# Security Monitoring with eBPF

ALEX MAESTRETTI - MANAGER, SIRT
BRENDAN GREGG - Sr ARCHITECT, PERFORMANCE

NETFLIX

# The Brief.

Extended Berkley Packet Filter (eBPF) is a new Linux feature which allows safe and efficient monitoring of kernel functions. This has dramatic implications for security monitoring, especially at Netflix scale. We are encouraging the security community to leverage this new technology to all of our benefit.

# Existing Solutions.

There are many security monitoring solutions available today that meet a wide range of requirements. Our design goals were: push vs poll, lightweight, with kernel-level inspection. Our environment is composed of micro-services running on ephemeral and immutable instances built and deployed from source control into a public cloud.

osquery

ossec

sysdig

auditd

# A new Option.

```
# capable
TIME      UID    PID    COMM             CAP  NAME                 AUDIT
22:11:23  114    2676   snmpd            12   CAP_NET_ADMIN        1
22:11:23  0      6990   run              24   CAP_SYS_RESOURCE     1
22:11:23  0      7003   chmod            3    CAP_FOWNER           1
22:11:23  0      7003   chmod            4    CAP_FSETID           1
22:11:23  0      7005   chmod            4    CAP_FSETID           1
22:11:23  0      7005   chmod            4    CAP_FSETID           1
22:11:23  0      7006   chown            4    CAP_FSETID           1
22:11:23  0      7006   chown            4    CAP_FSETID           1
22:11:23  0      6990   setuidgid        6    CAP_SETGID           1
22:11:23  0      6990   setuidgid        6    CAP_SETGID           1
22:11:23  0      6990   setuidgid        7    CAP_SETUID           1
22:11:24  0      7013   run              24   CAP_SYS_RESOURCE     1
22:11:24  0      7026   chmod            3    CAP_FOWNER           1
22:11:24  0      7026   chmod            4    CAP_FSETID           1
[...]
```

Snooping on Linux cap_capable() calls using bcc/eBPF

# SCREENSHOT 2

```
# argdist -i 5 -C 'p::cap_capable():int:ctx->dx'
[06:32:08]
p::cap_capable():int:ctx->dx
     COUNT        EVENT
     2            ctx->dx = 35
     5            ctx->dx = 21
     83           ctx->dx = 12
[06:32:13]
p::cap_capable():int:ctx->dx
     COUNT        EVENT
     1            ctx->dx = 1
     7            ctx->dx = 21
     82           ctx->dx = 12
[...]
```

Now frequency counting in-kernel
and only sending the summary to user
eBPF is much more than just a per-event tracer
(this is a bcc/eBPF hack; I should make this into a real tool like the previous one)

- 2004: kprobes (2.6.9)
- 2005: DTrace (not Linux); SystemTap (out-of-tree)
- 2008: ftrace (2.6.27)
- 2009: perf_events (2.6.31)
- 2009: tracepoints (2.6.32)
- 2010-2016: ftrace & perf_events enhancements
- 2012: uprobes (3.5)
- 2014-2016: Enhanced BPF patches

+ other out of tree tracers
LTTng, ktap, sysdig, ...

# KERNEL INSTRUMENTATION USING KPROBES
## PHRACK ZINE #67/6 2010-11-17


1 - Introduction
    1.1 - Why write it?
    1.2 - About kprobes
    1.3 - Jprobe example
    1.4 - Kretprobe example & Retu
2 - Kprobes implementation
    2.1 - Kprobe implementation
    2.2 - Jprobe implementation
    2.3 - File hiding with jprobes
    2.4 - Kretprobe implementation
    2.5 - A quick stop into modify
    2.6 - An idea for a kretprobe
3 - Patch to unpatch W^X (mprotect
4 - Notes on rootkit detection for
5 - Summing it all up.
6 - Greetz
7 - References and citations
8 - Code


"So why write this? Because... we are hackers. Hackers should be aware of any and all resources available to them -- some more auspicious than others -- Nonetheless, kprobes are a sweet deal when you consider that they are a native kernel API…"

http://phrack.org/issues/67/6.html
(also see http://phrack.org/issues/63/3.html)

```
# tcpdump host 127.0.0.1 and port 22 -d
(000) ldh       [12]
(001) jeq       #0x800          jt 2    jf 18
(002) ld        [26]
(003) jeq       #0x7f000001     jt 6    jf 4
(004) ld        [30]
(005) jeq       #0x7f000001     jt 6    jf 18
(006) ldb       [23]
(007) jeq       #0x84           jt 10   jf 8
(008) jeq       #0x6            jt 10   jf 9
(009) jeq       #0x11           jt 10   jf 18
(010) ldh       [20]
(011) jset      #0x1fff         jt 18   jf 12
(012) ldxb      4*([14]&0xf)
[...]
```

**2 x 32-bit registers & scratch memory**

User-defined bytecode executed by an in-kernel sandboxed virtual machine

Steven McCanne and Van Jacobson, 1993

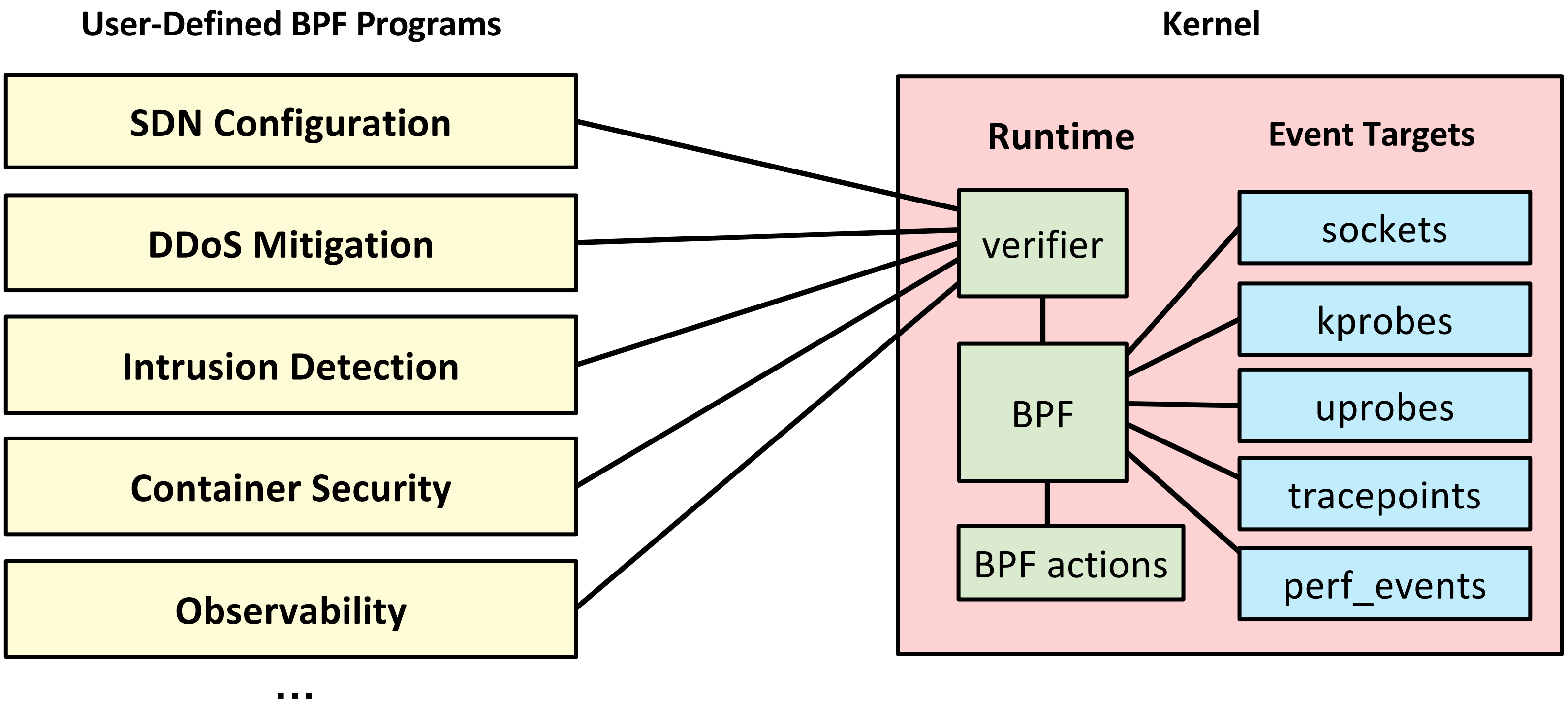
```
struct bpf_insn prog[] = {
        BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),
        BPF_LD_ABS(BPF_B, ETH_HLEN + offsetof(struct iphdr, protocol) /* R0 = ip->proto */),
        BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4), /* *(u32 *)(fp - 4) = r0 */
        BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
        BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4), /* r2 = fp - 4 */
        BPF_LD_MAP_FD(BPF_REG_1, map_fd),
        BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
        BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
        BPF_MOV64_IMM(BPF_REG_1, 1), /* r1 = 1 */
        BPF_RAW_INSN(BPF_STX | BPF_XADD | BPF_DW, BPF_REG_0, BPF_REG_1, 0, 0), /* xadd r0 += r1 */
        BPF_MOV64_IMM(BPF_REG_0, 0), /* r0 = 0 */
        BPF_EXIT_INSN(),
};
```

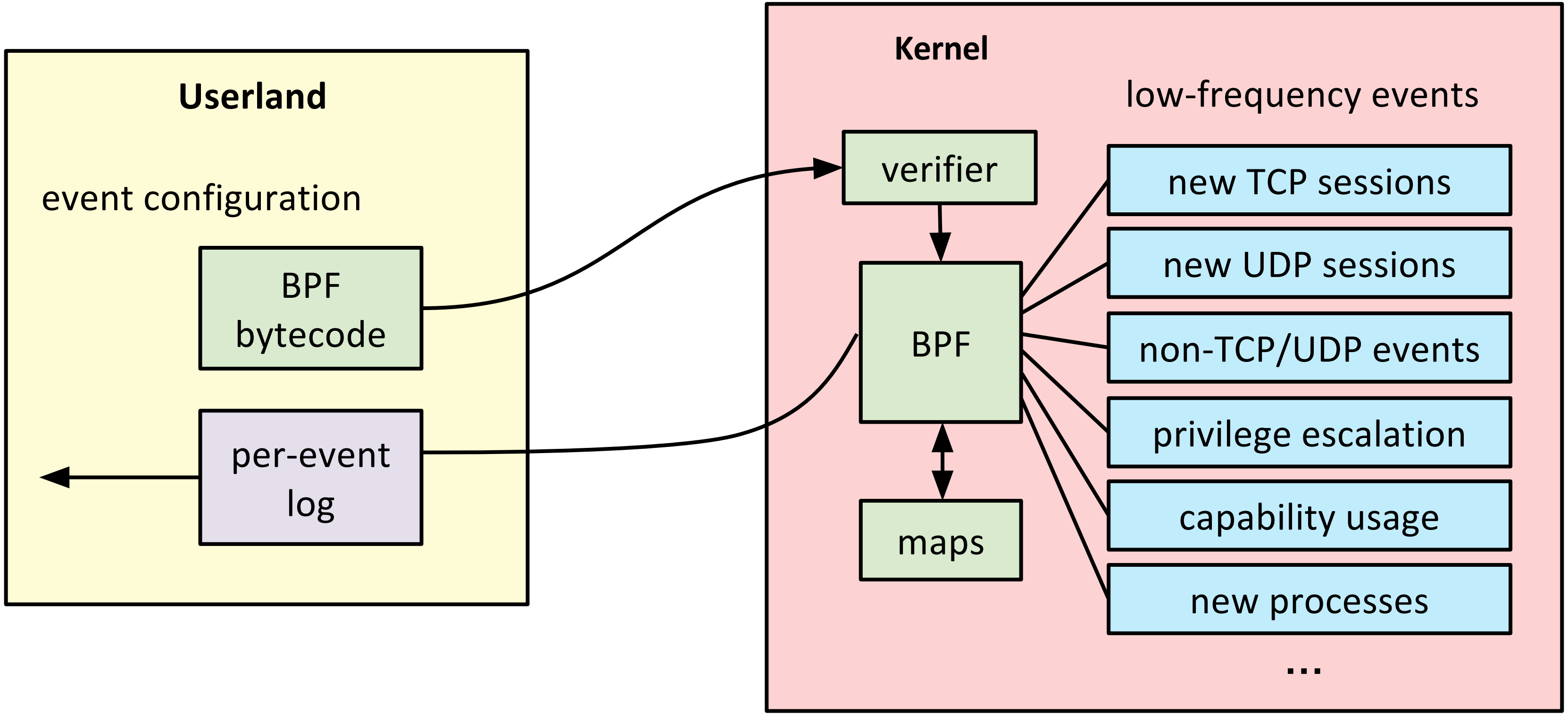
**10 x 64-bit registers**
**maps (hashes)**
**actions**

Alexei Starovoitov, 2015+

There are front-ends (eg, bcc) so we never have to write such raw eBPF

## eBPF USE CASES

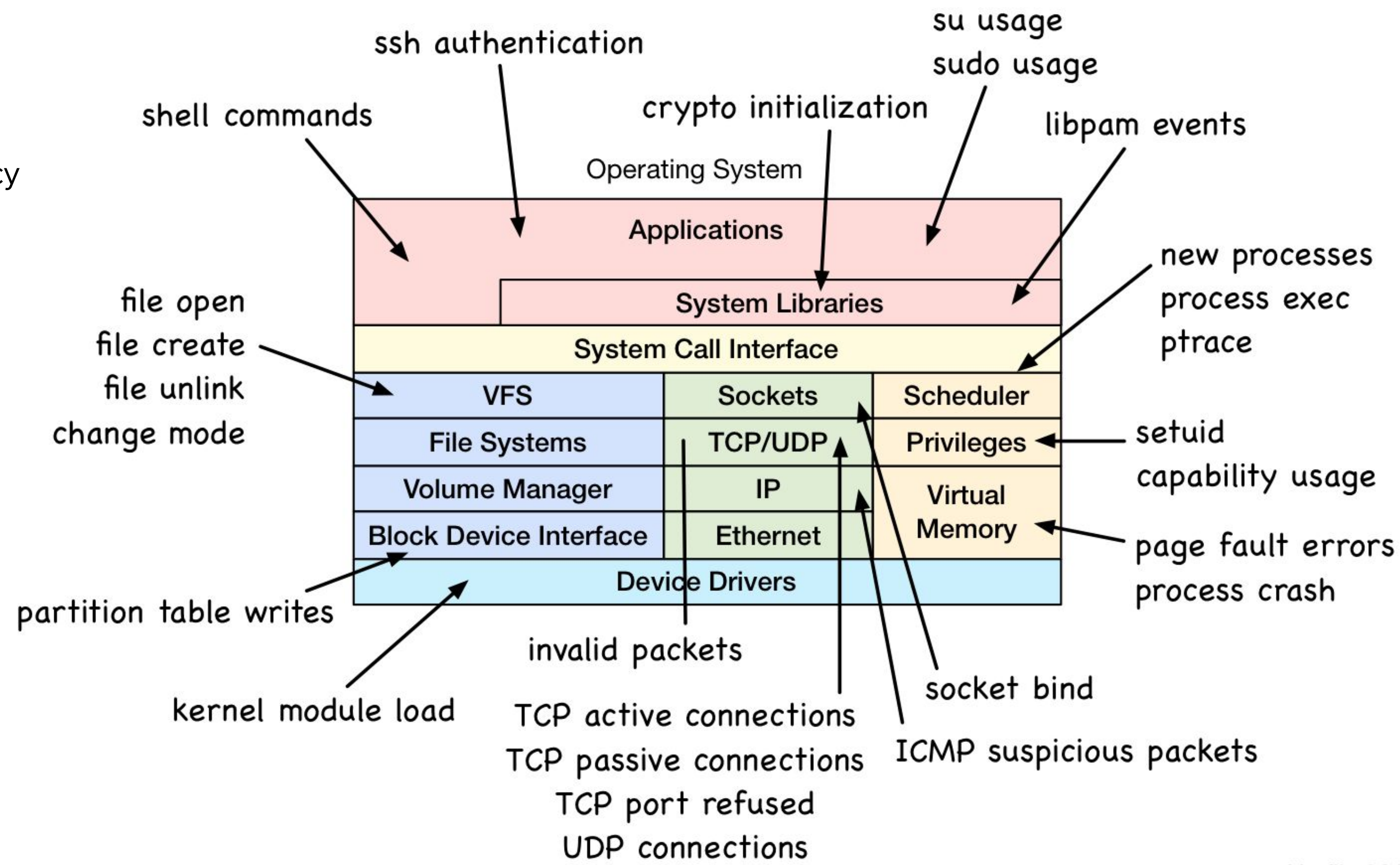



Userland
event configuration
BPF
bytecode
per-event
log
Kernel
low-frequency events
verifier
BPF
maps
new TCP sessions
new UDP sessions
non-TCP/UDP events
privilege escalation
capability usage
new processes
...

# WHAT TO MONITOR

Trace low-frequency events wherever possible to lower overhead

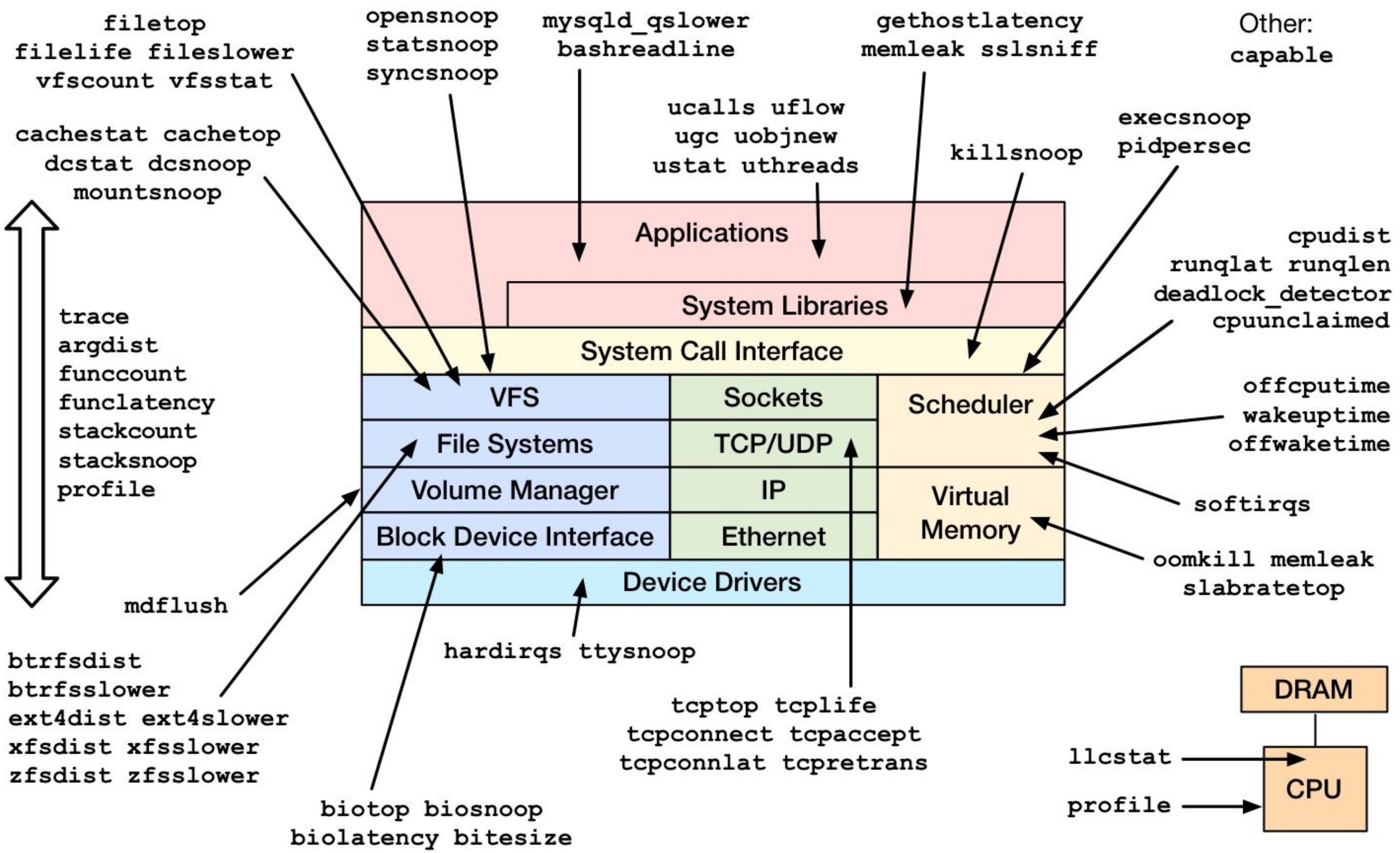
Eg, TCP connection init; not TCP send/receive




Netflix 2017

# BCC EXAMPLES

These bcc/BPF observability tools show what is possible




https://github.com/iovisor/bcc#tools 2017

From the bcc collection

```
# ./execsnoop -x
PCOMM            PID    RET ARGS
supervise        9661     0 ./run
mkdir            9662     0 /bin/mkdir -p ./main
run              9663     0 ./run
chown            9664     0 /bin/chown nobody:nobody ./main
run              9665     0 /bin/mkdir -p ./main
run              9660    -2 /usr/local/bin/setuidgid nobody
[...]

# ./tcpconnect -t
TIME(s)  PID    COMM         IP SADDR            DADDR            DPORT
31.871   2482   local_agent  4  10.103.219.236   10.251.148.38    7001
31.874   2482   local_agent  4  10.103.219.236   10.101.3.132     7001
31.878   2482   local_agent  4  10.103.219.236   10.171.133.98    7101
90.917   2482   local_agent  4  10.103.219.236   10.251.148.38    7001
90.928   2482   local_agent  4  10.103.219.236   10.102.64.230    7001
[...]
```

# Use the stable-ist API possible

In order of preference:

## Kernel events

a. Tracepoints: stable API, if available.
b. Kprobes: dynamic tracing of security hooks
c. Kprobes: dynamic tracing of kernel functions

## User events

d. User Statically Defined Tracing (USDT) probes: stable API, if available
e. Uprobes: dynamic tracing of API interface functions
f. Uprobes: dynamic tracing of internal functions

# Safe

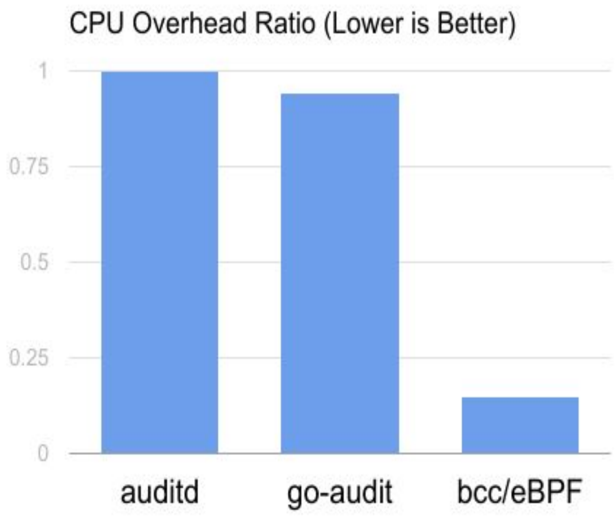
- Kernel verifies eBPF code (DAG and null reference check)
- Kernel memory access controlled through helper functions
- Part of the mainline kernel, no 3rd party kernel modules

# Flexible

- Add new instrumentation to production servers anytime
- Any event, any data

# Performant

- JIT'd instrumentation
- Data from kernel to user via async maps or per-events on a ring buffer
- Custom filters and summaries in kernel
- Can choose lower-frequency events to trace

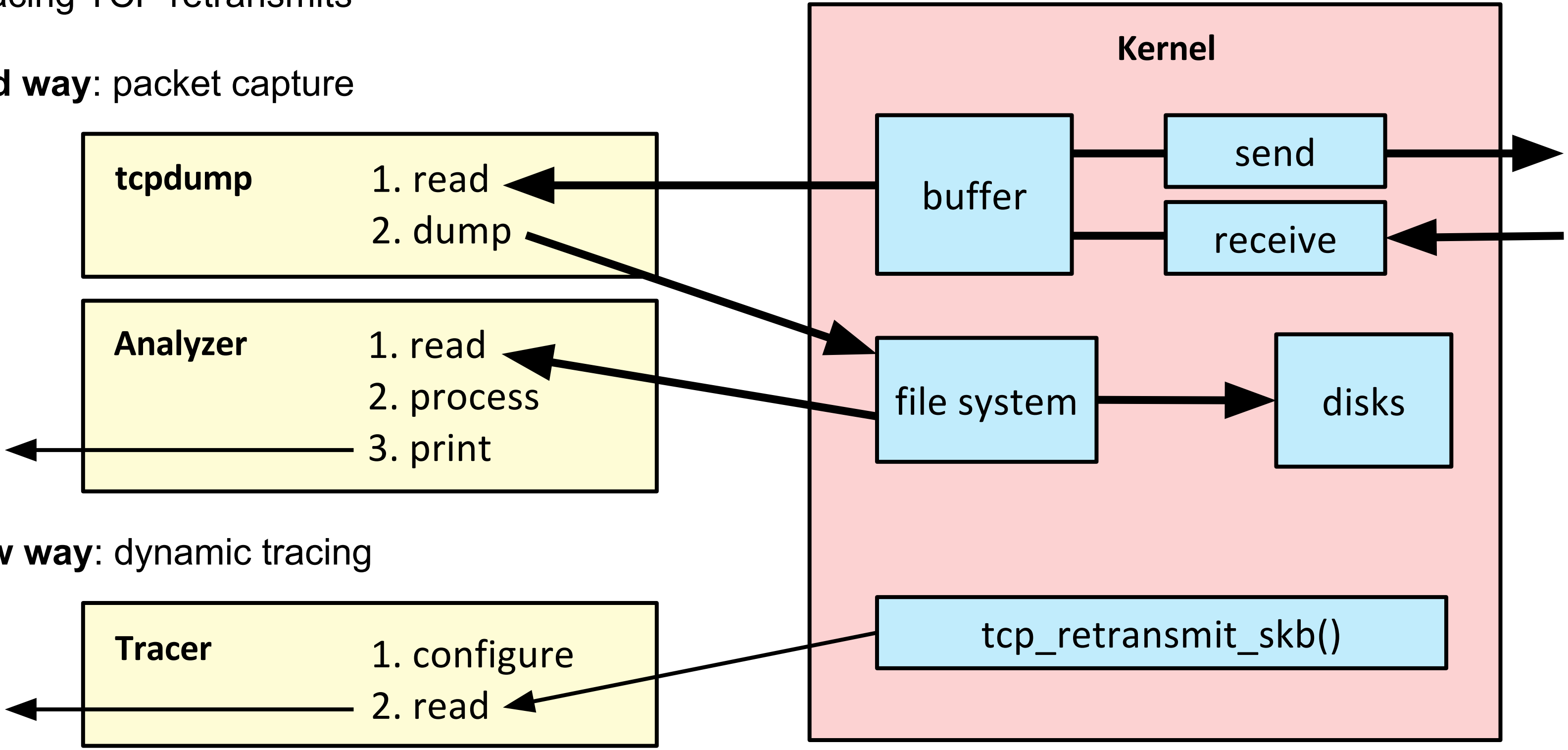



Preliminary results of logging TCP accept() to the file system, with a certain workload, and comparing overheads. Active benchmarking was performed. Each of these can likely be tuned further: results are not final.

Eg, tracing TCP retransmits

**Old way**: packet capture

**New way**: dynamic tracing



What is in a bcc eBPF Python file:

- Python code for userland reporting
- eBPF C code for event handling, in a variable (or file)
- BCC calls to initialize BPF and probes

**BPF Compiler Collection**

github.com/iovisor/bcc/

```
# load BPF program
b = BPF(text="""
#include <uapi/linux/ptrace.h>
#include <linux/blkdev.h>
BPF_HISTOGRAM(dist);
int kprobe__blk_account_io_completion(struct pt_regs *ctx,
    struct request *req)
{
    dist.increment(bpf_log2l(req->__data_len / 1024));
    return 0;
}
""")
```

bitehist.py example

```
# header
print("Tracing... Hit Ctrl-C to end.")

# trace until Ctrl-C
try:
    sleep(99999999)
except KeyboardInterrupt:
    print

# output
b["dist"].print_log2_hist("kbytes")
```

It gets more complicated...

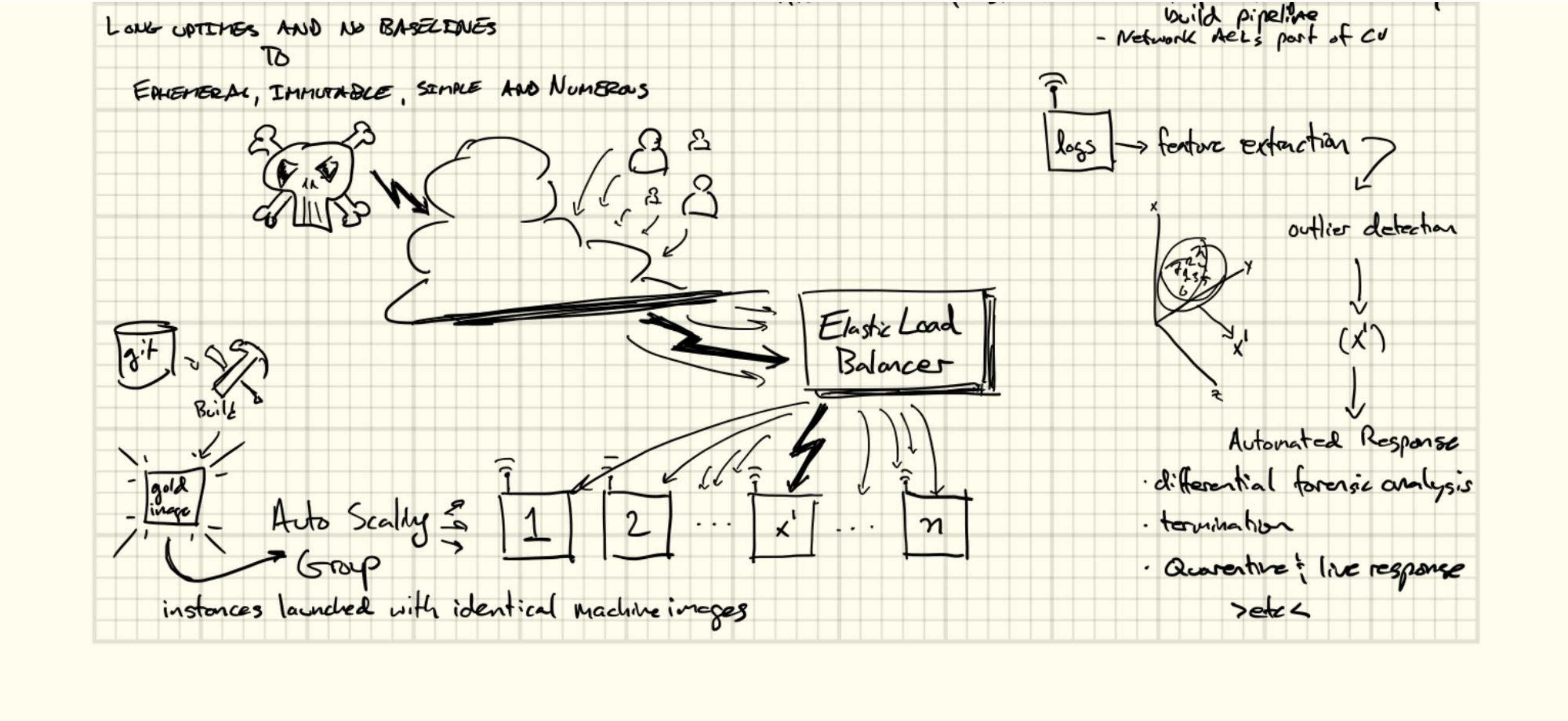
```
// pull in details
u16 family = 0, lport = 0;
bpf_probe_read(&family, sizeof(family), &newsk->__sk_common.skc_family);
bpf_probe_read(&lport, sizeof(lport), &newsk->__sk_common.skc_num);

if (family == AF_INET) {
    struct ipv4_data_t data4 = {.pid = pid, .ip = 4};
    data4.ts_us = bpf_ktime_get_ns() / 1000;
    bpf_probe_read(&data4.saddr, sizeof(u32),
        &newsk->__sk_common.skc_rcv_saddr);
    bpf_probe_read(&data4.daddr, sizeof(u32),
        &newsk->__sk_common.skc_daddr);
    data4.lport = lport;
    bpf_get_current_comm(&data4.task, sizeof(data4.task));
    ipv4_events.perf_submit(ctx, &data4, sizeof(data4));

} else if (family == AF_INET6) {
```

from tcpaccept.py

# Summary.

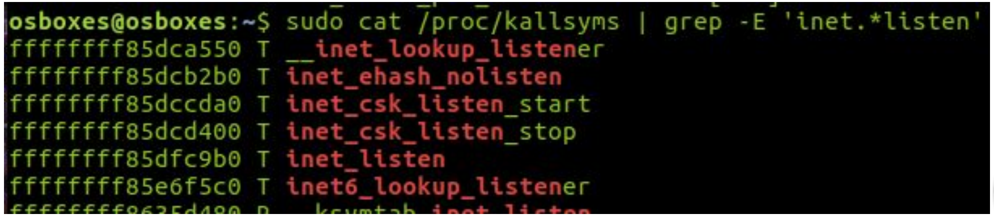

LONG UPTIMES AND NO BASELINES
TO
EPHEMERAL, IMMUTABLE, SIMPLE AND NUMEROUS
build pipeline
- Network ACLs part of CD
logs
feature extraction
outlier detection
(x')
Automated Response
· differential forensic analysis
· termination
· Quarantine & live response
>etc<
git
Build
gold image
Auto Scaling Group
Elastic Load Balancer
1
2
x'
n
instances launched with identical machine images

# Thank you.

# Bonus round.

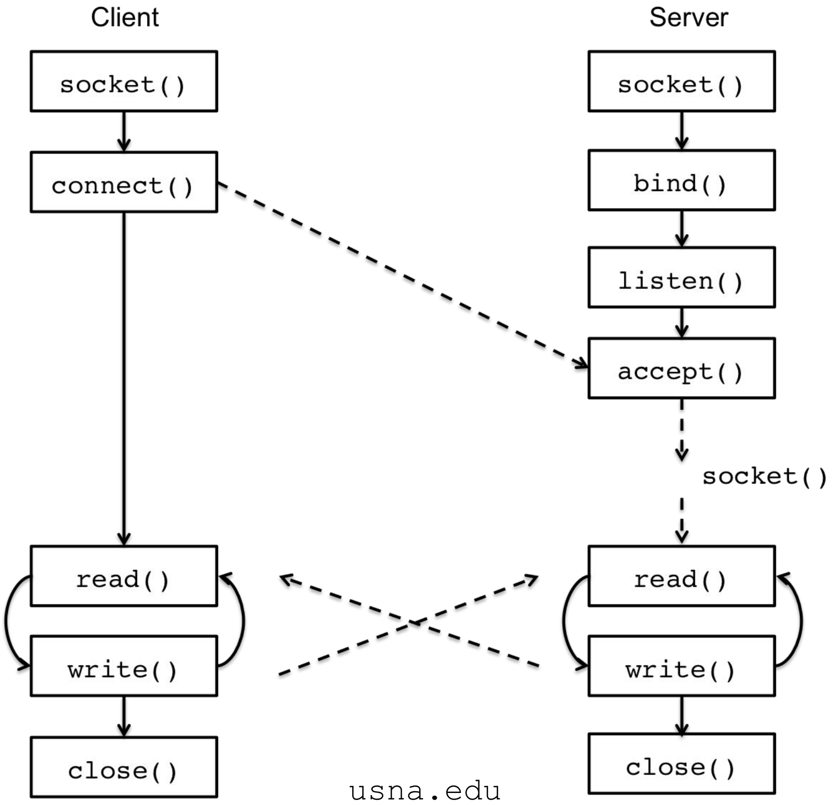
# WHAT'S YOUR SIGN (SYMBOL)

- Example: I want to detect unusual listening ports and what process has bound them.
- Let's look at the socket lifecycle...
  - socket() is too early, no port yet
  - bind() and listen() are good candidates
  - if access is the only concern, accept()
- We can find kernel symbols a number of ways
  - List them: `sudo cat /proc/kallsyms`
  - Use perf-tools to trace ex. `nc -l 12345`



```
osboxes@osboxes:~$ sudo cat /proc/kallsyms | grep -E 'inet.*listen'
ffffffff85dca550 T __inet_lookup_listener
ffffffff85dcb2b0 T inet_ehash_nolisten
ffffffff85dccda0 T inet_csk_listen_start
ffffffff85dcd400 T inet_csk_listen_stop
ffffffff85dfc9b0 T inet_listen
ffffffff85e6f5c0 T inet6_lookup_listener
```

- inet_ is the subsystem hooked in BCC examples and seems to have the context we need... but is not guaranteed stable across Linux builds.



## PROTIP:
## HOOK THE LSM

Most of the relevant functions we care about are already passing through the LSM (with good context), let's Kprobe there (if we can't find a tracepoint) as it will be more stable:

```
osboxes@osboxes:~$ sudo cat /proc/kallsyms | grep security_socket
ffffffff85971510 T security_socket_getpeersec_dgram
ffffffff85974d10 T security_socket_create
ffffffff85974d80 T security_socket_post_create
ffffffff85974e00 T security_socket_bind
ffffffff85974e60 T security_socket_connect
ffffffff85974ec0 T security_socket_listen
ffffffff85974f10 T security_socket_accept
ffffffff85974f60 T security_socket_sendmsg
```

```
_SECURITY_NETWORK

unix_stream_connect(struct sock *sock,
unix_may_send(struct socket *sock,  st
socket_create(int family, int type, int protocol, int kern);
socket_post_create(struct socket *sock, int family,
                   int type, int protocol, int kern);
socket_bind(struct socket *sock, struct sockaddr *address, int addrlen);
socket_connect(struct socket *sock, struct sockaddr *address, int addrlen);
socket_listen(struct socket *sock, int backlog);
socket_accept(struct socket *sock, struct socket *newsock);
socket_sendmsg(struct socket *sock, struct msghdr *msg, int size);
socket_recvmsg(struct socket *sock, struct msghdr *msg,
               int size, int flags);
```

/include/linux/security.h

# The end end.