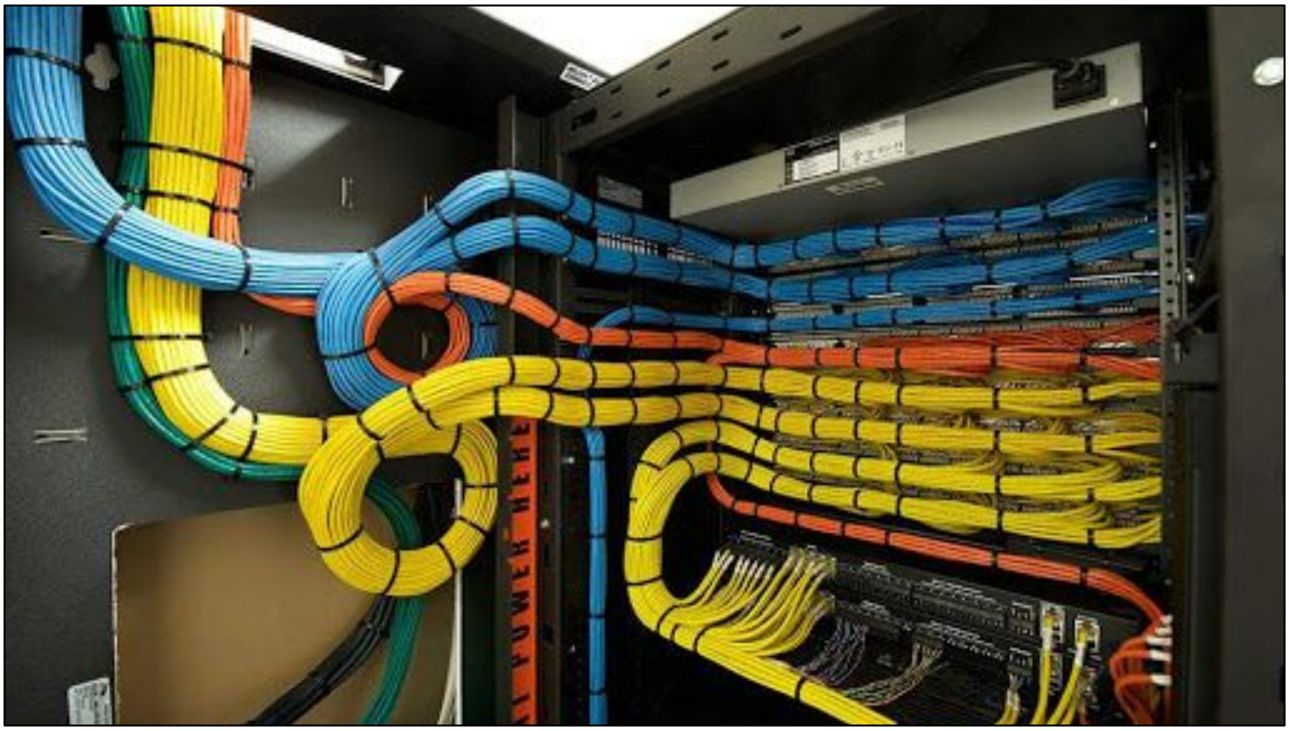
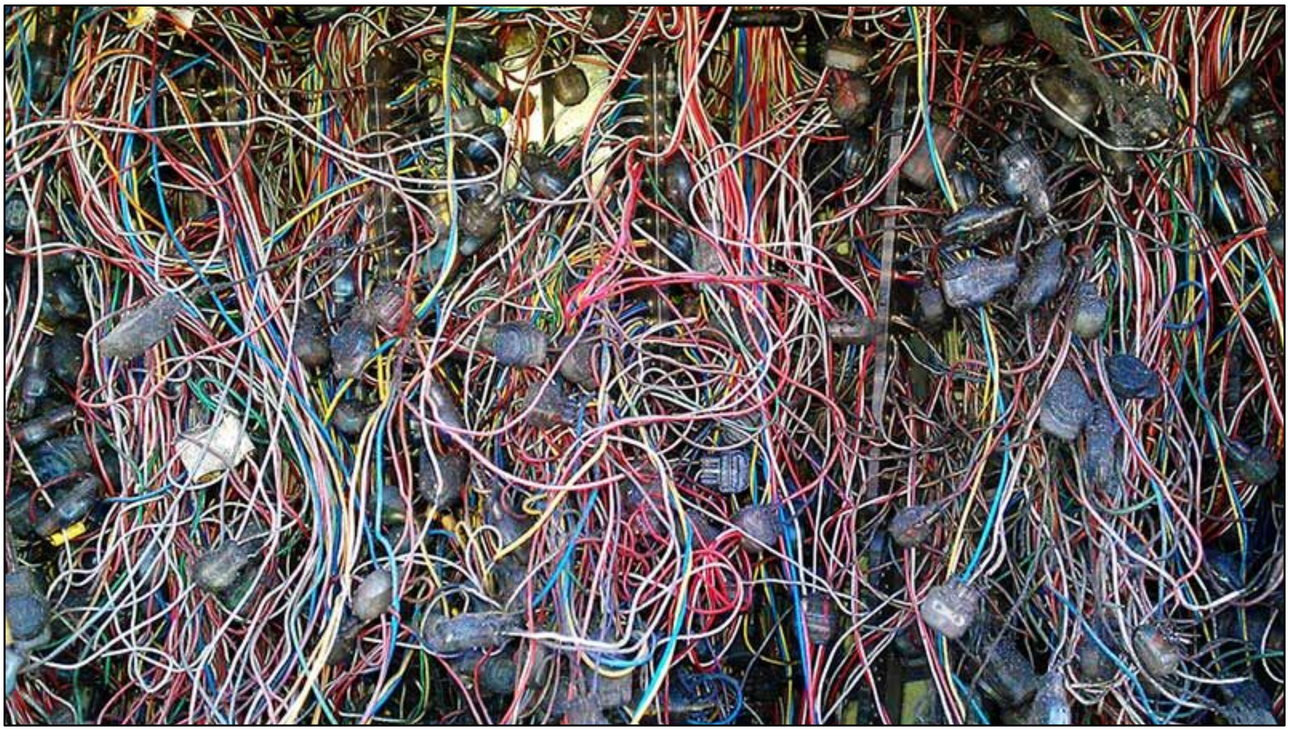
# Dynamic tracing of builds

- **Why!?**
- **Tech**
    - What is eBPF in the first place?
    - Instrumenting the kernel
    - Finding the relevant syscalls
    - Capturing & processing the data
- **Demo of capture + what's next analysis**



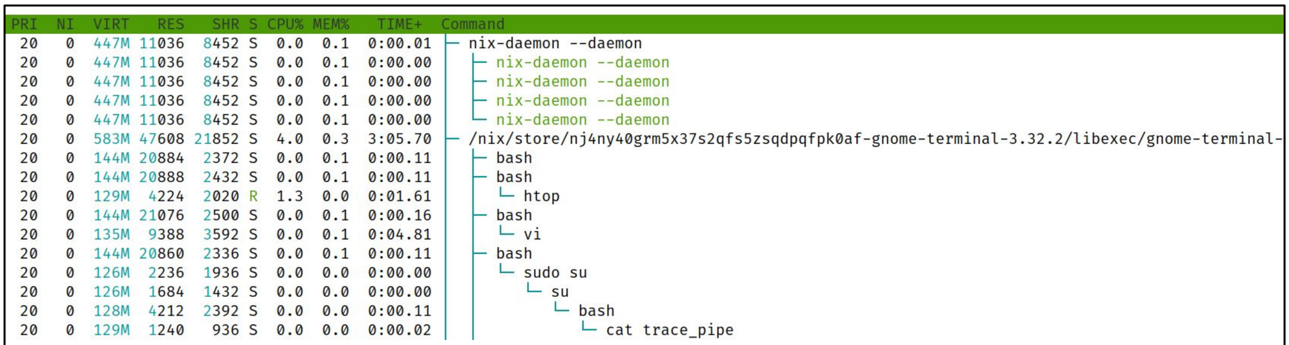
- Hopefully, when you build software, you’re careful about tracking dependencies, so that you’re able know which components depend on other components.
- However, this is not always what we encounter at clients.



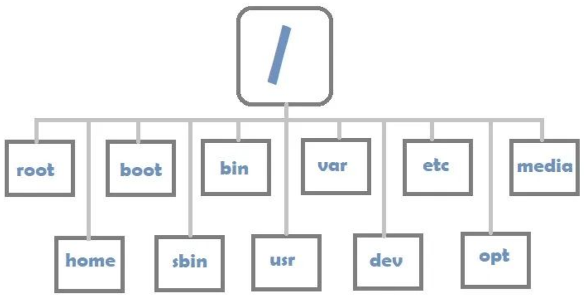
- This is a picture that represents how the build process of one of our clients looked like.
- They didn't understand what was there, and how it all connected.
- Which dependencies where in there, when they were build.
- During the project we found more and more unexpected dependencies.

- This was very hard to trace statically, and we missed things whilst observing the system.
- Humanly speaking, finding out all these dependencies upfront was impossible.

- I craved a tool that could analyze this, like wireshark can be used to analyze network traffic.
- A tool that could dump all the relevant details of a build in a file, which then can be filtered, and visualized.



```
PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 20   0  447M 11036  8452 S  0.0  0.1  0:00.01 ├─ nix-daemon --daemon
 20   0  447M 11036  8452 S  0.0  0.1  0:00.00 │  ├─ nix-daemon --daemon
 20   0  447M 11036  8452 S  0.0  0.1  0:00.00 │  ├─ nix-daemon --daemon
 20   0  447M 11036  8452 S  0.0  0.1  0:00.00 │  ├─ nix-daemon --daemon
 20   0  447M 11036  8452 S  0.0  0.1  0:00.00 │  └─ nix-daemon --daemon
 20   0  583M 47608 21852 S  4.0  0.3  3:05.70 ├─ /nix/store/nj4ny40grm5x37s2qfs5zsqdpqfpk0af-gnome-terminal-3.32.2/libexec/gnome-terminal-
 20   0  144M 20884  2372 S  0.0  0.1  0:00.11 │  ├─ bash
 20   0  144M 20888  2432 S  0.0  0.1  0:00.11 │  ├─ bash
 20   0  129M  4224  2020 R  1.3  0.0  0:01.61 │  │  └─ htop
 20   0  144M 21076  2500 S  0.0  0.1  0:00.16 │  ├─ bash
 20   0  135M  9388  3592 S  0.0  0.1  0:04.81 │  │  └─ vi
 20   0  144M 20860  2336 S  0.0  0.1  0:00.11 │  ├─ bash
 20   0  126M  2236  1936 S  0.0  0.0  0:00.00 │  │  └─ sudo su
 20   0  126M  1684  1432 S  0.0  0.0  0:00.00 │  │     └─ su
 20   0  128M  4212  2392 S  0.0  0.0  0:00.11 │  │        └─ bash
 20   0  129M  1240   936 S  0.0  0.0  0:00.02 │  │           └─ cat trace_pipe
```



- The relevant events are split into two categories. Processes and files.

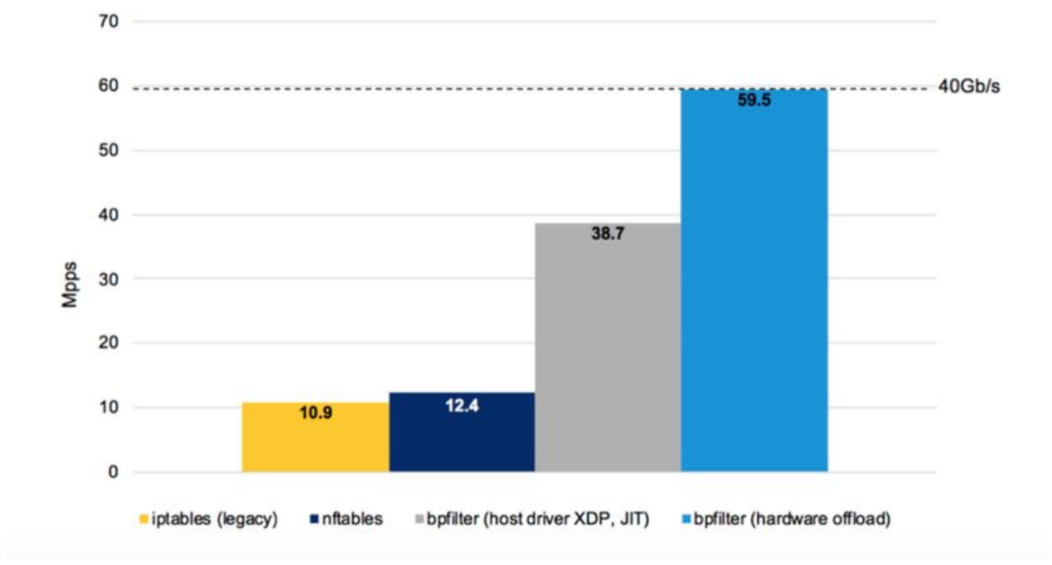
## First attempts

- strace
- fptrace [1]

[1] https://github.com/orivej/fptrace

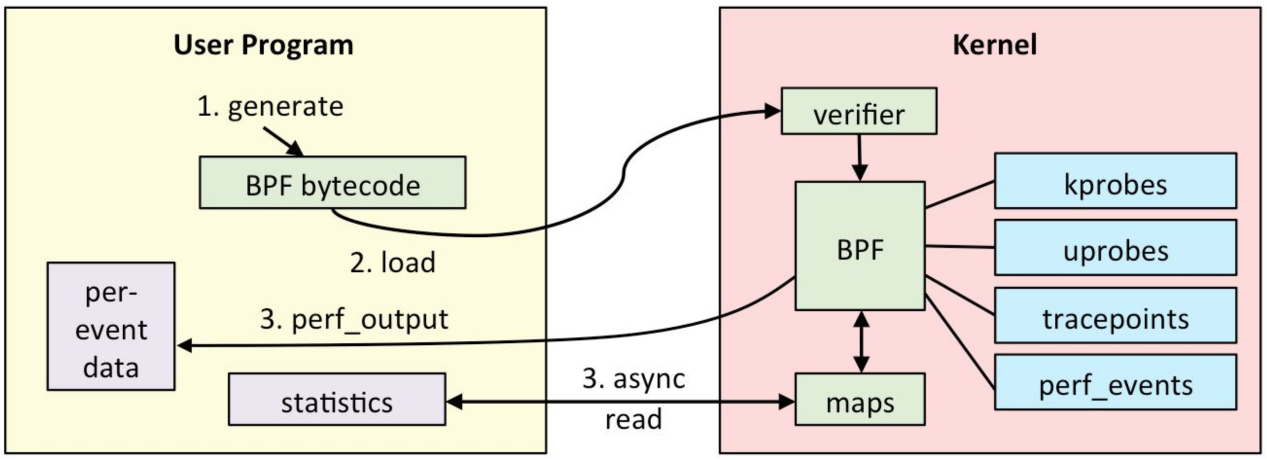
- I first attempted to use strace, which was very hard to parse, and quite slow
- The fptrace program kinda works, but builds up all information in memory, and crashed before the build was complete.
- Therefore, I thought about building my own version.

# eBPF - extended Berkeley Packet Filter



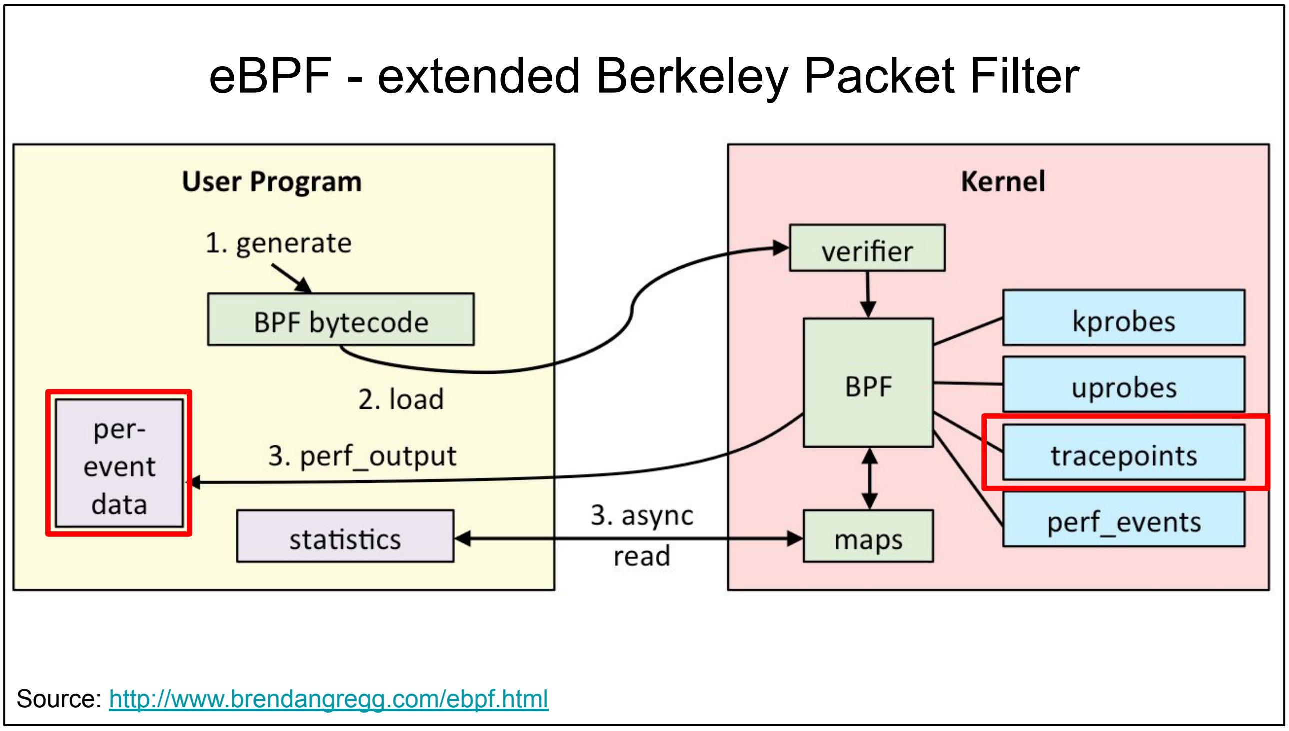
- Had heard about how BPF could help.
- BPF stands for Berkeley Packet Filter, and is a minimal in-kernel VM with it's own instruction set.
- It was build to be able to implement complex packet filtering rules
- Because it contains a JIT compiler, it is quite performant
- It even is possible to offload these instructions to specialized hardware, gaining even a higher speedup.
- This diagram is here to illustrate that bpf is performant.

## eBPF - extended Berkeley Packet Filter



Source: http://www.brendangregg.com/ebpf.html

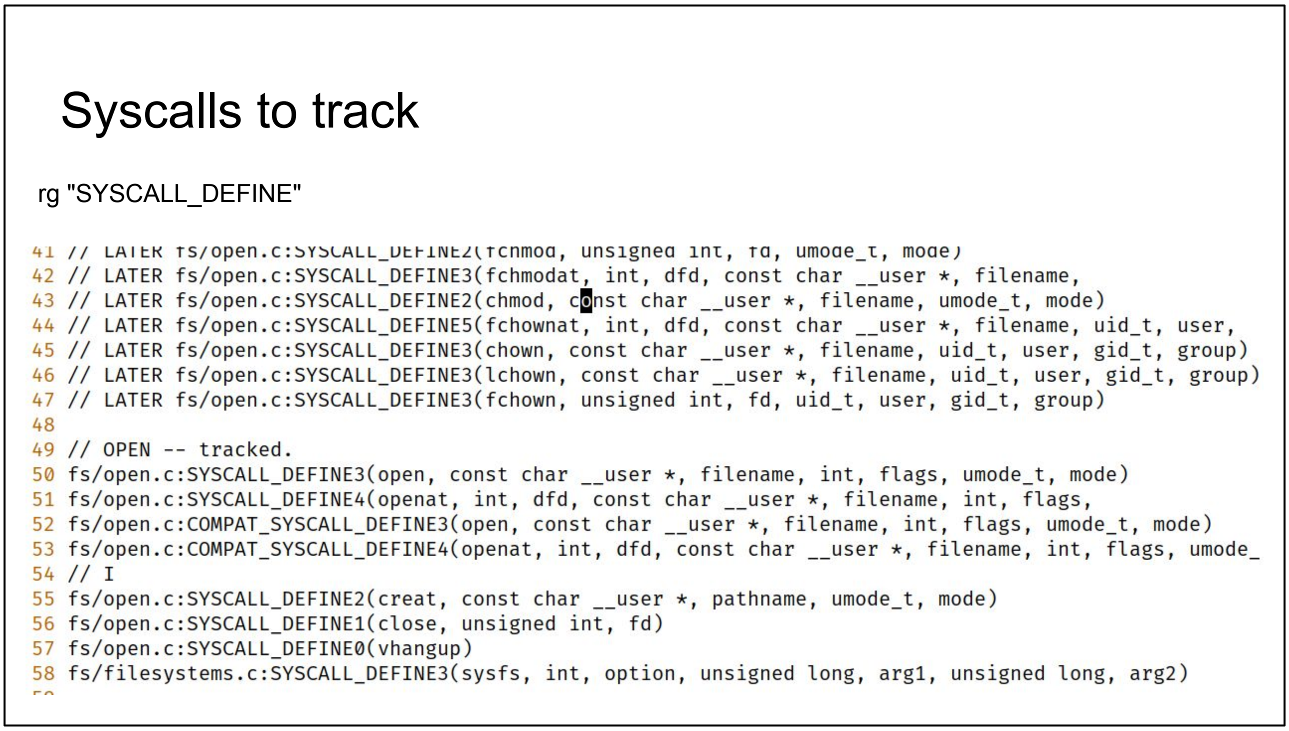
- However, we are not interested in filtereing packets.
- BPF can also be used to add instrumentation to your kernel.



## eBPF - extended Berkeley Packet Filter

Source: http://www.brendangregg.com/ebpf.html

- In particular, in our use case, we will be using tracepoints to link a probe into the relevant system calls
- And will emit data to userland via perf_output.
- Before we start, we need to know which system calls to track.



## Syscalls to track

rg "SYSCALL_DEFINE"

```
// LATER fs/open.c:SYSCALL_DEFINE2(fchmod, unsigned int, fd, umode_t, mode)
// LATER fs/open.c:SYSCALL_DEFINE3(fchmodat, int, dfd, const char __user *, filename,
// LATER fs/open.c:SYSCALL_DEFINE2(chmod, const char __user *, filename, umode_t, mode)
// LATER fs/open.c:SYSCALL_DEFINE5(fchownat, int, dfd, const char __user *, filename, uid_t, user,
// LATER fs/open.c:SYSCALL_DEFINE3(chown, const char __user *, filename, uid_t, user, gid_t, group)
// LATER fs/open.c:SYSCALL_DEFINE3(lchown, const char __user *, filename, uid_t, user, gid_t, group)
// LATER fs/open.c:SYSCALL_DEFINE3(fchown, unsigned int, fd, uid_t, user, gid_t, group)

// OPEN -- tracked.
fs/open.c:SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
fs/open.c:SYSCALL_DEFINE4(openat, int, dfd, const char __user *, filename, int, flags,
fs/open.c:COMPAT_SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
fs/open.c:COMPAT_SYSCALL_DEFINE4(openat, int, dfd, const char __user *, filename, int, flags, umode_
// I
fs/open.c:SYSCALL_DEFINE2(creat, const char __user *, pathname, umode_t, mode)
fs/open.c:SYSCALL_DEFINE1(close, unsigned int, fd)
fs/open.c:SYSCALL_DEFINE0(vhangup)
fs/filesystems.c:SYSCALL_DEFINE3(sysfs, int, option, unsigned long, arg1, unsigned long, arg2)
```

- To find the relevant system calls, I searched through the kernel source for SYSCALL_DEFINE macro’s.
- I think that I need to track about 20-50 syscalls.
- Tricky open_at($dirfd) calls
- We also need to keep track of process state. CWD / env vars are inherited in a child process from it’s parent. Just like FD’s are inherited.

## Example: trace execve syscall

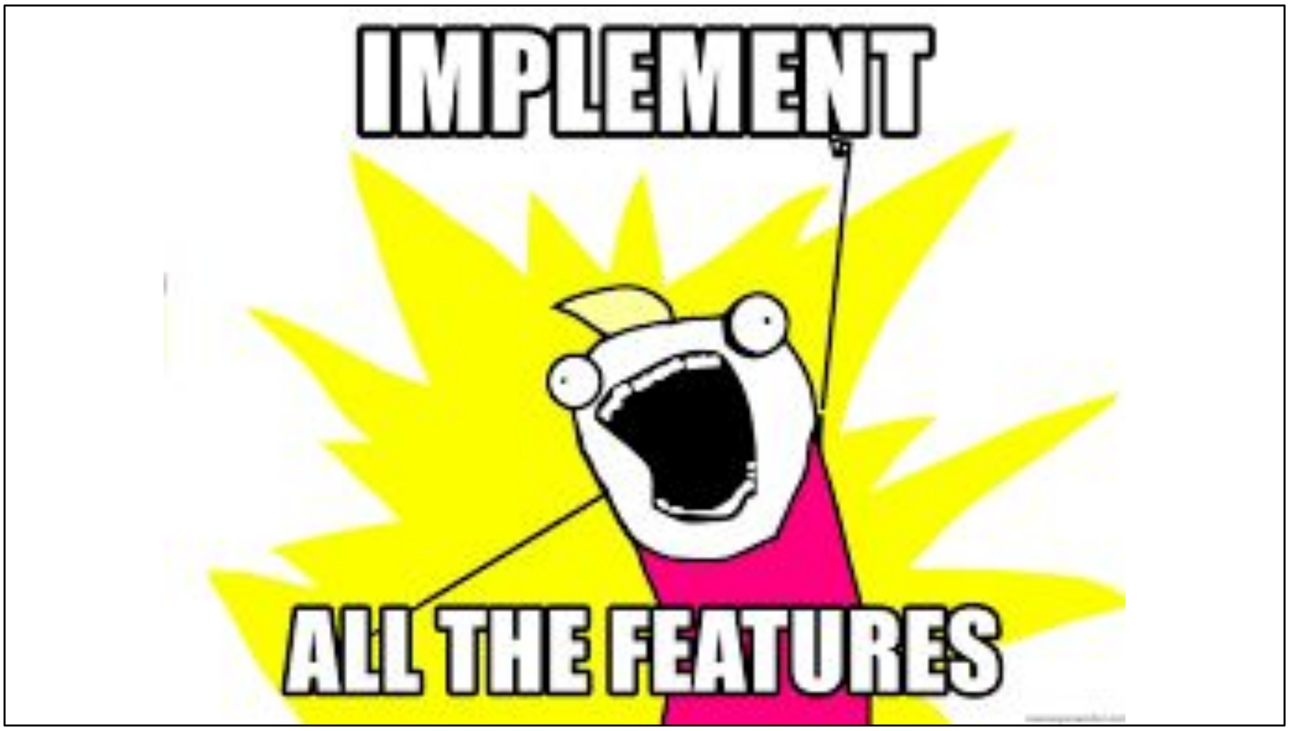
```
[root@maarten-yoga:/sys/kernel/debug/tracing/events/syscalls/sys_enter_execve]# cat format
name: sys_enter_execve
ID: 688
Format:
	field:unsigned short common_type;	offset:0;	size:2;	signed:0;
	field:unsigned char common_flags;	offset:2;	size:1;	signed:0;
	field:unsigned char common_preempt_count;	offset:3;	size:1;	signed:0;
	field:int common_pid;	offset:4;	size:4;	signed:1;
	field:int __syscall_nr;	offset:8;	size:4;	signed:1;
	field:const char * filename;	offset:16;	size:8;	signed:0;
	field:const char *const * argv;	offset:24;	size:8;	signed:0;
	field:const char *const * envp;	offset:32;	size:8;	signed:0;


print fmt: "filename: 0x%08lx, argv: 0x%08lx, envp: 0x%08lx", ((unsigned long)(REC->filename)),
((unsigned long)(REC->argv)), ((unsigned long)(REC->envp))
```

- For each syscall, we can inspect the data that is made available to the probe.

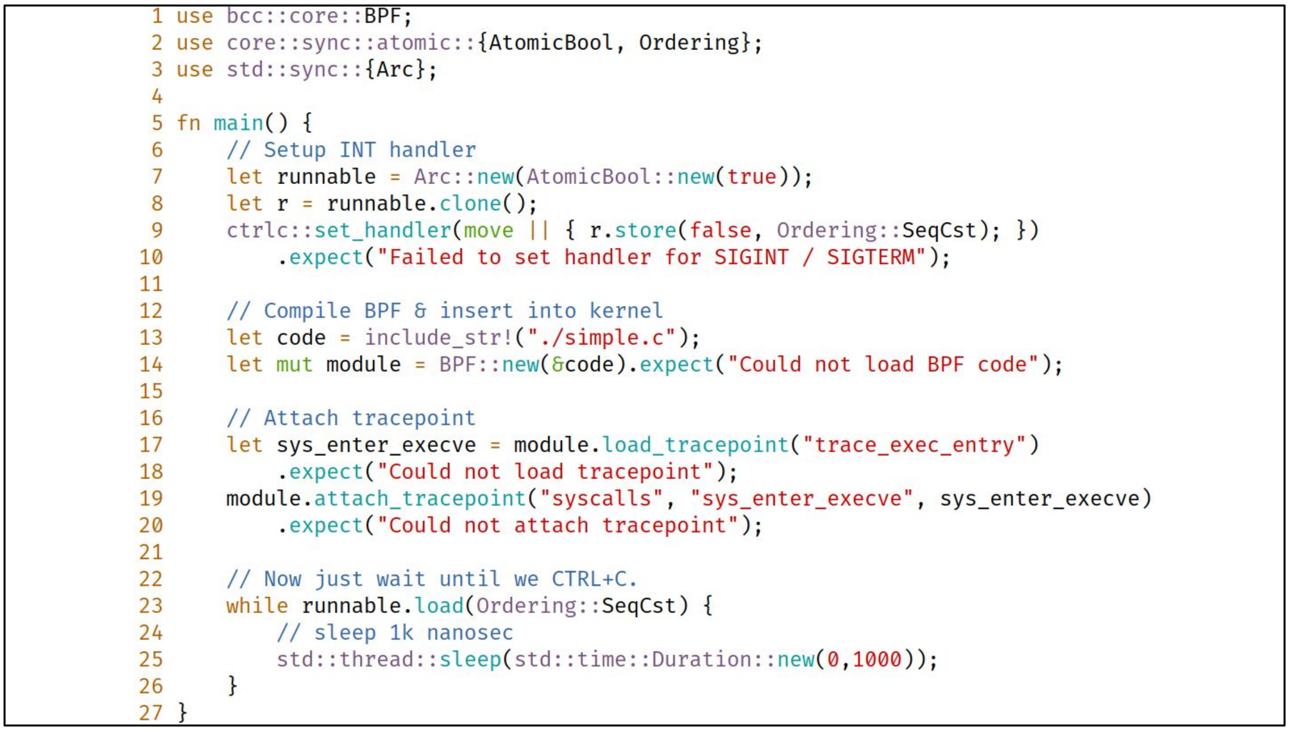
## Example: trace execve syscall

```
[root@maarten-yoga:/sys/kernel/debug/tracing/events/syscalls/sys_enter_execve]# cat format
name: sys_enter_execve
ID: 688
Format:
	field:unsigned short common_type;	offset:0;	size:2;	signed:0;
	field:unsigned char common_flags;	offset:2;	size:1;	signed:0;
	field:unsigned char common_preempt_count;	offset:3;	size:1;	signed:0;
	field:int common_pid;	offset:4;	size:4;	signed:1;
	field:int __syscall_nr;	offset:8;	size:4;	signed:1;
	field:const char * filename;	offset:16;	size:8;	signed:0;
	field:const char *const * argv;	offset:24;	size:8;	signed:0;
	field:const char *const * envp;	offset:32;	size:8;	signed:0;


print fmt: "filename: 0x%08lx, argv: 0x%08lx, envp: 0x%08lx", ((unsigned long)(REC->filename)),
((unsigned long)(REC->argv)), ((unsigned long)(REC->envp))
```

- For the enter_execve tracepoint, we can indeed see the relevant arguments.



- Then we should simply implement the capture of that.
- Simply do that? But....
    - How to capture only info from the relevant processes?
    - How to get this into a dump file?
    - How to actually WRITE this stuff?

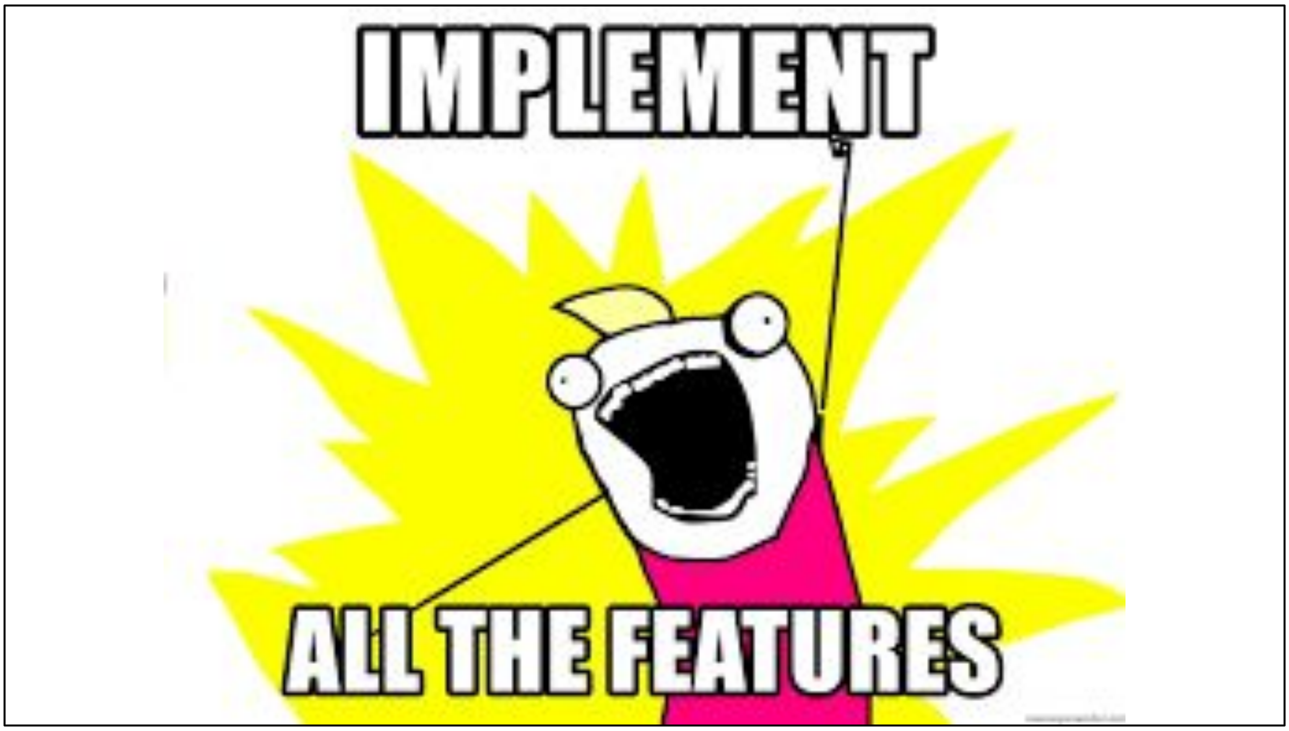


```rust
 1 use bcc::core::BPF;
 2 use core::sync::atomic::{AtomicBool, Ordering};
 3 use std::sync::{Arc};
 4
 5 fn main() {
 6     // Setup INT handler
 7     let runnable = Arc::new(AtomicBool::new(true));
 8     let r = runnable.clone();
 9     ctrlc::set_handler(move || { r.store(false, Ordering::SeqCst); })
10         .expect("Failed to set handler for SIGINT / SIGTERM");
11
12     // Compile BPF & insert into kernel
13     let code = include_str!("./simple.c");
14     let mut module = BPF::new(&code).expect("Could not load BPF code");
15
16     // Attach tracepoint
17     let sys_enter_execve = module.load_tracepoint("trace_exec_entry")
18         .expect("Could not load tracepoint");
19     module.attach_tracepoint("syscalls", "sys_enter_execve", sys_enter_execve)
20         .expect("Could not attach tracepoint");
21
22     // Now just wait until we CTRL+C.
23     while runnable.load(Ordering::SeqCst) {
24         // sleep 1k nanosec
25         std::thread::sleep(std::time::Duration::new(0,1000));
26     }
27 }
```

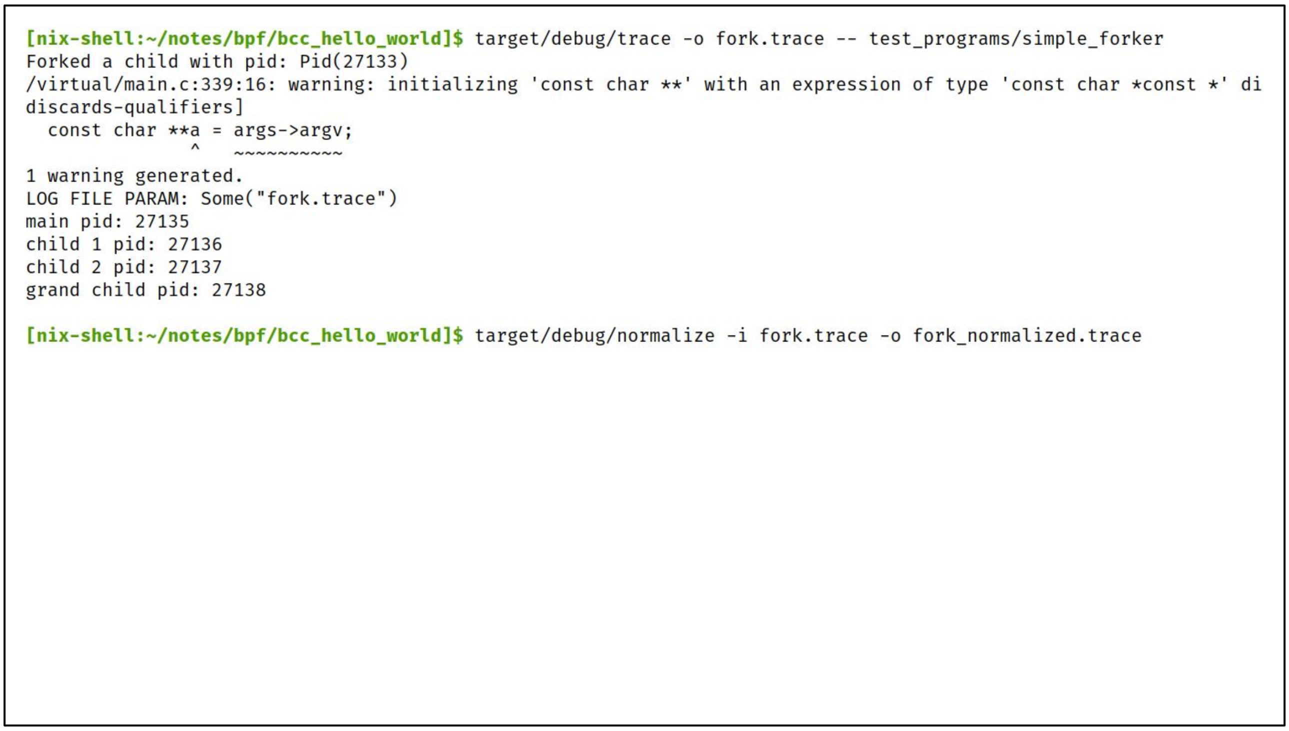
- Luckily, there is a simple rust crate called bcc  (inspired by python's bcc library)
- This loads a piece of C code (line 13), runs a special compiler during runtime, and then links the defined probe to the sys_enter_execve tracepoint.
-

```c
int trace_exec_entry(struct tracepoint__syscalls__sys_enter_execve *args) {
  u64 id = bpf_get_current_pid_tgid();
  u32 pid = id >> 32; // PID is higher part

  bpf_trace_printk("[%ld] EXEC %s\n", pid, args->filename);

  #pragma unroll
  for (int i = 0; i < 10; i++) {
    const char *arg;
    bpf_probe_read(&arg, sizeof(char *), &args->argv[i]);
    if (arg == 0)
      break;
    bpf_trace_printk("[%ld] EXEC ARG %d %s\n", pid, i, arg);
  }

  #pragma unroll
  for (int i = 0; i < 10; i++) {
    const char *env;
    bpf_probe_read(&env, sizeof(char *), &args->envp[i]);
    if (env == 0)
      break;
    bpf_trace_printk("[%ld] EXEC ENV %d %s\n", pid, i, env);
  }

  return 0;
}
```

- This is the c code that is executed in the kernel.
- The compiler automatically generates a struct with pointers to the arguments.
- This program prints the name of the program that is executed, and the first 10 arguments of the arguments,, and the first 10 env vars of the process that we want to execute.

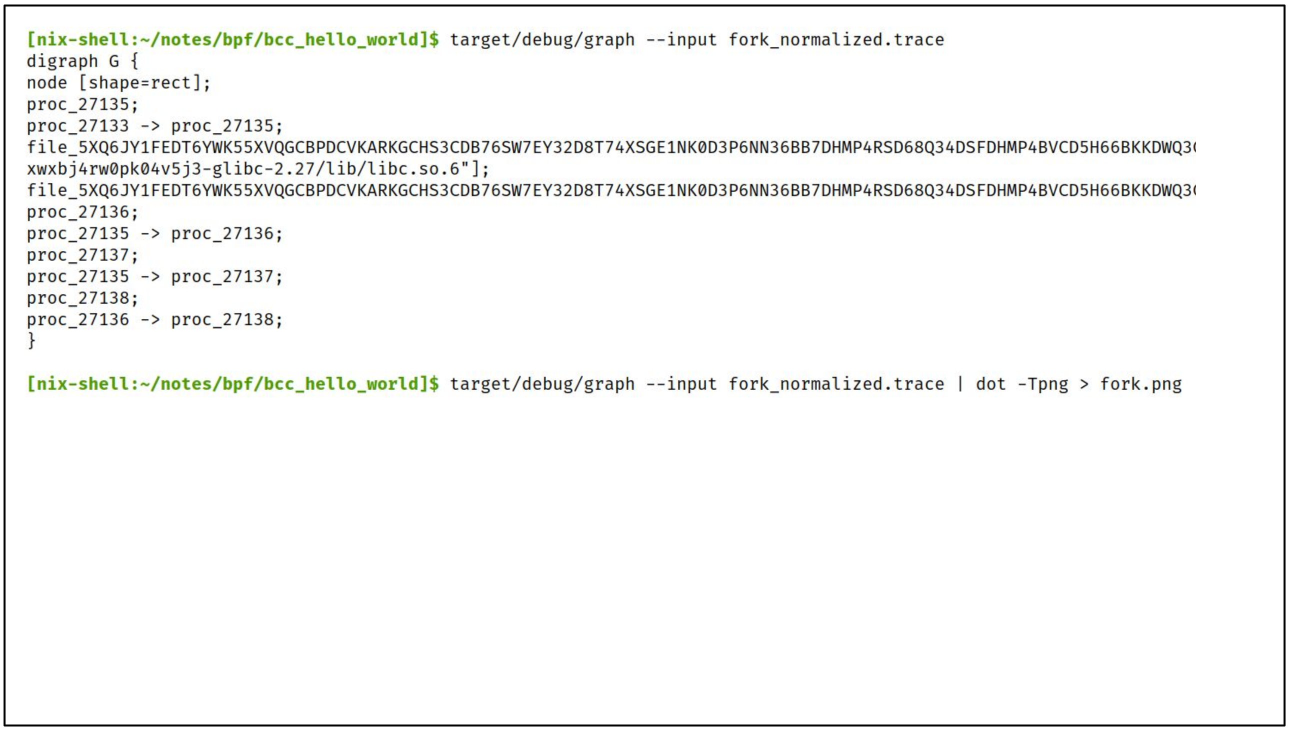


- Looks simple!
- This is when I sent out the original invite promising a analysis tool as well.
- Well... things are a actually a bit complicated.
- I'll show you what I got working, and then we'll talk about problems.



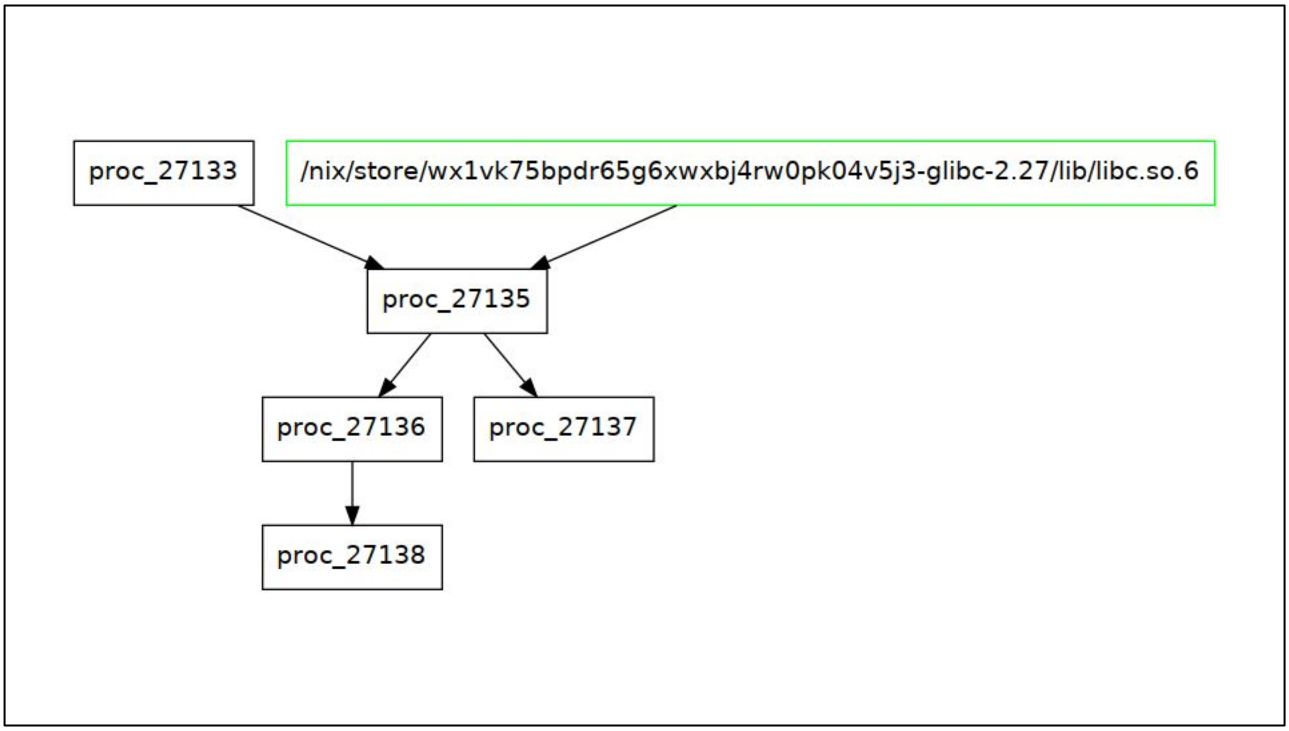
```
[nix-shell:~/notes/bpf/bcc_hello_world]$ target/debug/trace -o fork.trace -- test_programs/simple_forker
Forked a child with pid: Pid(27133)
/virtual/main.c:339:16: warning: initializing 'const char **' with an expression of type 'const char *const *' di
discards-qualifiers]
  const char **a = args->argv;
               ^   ~~~~~~~~~~
1 warning generated.
LOG FILE PARAM: Some("fork.trace")
main pid: 27135
child 1 pid: 27136
child 2 pid: 27137
grand child pid: 27138

[nix-shell:~/notes/bpf/bcc_hello_world]$ target/debug/normalize -i fork.trace -o fork_normalized.trace
```

- Here I run the tracer on a simple test program, which forks two processes. On of the children forks a grand child.
- The trace program writes a log of events to the fork.trace file as they are received from the kernel. This will ensure that the impact of the capture is minimal
- Then I ran a normalization step, which mainly orders the events by time.

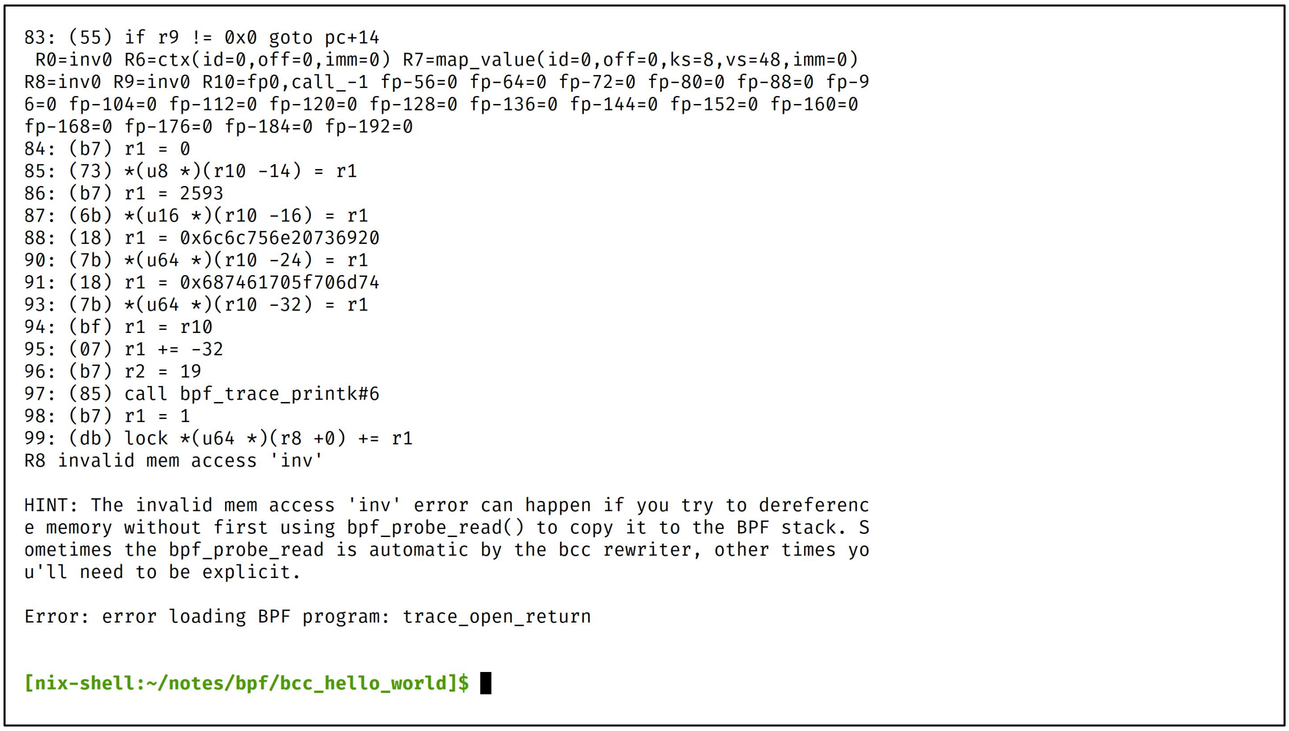


```
[nix-shell:~/notes/bpf/bcc_hello_world]$ target/debug/graph --input fork_normalized.trace
digraph G {
node [shape=rect];
proc_27135;
proc_27133 -> proc_27135;
file_5XQ6JY1FEDT6YWK55XVQGCBPDCVKARKGCHS3CDB76SW7EY32D8T74XSGE1NK0D3P6NN36BB7DHMP4RSD68Q34DSFDHMP4BVCD5H66BKKDWQ3(
xwxbj4rw0pk04v5j3-glibc-2.27/lib/libc.so.6"];
file_5XQ6JY1FEDT6YWK55XVQGCBPDCVKARKGCHS3CDB76SW7EY32D8T74XSGE1NK0D3P6NN36BB7DHMP4RSD68Q34DSFDHMP4BVCD5H66BKKDWQ3(
proc_27136;
proc_27135 -> proc_27136;
proc_27137;
proc_27135 -> proc_27137;
proc_27138;
proc_27136 -> proc_27138;
}

[nix-shell:~/notes/bpf/bcc_hello_world]$ target/debug/graph --input fork_normalized.trace | dot -Tpng > fork.png
```

- I also wrote a small tool to generate graphiz output


proc_27133
/nix/store/wx1vk75bpdr65g6xwxbj4rw0pk04v5j3-glibc-2.27/lib/libc.so.6
proc_27135
proc_27136
proc_27137
proc_27138

- Which can be seen here.
- (explain shell process in Rust, libc.so loading, the two children)
- This looks simple, but is quite complicated
- It filters on child processes, to not dump all system activity
- And writing eBPF programs is quite tricky...

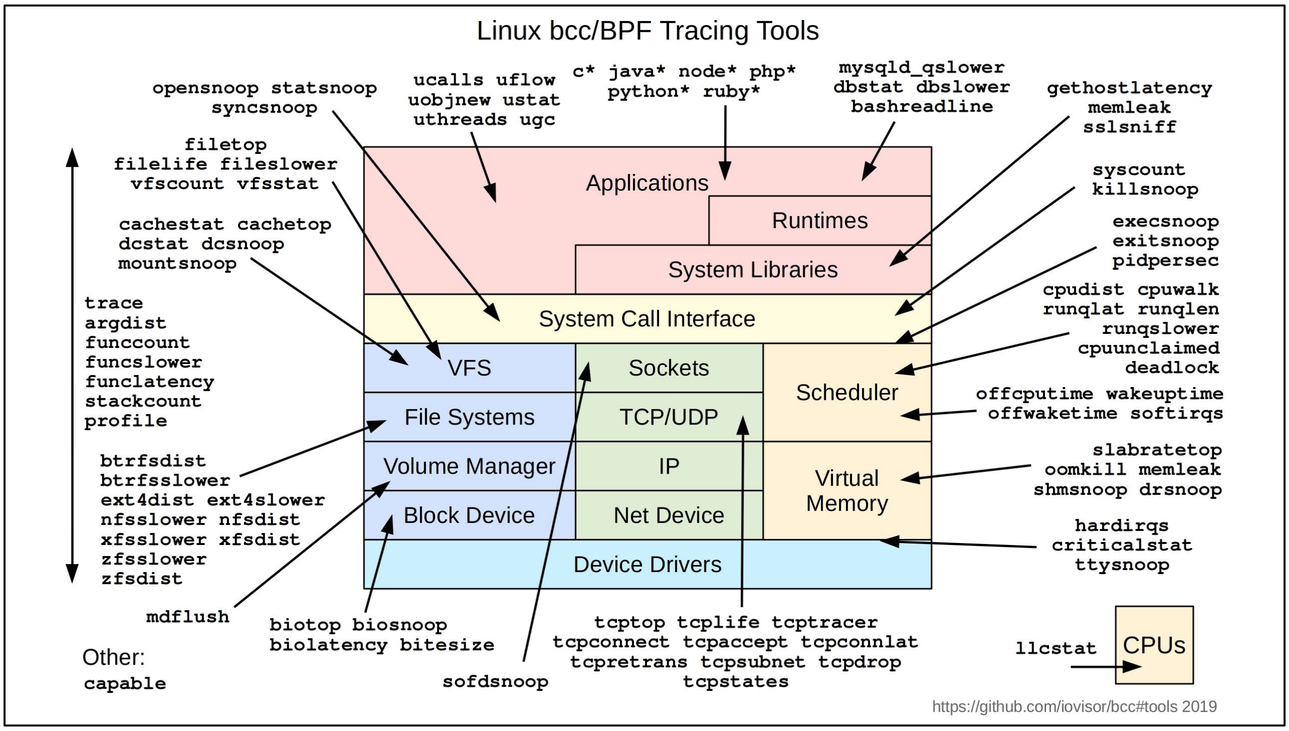


```
83: (55) if r9 != 0x0 goto pc+14
 R0=inv0 R6=ctx(id=0,off=0,imm=0) R7=map_value(id=0,off=0,ks=8,vs=48,imm=0)
R8=inv0 R9=inv0 R10=fp0,call_-1 fp-56=0 fp-64=0 fp-72=0 fp-80=0 fp-88=0 fp-9
6=0 fp-104=0 fp-112=0 fp-120=0 fp-128=0 fp-136=0 fp-144=0 fp-152=0 fp-160=0
fp-168=0 fp-176=0 fp-184=0 fp-192=0
84: (b7) r1 = 0
85: (73) *(u8 *)(r10 -14) = r1
86: (b7) r1 = 2593
87: (6b) *(u16 *)(r10 -16) = r1
88: (18) r1 = 0x6c6c756e20736920
90: (7b) *(u64 *)(r10 -24) = r1
91: (18) r1 = 0x687461705f706d74
93: (7b) *(u64 *)(r10 -32) = r1
94: (bf) r1 = r10
95: (07) r1 += -32
96: (b7) r2 = 19
97: (85) call bpf_trace_printk#6
98: (b7) r1 = 1
99: (db) lock *(u64 *)(r8 +0) += r1
R8 invalid mem access 'inv'

HINT: The invalid mem access 'inv' error can happen if you try to dereferenc
e memory without first using bpf_probe_read() to copy it to the BPF stack. S
ometimes the bpf_probe_read is automatic by the bcc rewriter, other times yo
u'll need to be explicit.

Error: error loading BPF program: trace_open_return


[nix-shell:~/notes/bpf/bcc_hello_world]$ 
```

- Boom! This is an example error that I got.
- You need to be very careful about the eBPF handlers you write.
- If the verifier disagrees with you, you lose.
- And as a bonus, you just get the VM instructions spewed back at you, which you need to correlate to your C code manually.

- Some rough edges I learned about:
  - Max number of instructions is 4096 in a single function.
    - Means I can only parse 29 arguments in exec handler :( Not including env.
    - Might solve this with tail calling another program though!
- Understanding the error messages is hard. INV!? Apparenlty that means that a register contains a value that cannot be proven to be in a certain state.
- Googling. Lots of googling. And experimtentation.
- I also learned that I shouldn't use kprobes. Those are unstable. Tracepoints are *meant* to be more stable.

## Extra



Linux bcc/BPF Tracing Tools
opensnoop statsnoop syncsnoop
ucalls uflow uobjnew ustat uthreads ugc
c* java* node* php* python* ruby*
mysqld_qslower dbstat dbslower bashreadline
gethostlatency memleak sslsniff
filetop filelife fileslower vfscount vfsstat
cachestat cachetop dcstat dcsnoop mountsnoop
syscount killsnoop
execsnoop exitsnoop pidpersec
cpudist cpuwalk runqlat runqlen runqslower cpuunclaimed deadlock
offcputime wakeuptime offwaketime softirqs
slabratetop oomkill memleak shmsnoop drsnoop
hardirqs criticalstat ttysnoop
trace argdist funccount funcslower funclatency stackcount profile
btrfsdist btrfsslower ext4dist ext4slower nfsslower nfsdist xfsslower xfsdist zfsslower zfsdist
mdflush
biotop biosnoop biolatency bitesize
sofdsnoop
tcptop tcplife tcptracer tcpconnect tcpaccept tcpconnlat tcpretrans tcpsubnet tcpdrop tcpstates
llcstat
Applications
Runtimes
System Libraries
System Call Interface
VFS
Sockets
Scheduler
File Systems
TCP/UDP
Volume Manager
IP
Virtual Memory
Block Device
Net Device
Device Drivers
CPUs
Other:
capable
https://github.com/iovisor/bcc#tools 2019