LSF/MM/BPF 2024

LSF/MM/BPF 2024 Schedule

Monday 13th May 2024

10:00 Polymorphic kfuncs

API may be the same for different kfuncs, but they have a different context. For example, scheduler kfunc to dispatch a task to a target CPU. Dispatching tasks may/may not be possible depending on the call path, which CPU the prog is running on, which target CPU you wish to schedule to. "Dispatch" is the same API and semantic, but needs to behave differently depending on where it's hooked.

Proposals

One way to solve this is basically to have multiple kfunc implementations for the same API, then the verifier decides how to attach them to the specific struct_ops targets at runtime.

Another way would be to build up context of struct_ops entries -> kfunc IDs -> kfunc IDs statically and compile it in at build time.

Feedback

(Attendee) Additional use case: allowing generic arguments to a kfunc. For instance, insert skb into RBTree. Or other datatypes.

10:30 Paravirt Scheduling with BPF

Context: Host + Guest VM double-scheduling. Can be inefficient as there are two schedulers and they're not coordinating to schedule processes. Goal is to allow them to coordinate scheduling using shared memory pages.

Solution has shifted design over v1,v2,v3. Latest has handshake in userspace, host injects BPF programs to the host to make scheduling decisions (design: Could alternatively be a module), and then KVM is only responsible for relaying the scheduling requests to the host BPF prog.

Some discussion about a new map type for shared memory between host and guest.

Feedback

David V.: No BPF prog in guest in v3 design? What if there was actually a comms channel between host vs. guest BPF progs. Could potentially reduce scheduler changes required in kernel.

11:30 Sched Ext

Safe hot swap for the scheduler to enable rapid experimentation; bespoke policies based on target workloads. Promising results from a range of users from Meta, Google, Canonical, Igalia/Valve. Community is growing. Latest state is a (still contentious) v6 patchset on the list.

PSA: https://github.com/sched-ext/scx - Shared pool of schedulers, low process for iteration. If you have an idea, welcome to submit one and iterate as you like.

Lots of potential for further expansion. Eg Hierarchical cgroup schedulers could move us towards per-process scheduler optimizations.

Feedback

What's next?

- Iterating on dynamic stack sizing. Going through six layers of cgroups can create big stacks.
- Struct_ops per cgroup

Q: Tail end latency vs. Generic latency

- Obviously for gaming p99 makes a big difference, given eg 60 fps requirements
- David's sense is we should be able to improve on a range of use cases.
- Brendan: There's presumably also a lot of effort into existing schedulers. Don't want to lose the benefit of all of the scrutiny that went into those.

12:00 BPF struct_ops + sched_ext

Various improvements to struct_ops have been introduced in order to help with some of the sched_ext work. Going through examples of some of the newer features, such as variable shadowing, NULL pointers, etc.

12:30 LLVM improvements for verification

We want to improve some of the LLVM code generation to allow useful developer tooling for BPF. For instance, enabling code coverage. Some info like LIKELY() hints are lost by the time the code reaches BPF.

Proposals

- Developers could write explicit assertions which are retained down into BPF
- Push the verification up front to LLVM (at least partially) to improve ease-of-use. Not changing the model; verifier still needs to run at load time.
    - PREVAIL vs. Linux verifiers have different approaches
    - Alexei: Integrating compiler is probably a non-starter. We can always just run the verifier directly after compile. The trick is how to feed / correlate the information back up.

- Alan: Giving the compiler feedback that specific optimization passes break verification for instance could be useful.
- Area to explore: Verifier running at IR level?

14:30 Compiled BPF with GCC

Since late last year there was a rewrite of GCC BPF backend due to some challenges in dealing with weird expectations from BPF. All BPF selftests are compiling with GCC 🎉. Still 108 running but failing. Releases with this support coming soon for binutils (~July), gcc (~Aug)

- CPU v4 flag in ELF header
- Gcc supports BPF inline assembly given it's already used in BPF programs these days
- builtin_memmove/memcpy/memset defaults to 1024 bytes, threshold configurable

David V: Should GCC/Clang emit bounded loop iterations?

- This is more of a feature of the ecosystem, not an instruction. For instance no support on Windows.
- Requires BTF.
- Alexei: "not a good idea", just plain unrolled for loop and let verifier complain

Big milestone was achieved that BPF selftests compile with gcc BPF, still ~100 tests failing verifier.

Upcoming work

- 32-bit subregister constraints & warnings.
- Obey BPF memory model in compiler (mem & compiler barriers)
- Maybe-goto support
- BTF pruning given BTF information is currently huge (clang only generates BTF for what's used by the program), kernel -> from DWARF. New gcc option for -fprune-btf to only emit what's needed by BPF program
- BTF and LTO: needs toolchain support to merge/deduplicate BTF at link time
- Support for BTF dumping in GNU binutils, merging & deduplicating BTF in linker

Maintenance model

Verified targets are a new challenge for compilers to maintain. Optimizations or bugfixes in the compiler can result in different code being generated, leading to unverifiable objects.

What's the experience from clang/LLVM community? Bugfix release breakage so far never happened, in clang trunk we catch breakages via BPF CI and either fix LLVM or verifier. Nightly clang testing is effective to catch these bugs early.

Android case - kernel and userspace lags behind by perhaps multiple years for devices out in the field.

BTF for non-C languages? BTF is basically built on the C type system. Causes problems for instance if developers are writing BPF programs in Rust. Missing complex enums for example. Sooner or later we'll likely need to solve this issue. David: This could be part of the BTF standardization effort @ IETF.

15:30 Cross-Platform BPF compiler issues

There are groupings of instructions for compilers,runtimes to claim compliance with the standard. Walked through examples on how to extend sets (add new groups) and deprecate (add new groups for deprecated insns, new group with an "exclude" for the deprecated insns)

From a cross-platform perspective, having BPF "CPU version v4" doesn't work well. IETF groups is a more generic mechanism to express sets of instructions that a particular compiler or runtime may support.

Processor-Specific ABI issues

How many registers are there, which ones are scratched vs saved across calls, which register is stack pointer, how large is stack, how much stack space does bpf2bpf calls get. All these can be different across runtimes.

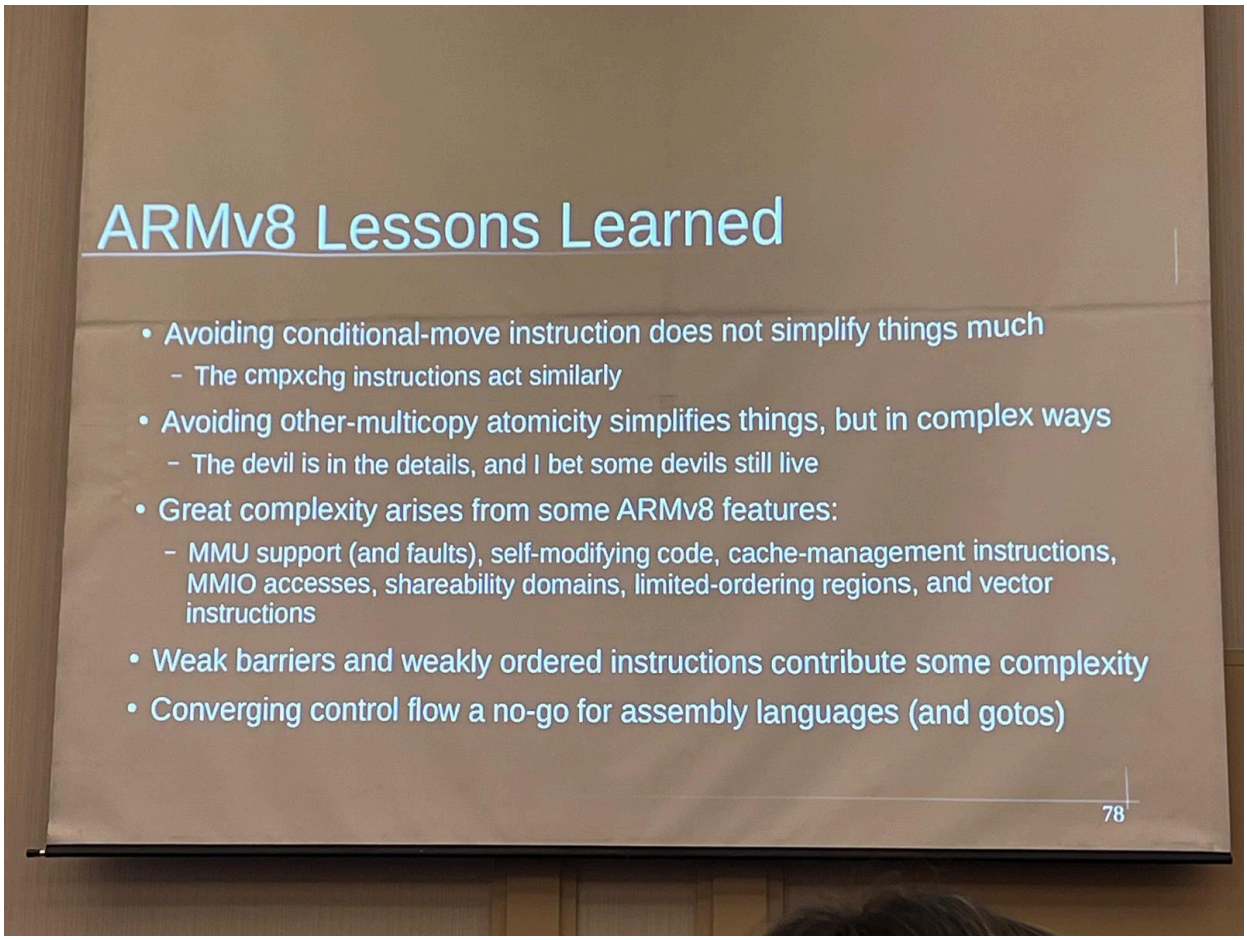
16:30 Instruction-level BPF memory model

BPF dependency ordered before -> went through armv8 and ppc models. BPF has no a-priori knowledge thus should follow PPC & ARM. JITs should not do smart stuff.

Hazard Ordered Before

Different PPC CPUs see stores in different orders, ARMv8 CPUs not doing stores agree on order.

Lessons learned from ARMv8 see below:



Herd7 BPF litmus demo

17:30 BPF performance testing

Windows doesn't do JIT, so can generate more optimal code from the C compiler (some followup discussion, brief notes below). LPM is similar for lookups, but Windows implements LPM as a set of hashtables which improves update performance (not expected to be the common case for LPM). The tests are multi-core, and kernel LPM apparently has single lock so that explains the update performance difference. LRU has been a struggle to match. Linux vs. Windows performance comparisons are very provisional, as Windows hardware is self-hosted runners with tight control vs. GitHub runners for Linux which can introduce noise.

Discussion

JIT performance metrics seem surprising. Case is a very short program, like two instructions - call and return. So number of register saves etc. can have high impact for that. Also the Linux numbers are less reliable just due to where they're running in GH infra.

Tuesday 14th May 2024

09:30 Modernize BPF for the next 10 years

eBPF has evolved over time to target use cases ranging from supporting flexible line-rate network processing to adaptable tracing programs that can make symbolic access into kernel headers. Various functionality has been introduced over time to target a range of use cases.

10 years in 10 minutes

There are some tricks/techniques that are regularly used in BPF code that are no longer necessary, and we need to do better at evangelizing that developers don't need these any more. For instance, always inlining all functions. Also loops have made huge advances through bounded loops, loop helpers, open-coded iterators and supporting cond_break.

Cond_break similar to cond_resched concept, break out of the loop if looking too much.

Looking towards the future

Interfaces are changing, too - change from helpers towards kfunc. New callback approaches coming in as non-UAPI "struct-ops", such as sched-ext, hid-bpf, fuse-bpf, qdisc-bpf. Building data structures with references to native kernel data structures via kptr.

Never ending race to add new algorithms to BPF, thus BPF arena as a solution to share memory between user space and kernel, goal: sandboxing for access to memory arena, so verifier does not need to do static analysis on the access within the arena which allows for building more complex algorithms such as regex etc.

We need infrastructure to share BPF code as libraries going forward, concept of Rust to distribute libraries as source only.

More work on BPF locks needed, so far spin lock too restricted and has bugs which risks BPF infra to cause deadlocks.

bpf_tail_call() is a hack for lack of indirect calls, indirect call instruction already exists in clang/LLVM, verifier needs to support it which is work in progress (Anton’s work).

New attribute no_caller_saved_registers needed for kfuncs which don’t take arguments to avoid R0-R5 spill/fill.

Calling convention back in the days was close to x86, but other archs are more expensive, so work is needed to make it more efficient on arm64/riscv64/s390.

Reloadable kernel module for verifier.

Discussion

How to propagate "modern bpf" examples better - how can we do this? Crowd-source docs, build them as a community.

Concerns around bpf instruction encoding

10:30 BPF LSM Updates

New maintainer, active development driven by several OSS projects. New recently is BPF token. This talk will focus on trusted BPF, static calls in LSM, calling kfuncs from LSM.

We've talked about "signed BPF" for some time, but the signing is just a representation of trust. Let's call it "Trusted BPF" instead. What does this mean? Trusted BPF loaders are signed to ensure that loaders will not load malicious programs.

Use fs-verity with a signed digest of the loader that gets attached to the task struct, then when the application makes BPF calls, LSM to enforce that the loader is signed with authority to make those BPF changes. Demo.

Static calls - helps to mitigate newly uncovered CPU bugs.

kfuncs - call for buy-in with other subsystems.

Discussion

Can you deny BPF calls without a particular BPF token? Use LSM hooks.
Signed shared libraries for loader? KP: Statically link your trusted loader
Andrii: FYI BPF token cannot be created in init user_ns currently (for backwards compatibility reasons)
The solution for KP's use case doesn't solve untrusted pid 0 with dynamically loaded libraries (systemd). Lots of back and forth about this.

11:30 Tetragon Auditing / Enforcement

Building a BPF-based networking stack, how would we do this? Walking through different L7 cases: Firstly, for observability avoid multiple stack walks that are purely there to redirect traffic into the observability stack, and secondly to eliminate double-encrypt.

Overview of some of the current issues with L7 parsers at Sockmap.

kTLS: Now in OpenSSL 3.0, coming soon to Go/Java.

L3 - Would be nice to have better wildcarding lookup primitives. Policy lookups often end up looking like a TCAM-shaped problem, but we don't have TCAM on a CPU.

Several interesting areas for further development in L2 - eg understanding NIC queues in a driver-independent way, qdisc occupancy

12:00 Hypervisor-enforced code integrity

Branch code integrity checks can be enabled on amd64, aarch64 using Control Flow Integrity (CFI). CFI using a hash based on the type parameters for a function call, which is not 100% - if the type is common with another target function, that can be called into. Also the hash function is not always reliable.

12:30 Implementing BPF_PROG_LOAD_FD

High level goal is to introduce modern BPF into Android. Android frames BPF into Networking, System and Vendor targets. Depending on these "users", there are prog type, hook attach allowlists. Currently, the core BPF functionality is loaded on boot and can be trusted, but Android would like to expand possibilities for vendor extension with their own BPF programs.

Feedback

Pushing the loader into the kernel before standardizing ELF etc. doesn't seem like a good idea. Still in flux, we shouldn't limit adaptability by pushing this into kernel.

14:30 Segmented Stacks for BPF progs

Allocating a larger stack before bpf_prog_run is called. Would not work for tail calls.

JIT proposals:

- Non-sleepable - percpu static allocation
- Sleepable - kmalloc. Feedback: Should move this to load time.

Discussion

Alexei: Runtime allocation is a nonstarter.

Aim for percpu per program. Other proposals are more complicated for unclear benefit. Global per cpu? No must be per prog.

15:00 Value tracking in BPF verifier

Wish: verifier simpler, more people would understand, more testing would happen. Agni and other tooling somewhat help. Complexity getting bigger.

Proposal 1: Looking to combine value tracking for signed and unsigned ranges to simplify internal logic that switches on the different types.

Proposal 2: Refactor value tracking to combine cases so there is less code that calls a common abstraction that then has all the details for handling tracking for various number types.

15:30 Static keys and jump labels

Static keys, more generic implementation: Needs static key jumps, a way to track sets of instructions impacted by the static key, and a way to manipulate the static key.

Introducing a new map type `BPF_MAP_TYPE_INSN_SET`, one map instance per static key, which is a list of offsets to instructions impacted by the static keys. This map won't be referenced by the BPF prog itself, so we need a way to bind it into the prog. For this, add a bind_fd_array + count during prog load.

Besides static keys this map allows to implement a new `goto *Rx` instruction for BPF. This will allow for llvm/libbpf to optimize large switches and to add [a restricted] support for label addresses (&&label, see [1], [2]).

Discussion

David V: sched-ext could use kfuncs for configuring the static keys. Anton: May need some care for manipulating the target code. Something about lock access.

Updates of instructions to jump based on static keys are not atomically applied to the program, so not appropriate for configuration changes.

16:30 BPF CI and local testing

s390 was slow due to runners being slow. s390x now cross compiling reduced build from 17min to 7min. P99 is now 534s for test suite. More load on x86 servers for cross compilation.

P90 from 50min to 5min now. Mid April: Now KVM on top of KVM for s390, reduced time from 20min to just 5min.

Moved to VMtest in Jan, build in container, run test in VM using same FS which eliminated the need to build a rootfs for the VM for each test.

Flaky tests now less flaky. Veristat against Meta object files.

Riscv would need full emulation, might be slow unless we get some concrete hardware to run tests on.

17:00 BPF dev hacks

Local development, able to get shell in VM for tests using danobi/vmtest. Manu has bash scripts for running and bisecting bpf selftests. Writeup: https://chantra.github.io/bpfcitools/bpfci-troubleshooting.html

Can use `gh run download` to pull VM artifact from CI run. Useful to reproduce CI failure.

Manu also has cross-compile examples on the same website.

17:30 BPF Conformance - handling undefined opcodes

As part of handling IETF standardization of BPF, it's useful to have a conformance testsuite that (a) confirms that the Linux implementation is aligned with the spec as we formalize it, and (b) other runtimes match the behaviour.

Some discussion about the extent that conformance should go to. Dave opinion to categorize the current tests as ISA tests, then develop additional sets for additional IETF documents.

Wednesday 15th May 2024

09:30 kprobe multi updates

Session for entry/exit attachment without needing 2 links. For both uprobes and kprobe links supported. Session cookie associated with it.

10:00 faster uprobes

Goal is to speed up uprobes. Override with breakpoint instruction on all instances of the address. Going through mmap of the file. Breakpoint will trigger breakpoint handler in kernel, before resuming user space the original instruction needs to be either emulated (not all insn can, push, jmp, call, nop can) or single stepped.

Single stepping: copy original instruction in special area, single stepping original insn, then returning back. Much slower than emulating.

New uretprobe syscall: easy to install given trampoline is there on function exit. 14-31% speedup on Intel, 7-10% on AMD CPUs vs trap.

Uprobe speedup: more tricky, no code yet. Idea: replace breakpoint insn with jmp to trampoline which would execute the syscall. USDT speedup is just nop, so easy to install. With nop5 we can easily change it into call to trampoline

Issues: safe 5byte update on parallel execution, original insn execution. 5byte insn has 4 byte offset, so limited where to jump must be in nearby.

Nop5 could have compat issues with older kernels, nop1+nop4 is better.

Early test: speed up is 2.5x.

nop1+nop4 → int3 + offset update (int3 would emulate jump) → jmp+offset update

Steven: issues, if sth jumps into the offset, application might crash. Masami working on optimizing kprobes. USDT is an easier case since we control layout, thus update would be safer given noone should jump between the two knobs.

Issue with trampolines: 4byte offset tricky given no access to whole userspace.

Indirect jump insn with 8byte offset exists! Maybe could be used.

10:30 Evolution of stack trace capture with BPF

Overview of stack map, how to use it in BPF program and push it through perf RB. Build ID support, 20 bytes which kernel can fill out.

Quirks:

- 32bit of stack ID convenient
- Cannot request kernel+user capture in one go even though kernel supports it
- No way to know how many IP addresses we captured. No zero initialization.
- Automatic stack deduplication, important as it has huge implications.

The good:

- Design favors space efficiency and performance

The bad:

- Collisions do happen (!) and are unavoidable.
- Few hacks FAST_STACK_CMP (only compare hashes) / RESUE_STACKID (override hashes -> loose data or corrupt data)
- Meta production never uses RESUE_STACKID.
- Stack deducplication is racy, meaning BPF adds to it, user space might remove from map

General: not suited for long running sessions. In practice: double buffering approach to make it more or less reliable.

Request: allow users to manage memory instead of map doing dedup. Evolution: bpf_get_stack().

New async API is needed: kernel stack traces work right now, user stack traces are different and can be postponed.

11:30 Perf tools issues

BPF usage in perf -> skeletons and libbpf.

Unpriviledged perf, perf requies CAP_PERFMON but does it also need CAP_BPF for accessing sample data only? Unpriv would be nice, options? Probably not given HW speculation issues.

Lock symbolization.. tricky.

12:00 pahole

–btf_features, reproducible builds, DECL_TAG for kfuncs, distilled base BTF

Btf_features: simplify Makefile in kernel, older pahole ignores unknown features.
Reproducible builds: every time you build different output, reproducible kernel image wanted. Parallel DWARF loading, but encode BTF in same DWARF CU order of vmlinux. Less than 100ms diff for reproducible/non-reproducible builds.

pfunct as a tool to generate kfunc function declarations.

Resilient split BTF for out of tree module. Problem is that the module is built with references to specific BTF IDs for the target kernel, but if you load another different kernel then the new kernel may have entirely different IDs. Need a way to "relocate" module BTF IDs.

Datatype profiling:

perf mem, perf c2c.

Kernel functions doing memloads, instruction latency for symbols.. Not in cache, thus causing cache pressure.

Perf c2c.. cacheline oriented and shows cacheline offset, but doesn’t resolve type. Only shows line-number. Tool helps to detect cases where CPU cache is thrashing due to reads/writes in multiple threads. Use output to reorder structure fields for better cacheline access patterns.

Integrating https://github.com/capstone-engine/capstone for disassembly. Expected to be faster, but doesn't cover all cases yet - fall back to objdump.

Perf report with type,symbol find overhead on datastructures in specific functions. Type,typeoff finds fields inside structs to gather data on wheather fields should be on same cacheline or not.

Perf annotate –data-type -> shows samples mixed with pahole struct output! Super useful!

12:30 Rethinking BPF conntrack

Cilium BPF CT: 5-tuple tracking with BPF LRU map, used for LB and policy enforcement.

Concerns: GC is imperfect. Runs on some timer, complicated logic for when it runs to balance CPU / accuracy. Tuple reuse for connections after initial connection can lead to LB skew (reuses LB decision from first connection, rather than rebalancing for each connection).

Can we use socket storage to tie the fate of the conntrack entries more closely into application socket connections? Socket storage can't replace the need for conntrack table altogether due to awkwardness mapping from NATed addresses back to socket. But we could perhaps at least improve GC.

Problem: Want to use state in cgroup storage at socket cleanup time / bpf_sk_storage_free() in order to hook additional cleanup. However, the verifier currently prevents referencing the same storage from that hook. Answer: Trying to prevent recreating an sk_storage entry while deleting the socket. Martin: Tracing is not intended for this sort of cleanup logic. Aditi: That's only when setting the CREATE flag, no? Could we just loosen the restriction to not allow this flag? Martin: Would have to look back into it.

Martin: Could we create another hashtable with the Service address that points back to the socket (via kptr)? Aditi: Storage size problem would be reintroduced here.

Aditi: Tracing is just for PoC - question is whether we could have a more formal callback to handle storage cleanup.

Stan: inet_socket_close hook, look up socket storage from there. Aditi: Concern is which sockets are "relevant" sockets to perform this cleanup.

John: Upper bound of map size problem is caused by the GC problem, so that specific issue would be less of a problem if GC worked better. So the factor of map size problem may not be a hard blocker on the solution space.

14:30 BPF qdisc

Goal: Simplify qdisc development, make it more flexible. v8 of series moving towards struct_ops approach.

Discussion about reference counting skbs, argument seems to be that qdisc works quite differently compared to existing points wrt reference ownership, so the verifier needs to be taught to be more smart about how references are handled. Some "exclusive reference" handling.

Adding sk_buffs to bpf collections

Alexei: Declarative approach for qdisc hierarchy makes more sense than imperative.

Alexei: Break the series down to deliver incremental value.

May need some explicit destructors to properly free skb refs, to ensure they don't get stuck in the RBTree

15:00 BPF Thrift RPC parsing

Looking into BPF MemCache (BMC) - initial academic proposal had various limitations; for example, small bounded data, only GET via UDP, unknown memcache configuration. Interesting idea, but can we make it work in a real-world environment? Alexei: Apparently Kumar made this actually work with upstream OSS Memcache.

Challenges for Meta - custom distributed version of memcache, protocol is Thrift over TCP, and there's a whole bunch of userspace functionality like ACLs, logging, LB, overload protection, etc which would also need to be accounted for in a BMC implementation.

Proposal: If copies are the primary saving, then userspace zerocopy may provide the same performance characteristics, invalidating the whole idea..

Interesting idea: If traversing PCIe bus is the primary cost and there is typically DMA from NIC to host to final destination (GPU/NVMe), what if BPF can run before DMA to host and then instead do DMA once directly to the target destination?

15:30 Updates on tcx, netkit and global socket iterator

Unified, more efficient tc datapath optimized for BPF - Linux 6.6, Cilium 1.15
Netkit for cloud-native virtual NIC optimized for BPF - Kernel 6.7 (Ubuntu 24.04), Cilium 1.16.

Couple of minor hiccups with netkit-l2 mode integrating with Cilium - expectation to set the MAC, and skb->pkt_type handling breaks local L7 transparent proxying.

Global socket iterators - progress so far is cleaning up host sockets, but there's still no way to iterate over child netns sockets. Investigating loops over nets -> loop over sockets.

Referenceless iteration?

16:30 BPF Standardization Updates

ISA - first deliverable for the BPF working group in the IETF. In last call - moving to publication. RFC publication estimated for late July.
After ISA, we can start to work on various additional work items from the charter in parallel.

Discussed about ability to use registers 11-15. Encoding allows this, but no implementations have this and the proposed standard does not allow this. Back and forth, but ultimately we can publish an update as/when the need arises.

Next steps: Verifier expectations, BTF, conventions/guidelines for portable BPF binaries. Some discussion about the conventions - for example, compiler expectations; psABI; ELF specification.

17:00 eBPF Foundation Updates

eBPF foundation is a directed fund under the Linux Foundation. Currently 9 member companies are paying membership dues which are distributed by the Governing Board. eBPF Steering Committee (BSC) are the technical advisory board for the foundation, populated by maintainers from the community including many present at this conf. BSC provides suggested priorities to the GB who ultimately decide how to best distribute the funds.

Went through overview of activities 2023 and planned 2024, including event sponsorship, statements of work for CI, arm64 support, research funding opportunities.

Discussion that tooling improvements for compilers/toolchain should perhaps be targeted to support multiple software, for instance rather than defining a SoW to improve LLVM, define that feature X should be added to LLVM and GCC.

Discussion point about defining a security threat model for BPF, security whitepaper to help companies to understand how BPF fits into their infrastructure and can improve their security posture.