PLUMgrid

BPF – in-kernel virtual machine

BPF is

- Berkeley Packet Filter
- low level instruction set
- kernel infrastructure around it
  - interpreter
  - JITs
  - maps
  - helper functions

Agenda

- status and new use cases
- architecture and design
- demo

extended BPF JITs and compilers

- x64 JIT upstreamed
- arm64 JIT upstreamed
- s390 JIT in progress
- ppc JIT in progress
- LLVM backend is upstreamed
- gcc backend is in progress

extended BPF use cases

1. networking
2. tracing (analytics, monitoring, debugging)
3. in-kernel optimizations
4. hw modeling
5. crazy stuff...

1. extended BPF in networking

- socket filters
- four use cases of bpf in openvswitch (bpf+ovs)
  - bpf as an action on flow-hit
  - bpf as fallback on flow-miss
  - bpf as packet parser before flow lookup
  - bpf to completely replace ovs datapath
- two use cases in traffic control (bpf+tc)
  - cls – packet parser and classifier
  - act – action
- bpf as net_device

2. extended BPF in tracing

- bpf+kprobe – dtrace/systemtap like
- bpf+syscalls – analytics and monitoring
- bpf+tracepoints – faster alternative to kprobes
- TCP stack instrumentation with bpf+tracepoints as non-intrusive alternative to web10g
- disk latency monitoring
- live kernel debugging (with and without debug info)

3. extended BPF for in-kernel optimizations

- kernel interface is kept unmodified. subsystems use bpf to accelerate internal execution
- predicate tree walker of tracing filters -> bpf
- nft (netfilter tables) -> bpf

4. extended BPF for HW modeling

- p4 – language for programing flexible network switches
- p4 compiler into bpf (userspace)
- pass bpf into kernel via switchdev abstraction
- rocker device (part of qemu) to execute bpf

5. other crazy uses of BPF

- 'reverse BPF' was proposed
  - in-kernel NIC drivers expose BPF back to user space as generic program to construct hw specific data structures
- bpf -> NPUs
  - some networking HW vendors planning to translate bpf directly to HW

classic BPF

- BPF - Berkeley Packet Filter
- inspired by BSD
- introduced in linux in 1997 in version 2.1.75
- initially used as socket filter by packet capture tool tcpdump (via libpcap)

classic BPF

- two 32-bit registers: A, X
- implicit stack of 16 32-bit slots (LD_MEM, ST_MEM insns)
- full integer arithmetic
- explicit load/store from packet (LD_ABS, LD_IND insns)
- conditional branches (with two destinations: jump true/false)

Ex: tcpdump syntax and classic BPF assembler

- tcpdump –d 'ip and tcp port 22'

```
(000) ldh      [12]                       // fetch eth proto
(001) jeq      #0x800   jt 2  jf 12  // is it IPv4 ?
(002) ldb      [23]                       // fetch ip proto
(003) jeq      #0x6     jt 4  jf 12  // is it TCP ?
(004) ldh      [20]                       // fetch frag_off
(005) jset     #0x1fff jt 12 jf 6   // is it a frag?
(006) ldxb     4*([14]&0xf)               // fetch ip header len
(007) ldh      [x + 14]                   // fetch src port
(008) jeq      #0x16    jt 11 jf 9   // is it 22 ?
(009) ldh      [x + 16]                   // fetch dest port
(010) jeq      #0x16    jt 11 jf 12 // is it 22 ?
(011) ret      #65535                     // trim packet and pass
(012) ret      #0                         // ignore packet
```

Classic BPF for use cases

- socket filters (drop or trim packet and pass to user space)
  - used by tcpdump/libpcap, wireshark, nmap, dhcp, arpd, ...
- in networking subsystems
  - cls_bpf (TC classifier), xt_bpf, ppp, team, ...
- seccomp (chrome sandboxing)
  - introduced in 2012 to filter syscall arguments with bpf program

Classic BPF safety

- verifier checks all instructions, forward jumps only, stack slot load/store, etc
- instruction set has some built-in safety (no exposed stack pointer, instead load instruction has ‘mem’ modifier)
- dynamic packet-boundary checks

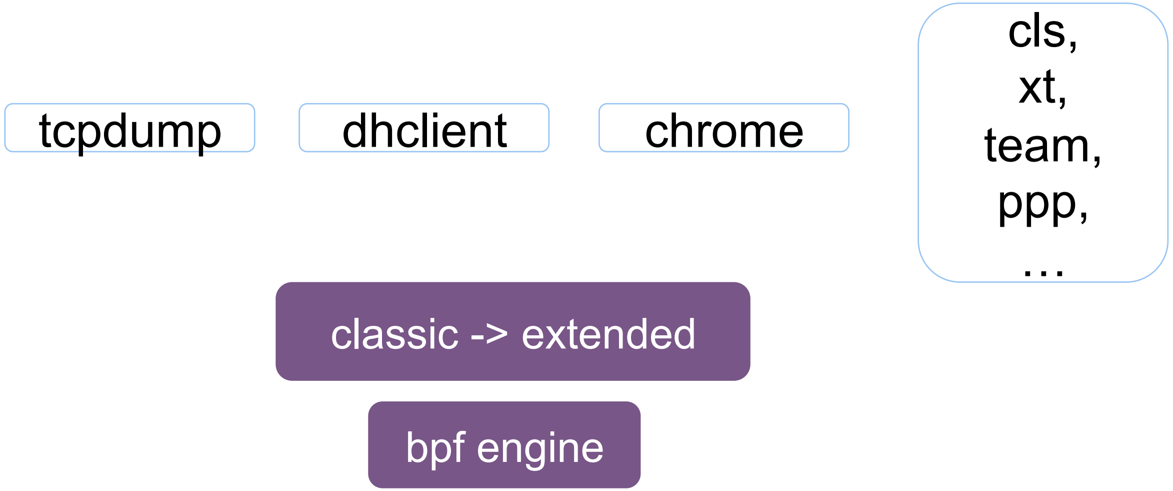
Classic BPF extensions

- over years multiple extensions were added in the form of ‘load from negative hard coded offset’
- LD_ABS -0x1000 – skb->protocol
  LD_ABS -0x1000+4 – skb->pkt_type
  LD_ABS -0x1000+56 – get_random()

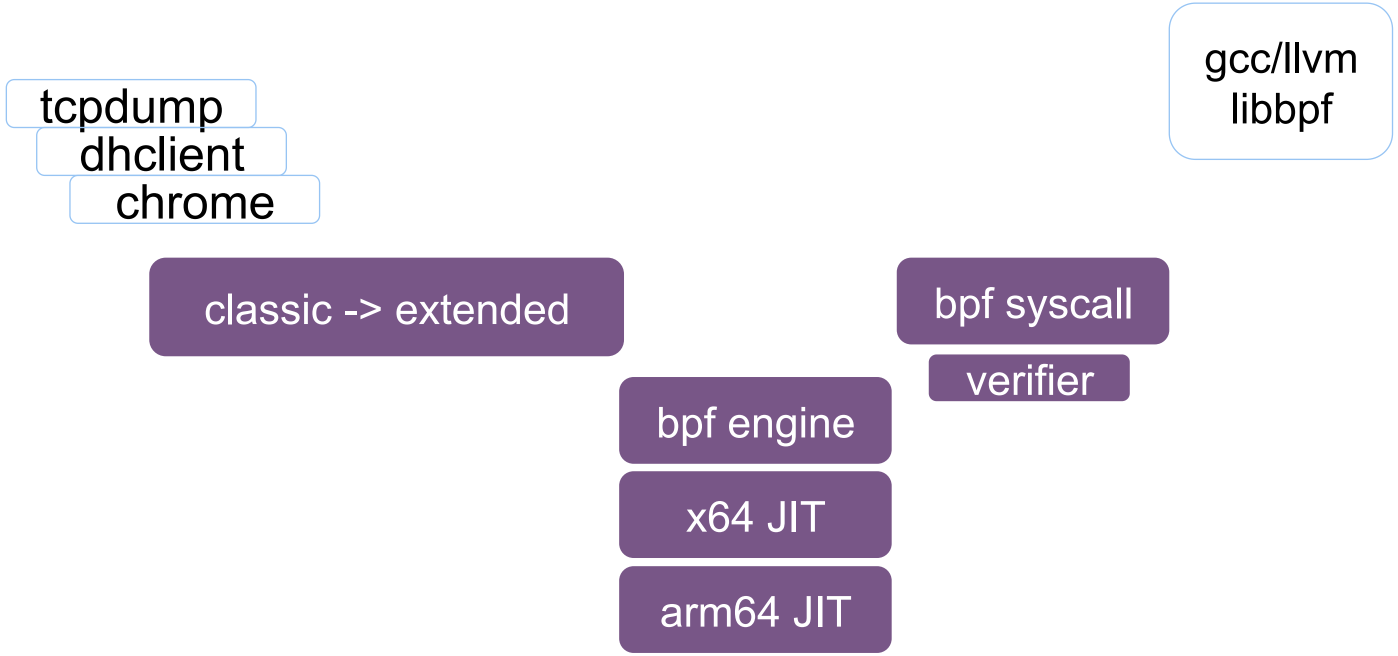
Extended BPF

- design goals:
  - parse, lookup, update, modify network packets
  - loadable as kernel modules on demand, on live traffic
  - safe on production system
  - performance equal to native x86 code
  - fast interpreter speed (good performance on all architectures)
  - calls into bpf and calls from bpf to kernel should be free (no FFI overhead)

in kernel 3.15



in kernel 3.18



Early prototypes

- Failed approach #1 (design a VM from scratch)
  - performance was too slow, user tools need to be developed from scratch as well
- Failed approach #2 (have kernel disassemble and verify x86 instructions)
  - too many instruction combinations, disasm/verifier needs to be rewritten for every architecture

Extended BPF

- take a mix of real CPU instructions
  - 10% classic BPF + 70% x86 + 25% arm64 + 5% risc
- rename every x86 instruction ‘mov rax, rbx’ into ‘mov r1, r2’
- analyze x86/arm64/risc calling conventions and define a common one for this ‘renamed’ instruction set
- make instruction encoding fixed size (for high interpreter speed)
- reuse classic BPF instruction encoding (for trivial classic->extended conversion)

extended vs classic BPF

- ten 64-bit registers vs two 32-bit registers
- arbitrary load/store vs stack load/store
- call instruction

Performance

- user space compiler ‘thinks’ that it’s emitting simplified x86 code
- kernel verifies this ‘simplified x86’ code
- kernel JIT translates each ‘simplified x86’ insn into real x86
  - all registers map one-to-one
  - most of instructions map one-to-one
  - bpf ‘call’ instruction maps to x86 ‘call’

Extended BPF calling convention

- BPF calling convention was carefully selected to match a subset of amd64/arm64 ABIs to avoid extra copy in calls:
- R0 – return value
- R1..R5 – function arguments
- R6..R9 – callee saved
- R10 – frame pointer

Mapping of BPF registers to x86

R0 – rax	return value from function
R1 – rdi	1st argument
R2 – rsi	2nd argument
R3 – rdx	3rd argument
R4 – rcx	4th argument
R5 – r8	5th argument
R6 – rbx	callee saved
R7 - r13	callee saved
R8 - r14	callee saved
R9 - r15	callee saved
R10 – rbp	frame pointer

calls and helper functions

- bpf ‘call’ and set of in-kernel helper functions define what bpf programs can do
- bpf code itself is a ‘glue’ between calls to in-kernel helper functions
- helpers
  - map_lookup/update/delete
  - ktime_get
  - packet_write
  - fetch

BPF maps

- maps is a generic storage of different types for sharing data between kernel and userspace
- The maps are accessed from user space via BPF syscall, which has commands:
  - create a map with given type and attributes
    map_fd = bpf(BPF_MAP_CREATE, union bpf_attr *attr, u32 size)
  - lookup key/value, update, delete, iterate, delete a map
- userspace programs use this syscall to create/access maps that BPF programs are concurrently updating

BPF compilers

- BPF backend for LLVM is in trunk and will be released as part of 3.7
- BPF backend for GCC is being worked on
- C front-end (clang) is used today to compile C code into BPF
- tracing and networking use cases may need custom languages
- BPF backend only knows how to emit instructions (calls to helper functions look like normal calls)

Extended BPF assembler

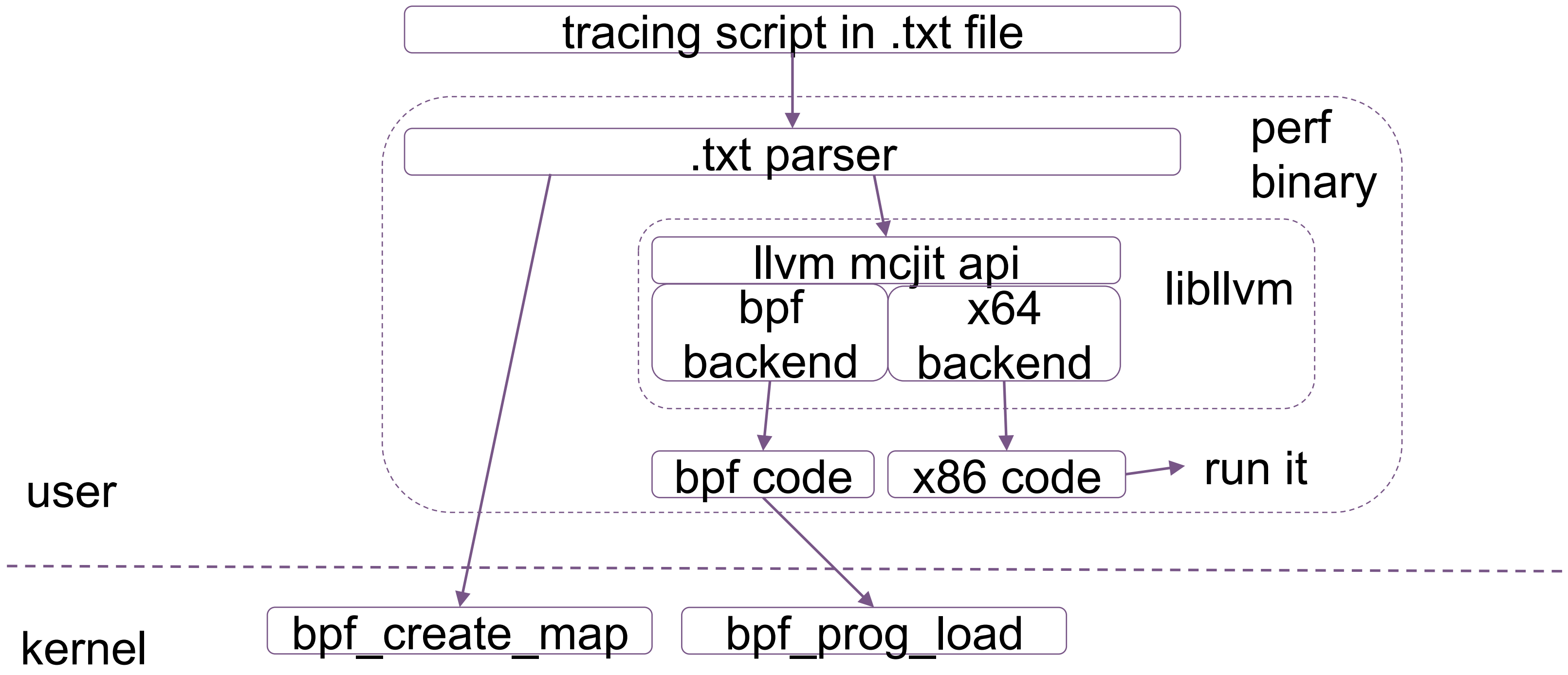
```
int bpf_prog(struct bpf_context *ctx)
{
 u64 loc = ctx->arg2;
 u64 init_val = 1;
 u64 *value;

 value = bpf_map_lookup_elem(&my_map, &loc);
 if (value)
   *value += 1;
 else
   bpf_map_update_elem(&my_map, &loc,
                        &init_val, BPF_ANY);
 return 0;
}
```

compiled by LLVM from C to bpf asm

```
0: r1 = *(u64 *)(r1 +8)
1: *(u64 *)(r10 -8) = r1
2: r1 = 1
3: *(u64 *)(r10 -16) = r1
4: r1 = map_fd
6: r2 = r10
7: r2 += -8
8: call 1
9: if r0 == 0x0 goto pc+4
10: r1 = *(u64 *)(r0 +0)
11: r1 += 1
12: *(u64 *)(r0 +0) = r1
13: goto pc+8
14: r1 = map_fd
16: r2 = r10
17: r2 += -8
18: r3 = r10
19: r3 += -16
20: r4 = 0
21: call 2
22: r0 = 0
23: exit
```

compiler as a library



BPF verifier (CFG check)

- To minimize run-time overhead anything that can be checked statically is done by verifier
- all jumps of a program form a CFG which is checked for loops
  - DAG check = non-recursive depth-first-search
  - if back-edge exists -> there is a loop -> reject program
  - jumps back are allowed if they don’t form loops
  - bpf compiler can move cold basic blocks out of critical path
  - likely/unlikely() hints give extra performance

BPF verifier (instruction walking)

- once it’s known that all paths through the program reach final ‘exit’ instruction, brute force analyzer of all instructions starts
- it descents all possible paths from the 1st insn till ‘exit’ insn
- it simulates execution of every insn and updates the state change of registers and stack

BPF verifier

- at the start of the program:
  - type of R1 = PTR_TO_CTX
    type of R10 = FRAME_PTR
    other registers and stack is unreadable
- when verifier sees:
  - ‘R2 = R1’ instruction it copies the type of R1 into R2
  - ‘R3 = 123’ instruction, the type of R3 becomes CONST_IMM
  - ‘exit’ instruction, it checks that R0 is readable
  - ‘if (R4 == 456) goto pc+5’ instruction, it checks that R4 is readable and forks current state of registers and stack into ‘true’ and ‘false’ branches

BPF verifier (state pruning)

- every branch adds another fork for verifier to explore, therefore branch pruning is important
- when verifiers sees an old state that has more strict register state and more strict stack state then the current branch doesn't need to be explored further, since verifier already concluded that more strict state leads to valid ‘exit’
- two states are equivalent if register state is more conservative and explored stack state is more conservative than the current one

unprivileged programs?

- today extended BPF is root only
- to consider unprivileged access:
  - teach verifier to conditionally reject programs that expose kernel addresses to user space
  - constant blinding pass

BPF for tracing

- BPF is seen as alternative to systemtap/dtrace
- provides in-kernel aggregation, event filtering
- can be 'always on'
- must have minimal overhead

BPF for tracing (kernel part)

```
struct bpf_map_def SEC("maps") my_hist_map = {
        .type = BPF_MAP_TYPE_ARRAY,
        .key_size = sizeof(u32),
        .value_size = sizeof(u64),
        .max_entries = 64,
};
```

sent to kernel as bpf map via bpf() syscall

```
SEC("events/syscalls/sys_enter_write")
int bpf_prog(struct bpf_context *ctx)
```

name of elf section - tracing event to attach via perf_event ioctl

```
{
        u64 write_size = ctx->arg3;
        u32 index = log2(write_size);
        u64 *value;

        value = bpf_map_lookup_elem(&my_hist_map, &index);
        if (value)
                __sync_fetch_and_add(value, 1);
        return 0;
}
```

compiled by llvm into .o and loaded via bpf() syscall

BPF for tracing (user part)

```
u64 data[64] = {}; u32 key; u64 value;

for (key = 0; key < 64; key++) {
        bpf_lookup_elem(fd, &key, &value);
        data[key] = value;
        if (value && key > max_ind)
                max_ind = key;
        if (value > max_value)
                max_value = value;
}
printf("syscall write() stats\n");
```

user space walks the map and fetches elements via bpf() syscall

```
syscall write() stats
    byte_size       : count     distribution
      1 -> 1        : 9        |***************************           |
      2 -> 3        : 0        |                                      |
      4 -> 7        : 0        |                                      |
      8 -> 15       : 2        |*****                                 |
     16 -> 31       : 0        |                                      |
     32 -> 63       : 10       |******************************        |
     64 -> 127      : 12       |*************************************|
    128 -> 255      : 1        |**                                    |
    256 -> 511      : 2        |*****                                 |
```

Cute Tracing Logos

(Brendan Gregg's slide)

ftrace

perf_events

SystemTap

ktap

LTTng

dtrace4linux

Ponies by Deirdré Straughan, using: http://generalzoi.deviantart.com pony creator

Extended BPF

PLUMgrid

demo