Rewriting Pointer Dereferences in bcc with Clang

Paul Chaignon

Orange Labs, France

February 3, 2019

The bcc Project

What's bcc?

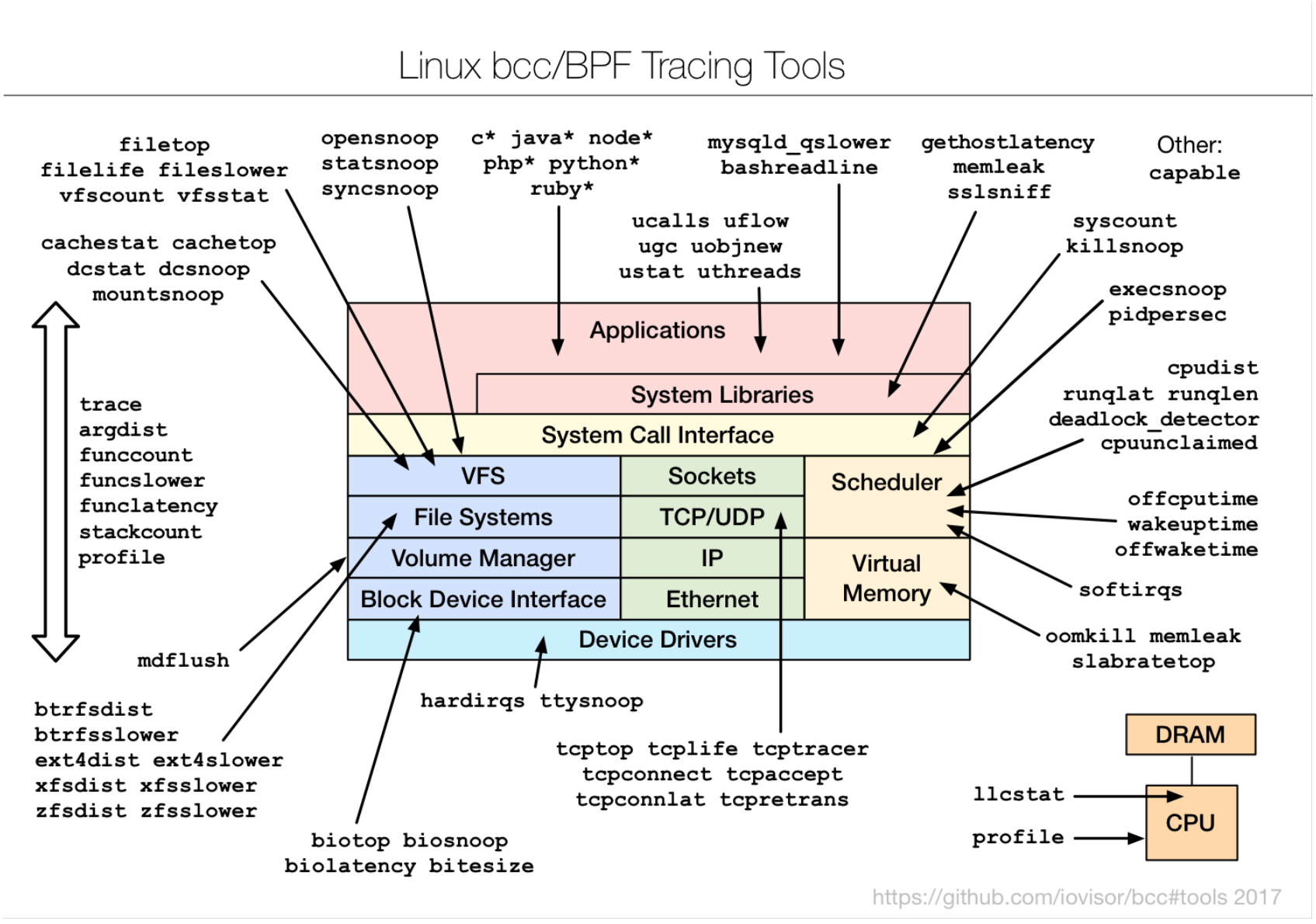
1. Collection of BPF-based tracing tools for Linux
2. Library to ease the development of BPF programs

BPF VM in the Linux Kernel

```
 1  r2 = 0
 2  *(u64 *)(r10 -8) = r2
 3  r1 = *(u64 *)(r1 +16)
 4  *(u64 *)(r10 -16) = r1
 5  r1 = 0xffff9cfb706bd000
 6  r2 = r10
 7  r2 += -16
 8  call bpf_map_lookup_elem#1
 9  if r0 != 0x0 goto pc+14
10  r1 = 0xffff9cfb706bd000
11  r6 = r10
12  r6 += -16
13  r3 = r10
14  r3 += -8
15  r2 = r6
16  r4 = 1
17  call bpf_map_update_elem#2
18  r1 = 0xffff9cfb706bd000
19  r2 = r6
20  call bpf_map_lookup_elem#1
21  if r0 == 0x0 goto pc+3
22  r1 = *(u64 *)(r0 +0)
23  r1 += 1
24  *(u64 *)(r0 +0) = r1
25  r0 = 0
26  exit
```

- Bytecode programs loaded in kernel
- Attached at hook points to process events
- Verified by the kernel to prevent crashes
- Can call some functions outside the VM

Tracing Tools in bcc



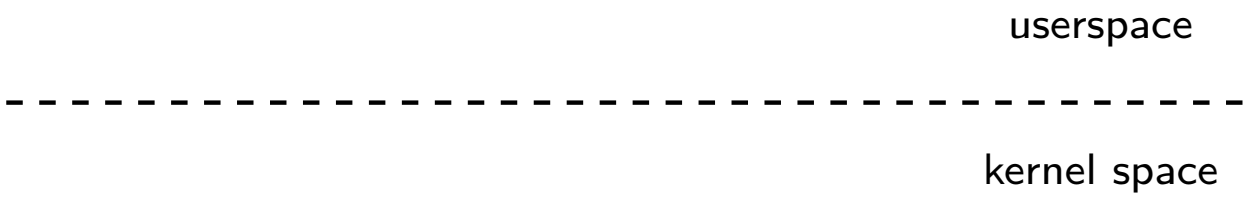
DEMO?

The bcc Library

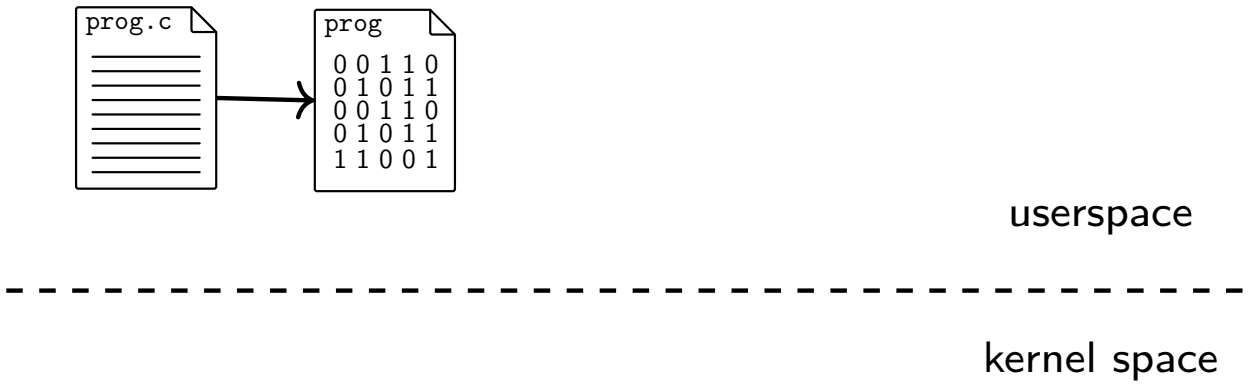
```
from bcc import BPF
# load BPF program
b = BPF(text="""
struct key_t {
    u64 location;
};

BPF_HASH(drops, struct key_t);

TRACEPOINT_PROBE(skb, kfree_skb) {
    u64 zero = 0, *count;
    struct key_t key = {};

    // args is from /sys/kernel/debug/tracing/e
    key.location = (u64)args->location;
    count = drops.lookup_or_init(&key, &zero);
    (*count)++;
    return 0;
};
""")
```

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

# format output
try:
    sleep(99999999)
except KeyboardInterrupt:
    pass

print("\n%-16s %-26s %8s" % ("ADDR", "FUNC", "COUNT"))
drops = b.get_table("drops")
print(drops.items()[0][1].value)
for k, v in sorted(drops.items(),
                   key=lambda elem: elem[1].value):
    print("%-16x %-26s %8d"
          % (k.location, b.ksym(k.location), v.value))
```

Using BPF in the Kernel

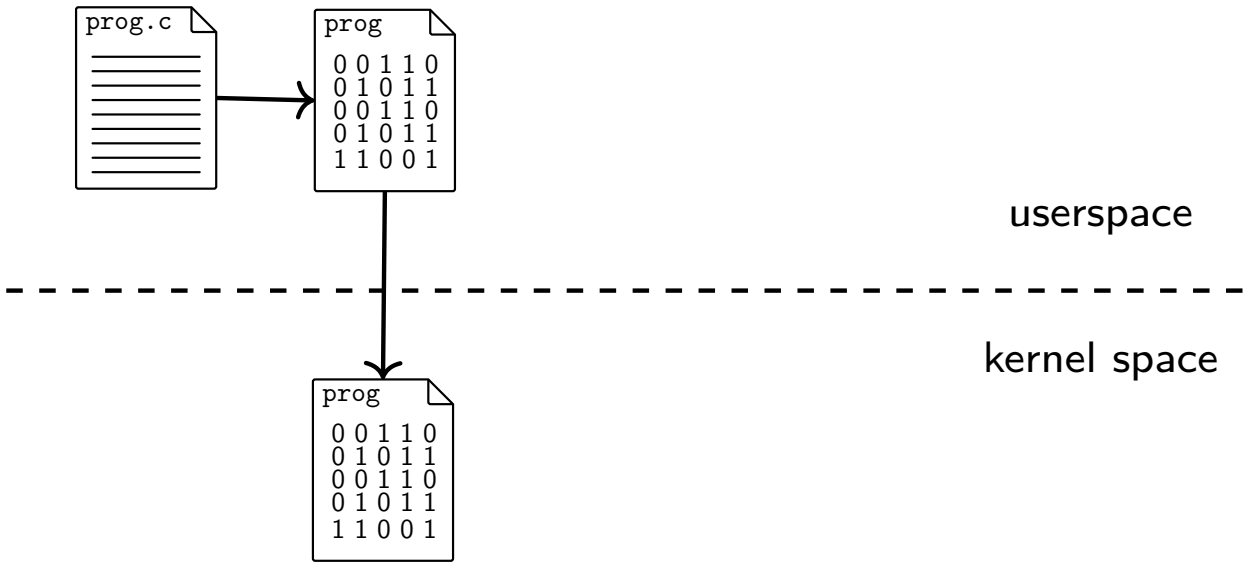


Using BPF in the Kernel



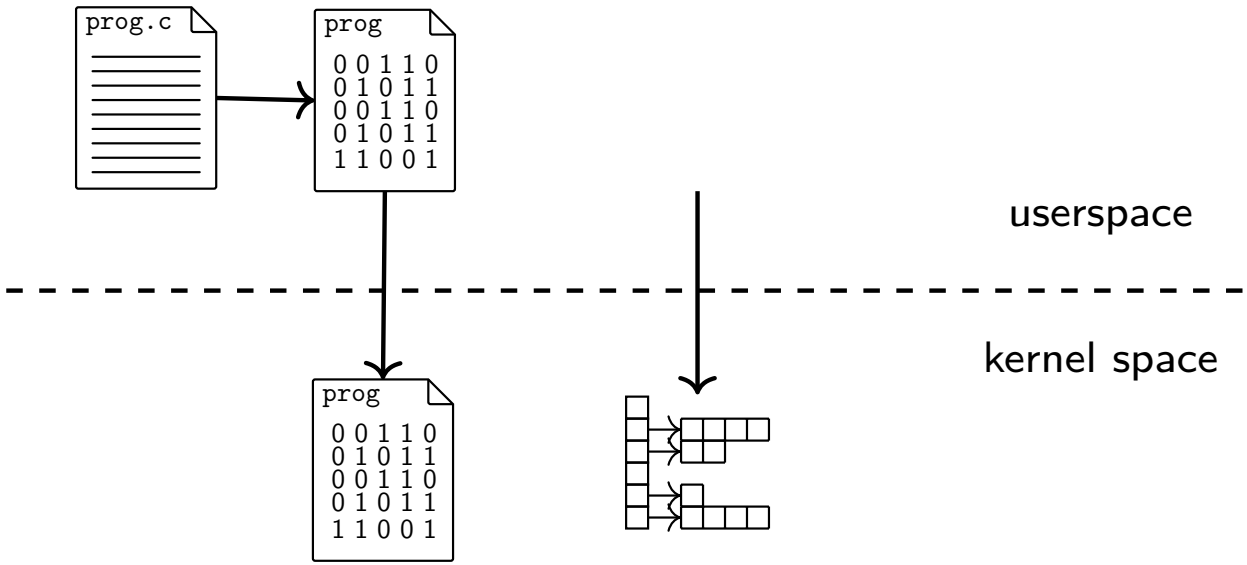
1. Compile from C to BPF using Clang

Using BPF in the Kernel



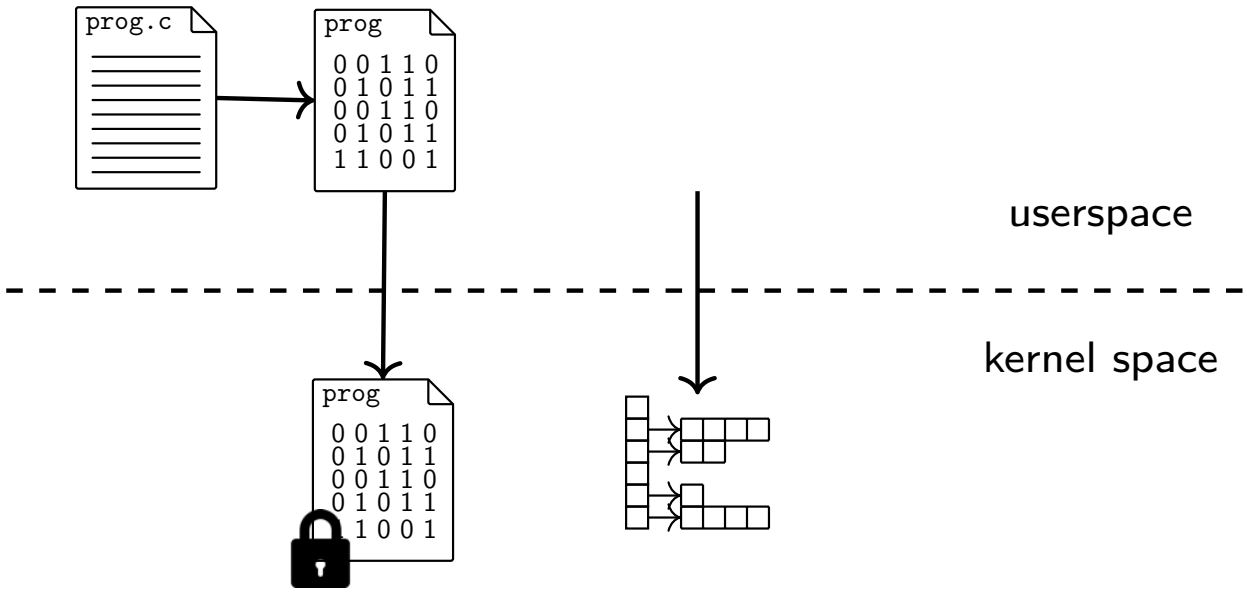
1. Compile from C to BPF using Clang
2. Load program in kernel

Using BPF in the Kernel



1. Compile from C to BPF using Clang
2. Load program in kernel
3. Create maps for BPF program

Using BPF in the Kernel



1. Compile from C to BPF using Clang
2. Load program in kernel
3. Create maps for BPF program
4. Attach BPF program to hook point

Clang Rewriter in bcc

Parsing and Rewriting the C Code

The Clang rewriter is used to:

- Parse map declarations and create them
- Parse function names and attach program correspondingly
- Rewrite function declarations
- Rewrite map accesses
- Replace dereferences of pointers to kernel memory

Reading Kernel Memory from the BPF VM

- Memory access can't be verified statically
- Use external function to read kernel memory
  - Allows for runtime checks

```
1 int count_sched(struct pt_regs *ctx,
2                 struct task_struct *prev) {
3     pid_t p = 0;
4     bpf_probe_read(&p, sizeof(p), (u64)&prev->pid);
5     return p != -1;
6 }
```

Reading Kernel Memory from the BPF VM

- With bcc, you can use dereferences as usual
- Rewritten into `bpf_probe_read` calls
  - Requires us to track all external pointers at C level!

```
1 int count_sched(struct pt_regs *ctx,
2                 struct task_struct *prev) {
3     return prev->pid != -1;
4 }
```

Doesn't aim to be perfect!

- False positives: unnecessary `bpf_probe_read`
  - May lead to syntax errors
  - Additional overhead due to call to external function
- False negatives: missing `bpf_probe_read`
  - Program will be rejected by kernel verifier
  - Hard for user to figure out why

Sources of external pointers

- From the context argument
- From calls to external functions

```
1 int count_sched(struct pt_regs *ctx,
2                  struct task_struct *prev) {
3     struct task_struct *task = bpf_get_current_task();
4     return task->pid != 1 && prev->pid;
5 }
```

What identifies a variable?

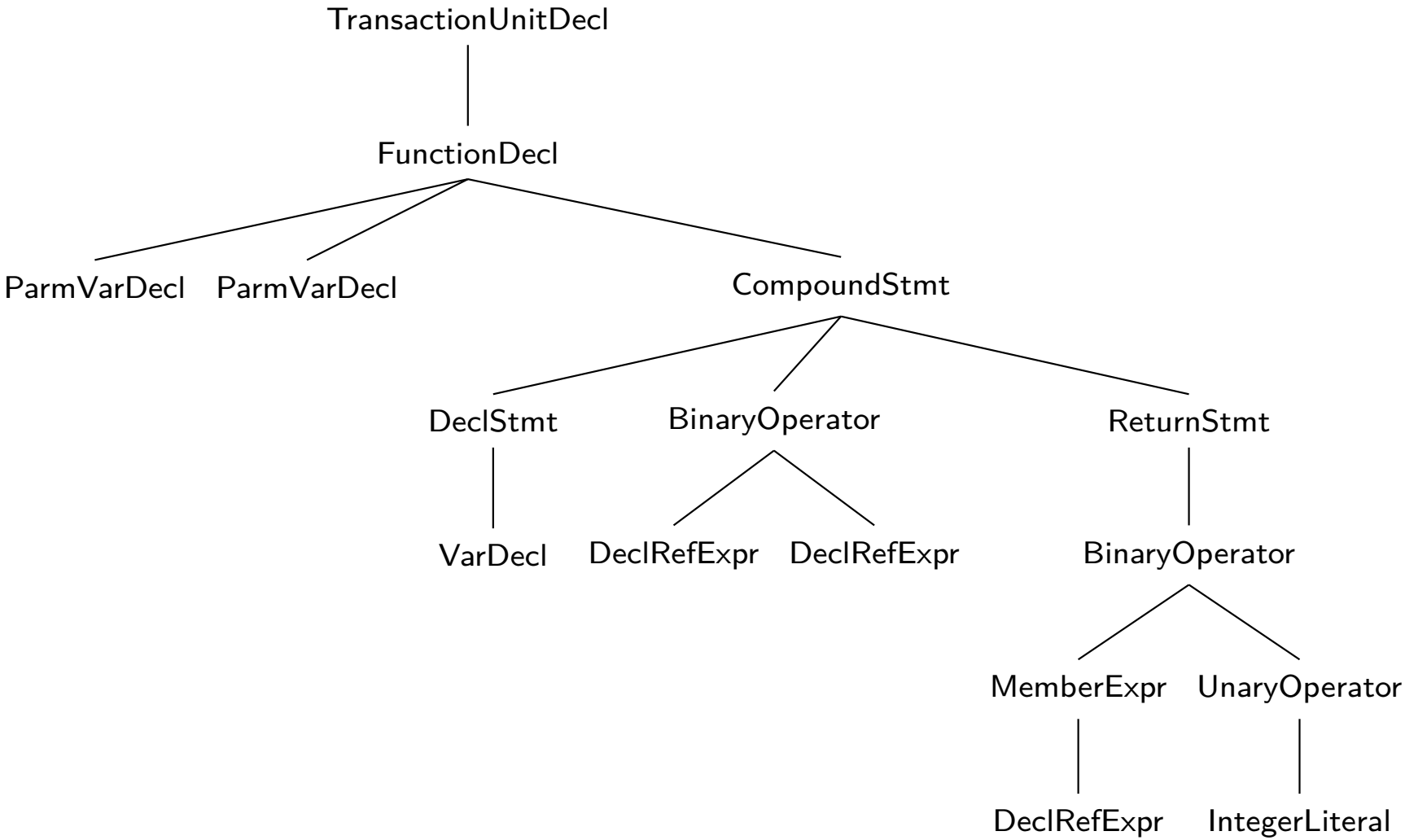
- How to identify and compare variables when looking for external pointers?
  - We use declarations: `Clang::Decl`

```
1  static inline
2  void init_task() {
3      struct task_struct task = {};
4      [...]
5  }
6
7  int count_sched(struct pt_regs *ctx,
8                  struct task_struct *task) {
9      return task->pid != -1;
10 }
```

Traversing the AST

1. First AST traversal to track all external pointers through the code
   - Follow function calls
   - Update set of external pointers as we go
2. Second AST traversal to rewrite external pointer dereferences

Traversing the AST



Tracking Levels of Indirections

```
1 int test(struct pt_regs *ctx, struct sock *sk) {
2     struct sock **ptr;
3     [...]
4     *ptr = sk;
5     return ((struct sock *)(*ptr))->sk_daddr;
6 }
```

- We don't want to rewrite pointers to external pointers

Tracking Levels of Indirections

```
1 int test(struct pt_regs *ctx, struct sock *sk) {
2     struct sock **ptr;
3     [...]
4     *ptr = sk;
5     return ((struct sock *)(*ptr))->sk_daddr;
6 }
```

- We don't want to rewrite pointers to external pointers
- Need to track the levels of indirections for all external pointers!

Tracking External Pointers Through Maps

```
 1 BPF_HASH(currsock, u32, struct sock *);
 2 int trace_entry(struct pt_regs *ctx, struct sock *sk,
 3                 struct sockaddr *uaddr, int addr_len) {
 4     [...]
 5     currsock.update(&pid, &sk);
 6     return 0;
 7 };
 8 int trace_exit(struct pt_regs *ctx) {
 9     [...]
10     struct sock **skpp = currsock.lookup(&pid);
11     if (skpp) {
12         return (*skpp)->__sk_common.skc_dport;
13     }
14     return 0;
15 }
```

- External pointers might be stored in maps

Tracking External Pointers Through Maps

```
 1 BPF_HASH(currsock, u32, struct sock *);
 2 int trace_entry(struct pt_regs *ctx, struct sock *sk,
 3                 struct sockaddr *uaddr, int addr_len) {
 4     [...]
 5     currsock.update(&pid, &sk);
 6     return 0;
 7 };
 8 int trace_exit(struct pt_regs *ctx) {
 9     [...]
10     struct sock **skpp = currsock.lookup(&pid);
11     if (skpp) {
12         return (*skpp)->__sk_common.skc_dport;
13     }
14     return 0;
15 }
```

- External pointers might be stored in maps
  1. Track external pointers
  2. Identify maps with external pointers
  3. Track remaining external pointers

Conclusion

- Rewriting external pointers at C level is a pain
  - Requires several AST traversals
  - Implementation more complex than we'd like
  - Struggles with `Clang::Rewriter::ReplaceText`
  - Still not complete
- Other, better approaches?
  - Rewrite at bytecode level?
  - Rewrite all structures from kernel headers?
  - Ask developer to label external pointers
  - No rewriting at all?
  - ...

Thank you for listening!