

The story of BPF

A practical guide to land patches

- What BPF stands for?
 - Does it matter ?
 - The name given to an instruction set 30 years ago by Steven McCanne and Van Jacobson.

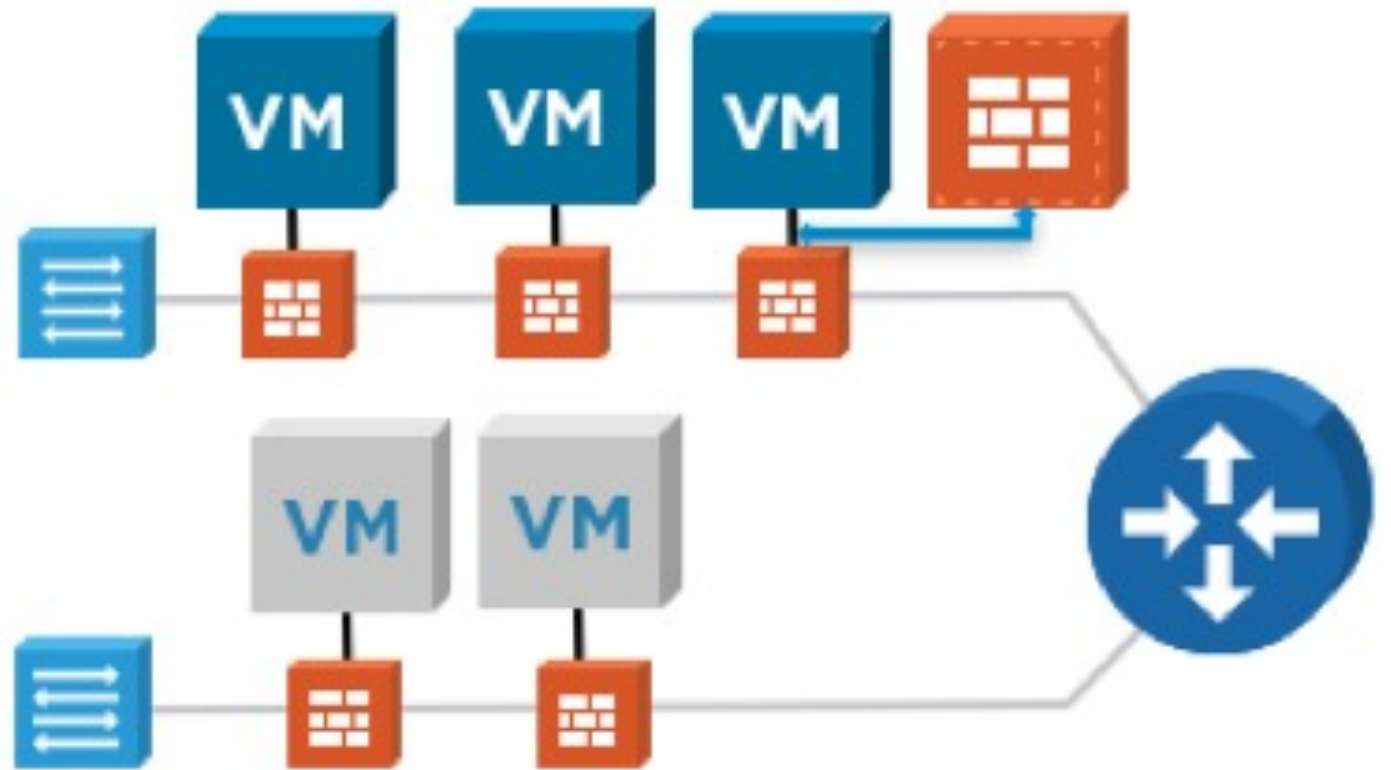
- Little they knew that in 2011 a startup decides to revolutionize Software Defined Networking.



- Physical -> Virtual
 - Servers -> Virtual Machines
 - Technology: hypervisor
 - KVM, QEMU
 - Networking hardware -> Virtual routers, switches, firewalls
 - Technology: iovisor

One physical server:

- 5 VMs
- 1 router
- 2 switches
- 5 firewalls



Traditional approach

- VM -> kvm.ko
- Virtual router -> vrouter.ko
- Virtual switch -> vswitch.ko
- Virtual firewall -> vfirewall.ko


PLUMgrid's solution v1

- iovisor.ko
 - switch, router, firewall – binary blobs of x86 code
 - pushed to a host by a remote controller
 - Including 3rd party NAT, packet captures, etc

- What can go wrong?
- After 4Gbyte of networking traffic the kernel would crash
- 32-bit overflow ?
- Race condition ?

- What can go wrong?

arch/x86/Makefile: KBUILD_CFLAGS += -mno-red-zone

 stackoverflow

AboutProductsFor Teams

Search...

Home

PUBLIC

Questions

Tags

Users

Companies

Why can't kernel code use a Red Zone

Asked 7 years, 9 months ago Modified 5 years, 3 months ago Viewed 5k times

▲

26

It is highly recommended when creating a 64-bit kernel (for x86_64 platform), to instruct the compiler not to use the 128-byte Red Zone that the user-space ABI does. (For GCC the compiler flag is `-mno-red-zone`).

▼

The kernel would not be interrupt-safe if it is enabled.

PLUMgrid's solution v2

- Verify x86 code
- The verifier was born.



- Verification pain points with x86 asm
 - Lots of ways to compute an address.
 - Lots of memory access instructions.
- Solution: reduced x86 instruction set.
 - Hack GCC x86 backend.
- The first iovisor.ko had the verifier and no JIT.

PLUMgrid's solution v3

- New instruction set (x86 like)
- GCC backend that emits binary code
- iovisor.ko
 - The verifier for this instruction set
 - JIT to x86
 - No interpreter

How to upstream iovisor.ko ?

- Talk to key people when possible
- New instruction set is scary to compiler folks
- Even scarier to kernel maintainers
- Solution: make it look familiar

Make it look familiar

- Is there an instruction set in the kernel with similar properties?
 - BPF, iptables, netfilter tables, inet_diag
- Make new instruction set look as close as possible to BPF
 - Reuse opcode encoding and 8-byte size of insn
 - Call it 'extended' BPF

```
struct sock_filter { /* Filter block */
    __u16 code; /* Actual filter code */
    __u8 jt; /* Jump true */
    __u8 jf; /* Jump false */
    __u32 k; /* Generic multiuse field */
};
```

```
struct bpf_insn {
    __u8 code; /* opcode */
    __u8 dst_reg:4; /* dest register */
    __u8 src_reg:4; /* source register */
    __s16 off; /* signed offset */
    __s32 imm; /* signed immediate constant */
};
```

Next steps

- Read netdev@vger mailing list for 6 month
- Understand the land
- Identify key people
- And post the jumbo patch? No.

Build reputation

- Find lockdep report in your area of interest.

```
[ 56.766097] Possible unsafe locking scenario:
[ 56.766097]
[ 56.780146]         CPU0
[ 56.786807]         ----
[ 56.793188]         lock(&(&vb->lock)->rlock);
[ 56.799593]         <Interrupt>
[ 56.805889]         lock(&(&vb->lock)->rlock);
[ 56.812266]
[ 56.812266] *** DEADLOCK ***
[ 56.812266]
[ 56.830670] 1 lock held by ksoftirqd/1/13:
[ 56.836838]  #0:  (rcu_read_lock){.+.+.}, at: [<ffffffff8118f44c>] vm_unmap_aliases+0x8c/0x380
```


My 1st kernel patch:

Move module_free() of x86 JITed memory into a worker.

```
+static void bpf_jit_free_deferred(struct work_struct *work)
+{
+    struct sk_filter *fp = container_of(work, struct sk_filter, work);
+    unsigned long addr = (unsigned long)fp->bpf_func & PAGE_MASK;
+    struct bpf_binary_header *header = (void *)addr;
+
+    set_memory_rw(addr, header->pages);
+    module_free(NULL, header);
+    kfree(fp);
+}
+
void bpf_jit_free(struct sk_filter *fp)
{
    if (fp->bpf_func != sk_run_filter) {
-        unsigned long addr = (unsigned long)fp->bpf_func & PAGE_MASK;
-        struct bpf_binary_header *header = (void *)addr;
-
-        set_memory_rw(addr, header->pages);
-        module_free(NULL, header);
+        INIT_WORK(&fp->work, bpf_jit_free_deferred);
+        schedule_work(&fp->work);
    }
}
```

Keep building reputation...

my kernel commit #5

commit dcd9df56b4a6c9437fc37dbc9cee94a788f9b0c4

Author: Alexei Starovoitov <ast@kernel.org>

Date: Tue Nov 19 19:12:34 2013 -0800

ipv4: fix race in concurrent ip_route_input_slow()

CPUs can ask for local route via ip_route_input_noref() concurrently. if nh_rth_input is not cached yet, CPUs will proceed to allocate equivalent DSTs on 'lo' and then will try to cache them in nh_rth_input via rt_cache_route()

Most of the time they succeed, but on occasion the following two lines:

```
orig = *p;
```

```
prev = cmpxchg(p, orig, rt);
```

in rt_cache_route() do race and one of the cpus fails to complete cmpxchg.

But ip_route_input_slow() doesn't check the return code of rt_cache_route(),

so dst is leaking. dst_destroy() is never called and 'lo' device

refcnt doesn't go to zero, which can be seen in the logs as:

```
unregister_netdevice: waiting for lo to become free. Usage count = 1
```

Adding mdelay() between above two lines makes it easily reproducible.

Fix it similar to nh_pcpu_rth_output case.

Fixes: d2d68ba9fe8b ("ipv4: Cache input routes in fib_info nexthops.")

Signed-off-by: Alexei Starovoitov <ast@plumgrid.com>

Signed-off-by: David S. Miller <davem@davemloft.net>

```
diff --git a/net/ipv4/route.c b/net/ipv4/route.c
```

```
index f428935c50db..f8da28278014 100644
```

```
--- a/net/ipv4/route.c
```

```
+++ b/net/ipv4/route.c
```

```
@@ -1776,8 +1776,12 @@ out:    return err;
```

```
    rth->dst.error= -err;
```

```
    rth->rt_flags   &= ~RTCF_LOCAL;
```

```
}
```

```
-    if (do_cache)
```

```
-        rt_cache_route(&FIB_RES_NH(res), rth);
```

```
+    if (do_cache) {
```

```
+        if (unlikely(!rt_cache_route(&FIB_RES_NH(res), rth))) {
```

```
+            rth->dst.flags |= DST_NOCACHE;
```

```
+            rt_add_uncached_list(rth);
```

```
+        }
```

```
+    }
```

```
    skb_dst_set(skb, &rth->dst);
```

Finally post eBPF patchset

Did it work?

Finally post eBPF patchset

Nope. It was rejected.

What is the biggest maintainer's concern?

UAPI !

Need a plan B for eBPF

Add eBPF without exposing it in UAPI

How?

Need a plan B for eBPF

Add eBPF without exposing it in UAPI

Answer: Make existing code faster

Rewrite existing BPF interpreter

Thankfully it was easy to make it 2 times faster.

10% of the speedup came from eBPF instruction set itself.

90% of the speedup from jump-threaded implementation.

That's how 'internal BPF' was created.

Need to disambiguate two BPFs.

Daniel Borkmann came up with a name 'classic BPF'.

The state of BPF in May 2014:

- cBPF converter to iBPF (internal BPF)
- Interpreter that runs iBPF
- x86, sparc, arm JIT compilers from iBPF to native code

eBPF doesn't exist yet. There is no verifier either.

Where to apply iBPF 'engine' ?

The concepts of the verifier, maps, helpers were proposed.
Programs suppose to run from `netif_receive_skb`.

The networking use case still struggles.

Arguments against:

- [ei]BPF instruction set is not extensible. Should be using TLV ?
- u8 opcode looks small. eBPF 2.0 will be coming ?
- The verifier is not supported by static analysis theory.
- It bypasses networking stack.

If the mountain will not come to Mohammed...

Strategy: **Compromise** on networking, pivot eBPF into tracing.

Strategy: **Make it look familiar**.

F - filter.

Proposal to 'filter' perf events.

Reuse verifier, maps, helpers concepts, but instead of network stack execute programs from perf events and kprobes.

Unfortunate trade-off: clean design vs upstreamability.

BPF programs attached to kprobes return 0 to filter out kprobe.

‘filter out’ == don’t output kprobe event into perf ring buffer.

Since then all kprobe+bpf programs return 0.

Made sense to upstream, since program == filter.

Pointless and confusing ‘feature’ long term.

Strategy: Make existing code faster.

Demonstrate that BPF tracing 'filter' is faster than predicate tree walker.

Demonstrate that BPF TC 'classifier' is faster than TC u32 classifier.

Finally on September 26, 2014

- `cbd357008604` (bpf: verifier (add ability to receive verification log), 2014-09-26)
- `51580e798cb6` (bpf: verifier (add docs), 2014-09-26)
- `0a542a86d73b` (bpf: handle pseudo BPF_CALL insn, 2014-09-26)
- `09756af46893` (bpf: expand BPF syscall with program load/unload, 2014-09-26)
- `db20fd2b0108` (bpf: add lookup/update/delete/iterate methods to BPF maps, 2014-09-26)
- `99c55f7d47c0` (bpf: introduce BPF syscall and maps, 2014-09-26)



eBPF is learning to walk.

[89aa075832b0](#) (net: sock: allow eBPF programs to be attached to sockets, 2014-12-01)

[e2e9b6541dd4](#) (cls_bpf: add initial eBPF support for programmable classifiers, 2015-03-01)

[2541517c32be](#) (tracing, perf: Implement BPF programs attached to kprobes, 2015-03-25)

Are we done?

Are we done?

Kernel was just the beginning.

Landing new backend in LLVM was just as difficult.

LLVM community

- Most developers have direct write access
- Anyone can revert anyone else's commit
- [s/MAINTAINERS/CODE_OWNERS.TXT/](#)
- Back then LLVM was using SVN
- Phabricator for diffs
- C++ in CamelStyle

LLVM community

- No UAPI concerns
 - Compiler internals are changing a lot
 - Backward incompatible backend changes is not a concern
- Kernel UAPI doesn't justify or restrict LLVM choices
- Continuous integration and testing is mandatory
- Build bots run tests right after diff lands
 - Backends have to contribute build bots
 - Many operating systems
 - Approved diffs might get reverted and re-landed many times
- Monthly meetup at Tied House, Mountain View, CA

LLVM BPF backend

Differential Revision: <http://reviews.llvm.org/D6494>

llvm-svn: 227008

llvm/CODE_OWNERS.TXT	4 +
llvm/include/llvm/ADT/Triple.h	1 +
llvm/include/llvm/IR/Intrinsics.td	1 +
llvm/include/llvm/IR/IntrinsicsBPF.td	22 +++++
llvm/lib/Support/Triple.cpp	8 ++
llvm/lib/Target/BPF/BPF.h	22 +++++
llvm/lib/Target/BPF/BPF.td	31 ++++++
llvm/lib/Target/BPF/BPFAsmPrinter.cpp	87 ++++++
llvm/lib/Target/BPF/BPFCallingConv.td	29 ++++++
llvm/lib/Target/BPF/BPFFrameLowering.cpp	39 ++++++
llvm/lib/Target/BPF/BPFFrameLowering.h	41 ++++++
llvm/lib/Target/BPF/BPFISelDAGToDAG.cpp	159 ++++++
llvm/lib/Target/BPF/BPFISelLowering.cpp	642 ++++++
...	
llvm/lib/Target/LLVMBuild.txt	2 +-

69 files changed, 4644 insertions(+), 1 deletion(-)

Proposed in Dec 2014

<http://reviews.llvm.org/D6494>
took 2 month to land in Jan 2015 as
experimental backend.

Subscribers



ealfie (Ezequiel Alfie)



majnemer (David Majnemer)



chandlerc (Chandler Carruth)



echristo (Eric Christopher)



joerg (Joerg Sonnenberger)



pete (Pete Cooper)



rengolin (Renato Golin)



kristof.beyls (Kristof Beyls)



arsenm (Matt Arsenault)



t.p.northover (Tim Northover)



tstellarAMD (Tom Stellard)



llvm-commits (Mailing List "llvm-commits")



aemerson (Amara Emerson)

To graduate BPF backend from experimental status

- It has to have users
- It needs more than one developer
- Developers must help with tree wide refactoring
- Build bot

BPF backend in GCC

- Emits BPF byte code directly. Upstream blocker.
 - Unlike LLVM GCC doesn't have integrated assembler. GCC has to emit plain text
 - Would have to make libbfd/gas/ld work
- Being lazy as an upstream strategy sometimes works too
- In 2019 Oracle GCC engineers implemented everything

Steps that did NOT help to land patches

- Present at the conferences
- Describe amazing future

Summary: Strategies to land patches

- Learn the community
- Understand maintainer's concerns
- Build the **reputation**
- Make new ideas **look familiar**
- Make existing code **faster**
- Split big ideas into small building blocks
- Be prepared to **compromise**

Slide 42

What questions do you have?