs. When pointed to a cluster, it schedules a temporary job called trace-runner that executes bpftrace.

GITHUB

BumbleBee
OCI compliant eBPF tooling

BumbleBee simplifies building eBPF tools and allows you to package, distribute, and run eBPF programs using OCI images. It allows you to just focus on the eBPF portion of your code and BumbleBee automates away the boilerplate, including the userspace code.

GITHUB WEBSITE

pwru
eBPF-based Linux kernel network packet tracer

pwru is an eBPF-based tool for tracing network packets in the Linux kernel with advanced filtering capabilities. It allows fine-grained introspection of kernel state to facilitate debugging network connectivity issues.

GITHUB

Katran - production BPF prog written in "Restricted C"

```
SEC("xdp")
int balancer_ingress(struct xdp_md *ctx)
{
        void *data_end = (void *)(long)ctx->data_end;
        void *data = (void *)(long)ctx->data;
        struct eth_hdr *eth = data;
        __u32 eth_proto;
        __u32 nh_off;

        nh_off = sizeof(struct eth_hdr);
        if (data + nh_off > data_end)
                return XDP_DROP;
        eth_proto = eth->eth_proto;
        if (eth_proto == bpf_htons(ETH_P_IP))
                return process_packet(data, nh_off, data_end, false, ctx);
        else if (eth_proto == bpf_htons(ETH_P_IPV6))
                return process_packet(data, nh_off, data_end, true, ctx);
        else
                return XDP_PASS;
}
```

Fixed input context

Fixed return code

Early days of "Restricted C"

- All functions are __always_inline
- Single input argument
  - a pointer to context that is program type dependent. Ex: struct __sk_buff.
- No loops
- No global or static variables
- No global or static functions
- No memory allocation
- No type information

On the way to support 100% of C

- __always_inline
  - global and static functions
  - libbpf is a linker
- No loops
  - Bounded loops
  - bpf_loop, bpf_for_each_map_elem
  - Iterators
- struct __sk_buff
  - Compile Once Run Everywhere (CO-RE)

Extending C language with Symbolic Access

```
struct __sk_buff {
        __u32 len;
        __u32 pkt_type;
        __u32 mark;
        __u32 queue_mapping;
        ...
};
```

```
struct sk_buff {
        /* field names and sizes should match to those in the kernel */
        unsigned int len, data_len;
        __u16 mac_len, hdr_len, queue_mapping;
        struct net_device *dev;
        /* order of the fields doesn't matter */
        refcount_t users;
} __attribute__((preserve_access_index));
```

Instructs LLVM to generate symbolic field access instead of constant integer offsets.
Dynamic structure layout.
BPF program adjusts itself depending on the target kernel.

Extending C language with Type Information

```
SEC("tp_btf/netif_receive_skb")
int BPF_PROG(trace_netif_receive_skb, struct sk_buff *skb)
{
        p.type_id = bpf_core_type_id_kernel(struct sk_buff);
        p.ptr = skb;
        /* pretty print an skb */
        bpf_snprintf_btf(str, STRSIZE, &p, sizeof(p), 0);

        int .. = bpf_core_type_size(struct task_struct);

        bool .. = bpf_core_type_exists(struct io_uring);

}
```

BTF type id is determined at load time

Extending C language with Kconfig

```
extern unsigned long CONFIG_HZ __kconfig;
extern int LINUX_KERNEL_VERSION __kconfig;

SEC("tc")
int nf_skb_ct_test(struct __sk_buff *ctx)
{
        struct nf_conn *ct_lk;

        test_delta_timeout = ct_lk->timeout - bpf_jiffies64();
        test_delta_timeout /= CONFIG_HZ;
}
```

Unlike kernel modules the kconfig values are not known at compile time. They become known at load time.

The verifier can optimize the code with dead-code-elimination.

Extending C language with Exception Tables

```
#pragma clang attribute push (__attribute__((preserve_access_index)), apply_to = record)

struct net_device {
        int ifindex;
};

struct sk_buff {
        struct net_device *dev;
};

SEC("tp_btf/kfree_skb")
int BPF_PROG(trace_kfree_skb, struct sk_buff *skb, void *location)
{
  return skb->dev->ifindex;
}
```

Load instructions are replaced with inline version of copy_from_kernel_nofault() and exception tables generated.

Extending C language with Type Tags

```
// include/linux/compiler_types.h

# if defined(CONFIG_DEBUG_INFO_BTF) && defined(CONFIG_PAHOLE_HAS_BTF_TAG) && \
        __has_attribute(btf_type_tag)
# define BTF_TYPE_TAG(value) __attribute__((btf_type_tag(#value)))

# define __user         BTF_TYPE_TAG(user)
# define __percpu       BTF_TYPE_TAG(percpu)
```

In vmlinux that was compiled by clang the type and declaration tags go through dwarf into BTF and used by the verifier for safety analysis.

Extending C language with Operator new

```c
#define __kptr_ref __attribute__((btf_type_tag("kptr_ref")))
struct foo {
        int var;
};
struct map_value {
        // __kptr_ref tag makes C pointer behave like std::unique_ptr<struct foo>
        struct foo __kptr_ref *ptr;
};
struct {
        __uint(type, BPF_MAP_TYPE_LRU_HASH);
        __type(value, struct map_value);
} lru_map SEC(".maps");

SEC("fentry/do_nanosleep")
int nanosleep(void *ctx)
{
        // equivalent to C++ operator new that returns std::unique_ptr<struct foo>
        // std::make_unique<struct foo>();
        struct foo *p = bpf_kptr_new(bpf_core_type_id_local(typeof(*p)));

        struct map_value *v = bpf_map_lookup_elem(&lru_map, ...);

        // equivalent to C++ std::swap(v->ptr, p)
        bpf_kptr_xchg(&v->ptr, p);
}
```

Extending C language with Safe Locks, Lists, RB-trees

```
struct foo {
    struct bpf_list_node node __kernel;
    int data;
};
struct bar {
    struct bpf_rb_node node __kernel;
    int var;
};
SEC("prog_only") struct bpf_spin_lock lock;
SEC("prog_only") struct bpf_list_head head __contains(foo, node);
SEC("prog_only") struct bpf_rb_root root __contains(bar, node);

static bool cmp_less(struct bar *a, struct bar *b)
{
    return a->var < b->var;
}
void bpf_prog(struct foo *f, struct bar *b)
{
    bpf_spin_lock(&lock);
    f->data = 42;
    b->var = 0xeB9F;
    bpf_list_add(&f->node, &head);
    bpf_rb_add(&b->node, &root, cmp_less);
    bpf_spin_unlock(&lock);
}
```

Extending C language with Assertions

```
u8 cpu_to_dom_id(u32 cpu)
{
        u8 dom_id;

        assert(cpu < MAX_CPUS);
        dom_id = cpu_dom_id_map[cpu];
        assert(dom_id < MAX_DOMS);
        return dom_id;
}

void dom_add_cpu(u32 cpu, u8 dom_id)
{
        u64 *word = &dom_cpu[dom_id][cpu / 64];

        assert_within(word, dom_cpu, sizeof(dom_cpu));
        *word |= 1LLU << (cpu % 64);
}
```

assert() is a verifier aid. The verifier doesn't have to compute and enforce the bounds. The BPF program will automatically abort. The stack will be unwound, destructors called and program detached.

BPF extended C is a safer C

```c
int err_cast(struct task_struct *tsk)
{
   return((struct sk_buff *)tsk)->len;
}

int err_release_twice(struct __sk_buff *skb)
{
    struct bpf_sock_tuple tuple = {};

    struct bpf_sock *sk = bpf_sk_lookup_tcp(skb, &tuple, sizeof(tuple), 0, 0);
    bpf_sk_release(sk);
    bpf_sk_release(sk);
    return 0;
}
```

OK in C.
NOT OK in BPF C.

Early days BPF vs modern BPF

	Early BPF	Modern BPF
Execution context	rcu_read_lock + preempt_disable	rcu_read_lock_trace || rcu_read_lock + migrate_disable
API	stable uapi/bpf.h fixed input context (single argument) fixed set of helpers fixed output codes fixed set of hooks	Unstable and kernel dependent many arguments (type match) whitelisted set of kernel functions scalars and pointer return values (type match) whitelisted empty functions
New features appear as	new prog types new map types new hooks	one prog type new kernel dependent 'struct bpf_xxx' types scattered nop5
Backward compatibility	guaranteed	relies on CO-RE. May fail to load depending on kconfig, version

Like user space

Like kernel modules

What's wrong with kernel modules?

- One wrong step and panic...
  - kernel modules that were not code-reviewed (typical for out-of-tree) *have* to taint the kernel
- Have to be compiled with the kernel sources
  - Dynamic Kernel Modules Support require compiler on the host
- Once compiled becomes a binary blob with zero visibility

BPF programs are safe and portable kernel modules

- Safety is builtin
- Portability is achieved with CO-RE, kconfigs, type info
  - It's not guaranteed. BPF program may need to be adjusted to remain portable.
- Debuggable
  - All types are embedded in BPF prog and maps. Introspection is easy.
  - Pieces of source code are embedded in binary
  - The verifier understands the purpose. No way to hide what it's doing.
- EXPORT_SYMBOL_GPL == BPF kfuncs
  - there is no EXPORT_SYMBOL equivalent. All modern BPF progs are GPL.

BPF flavor of the C language is a better choice for kernel programming

Any kernel subsystem may choose to extend itself with BPF programs without touching BPF core and sending emails to bpf@vger