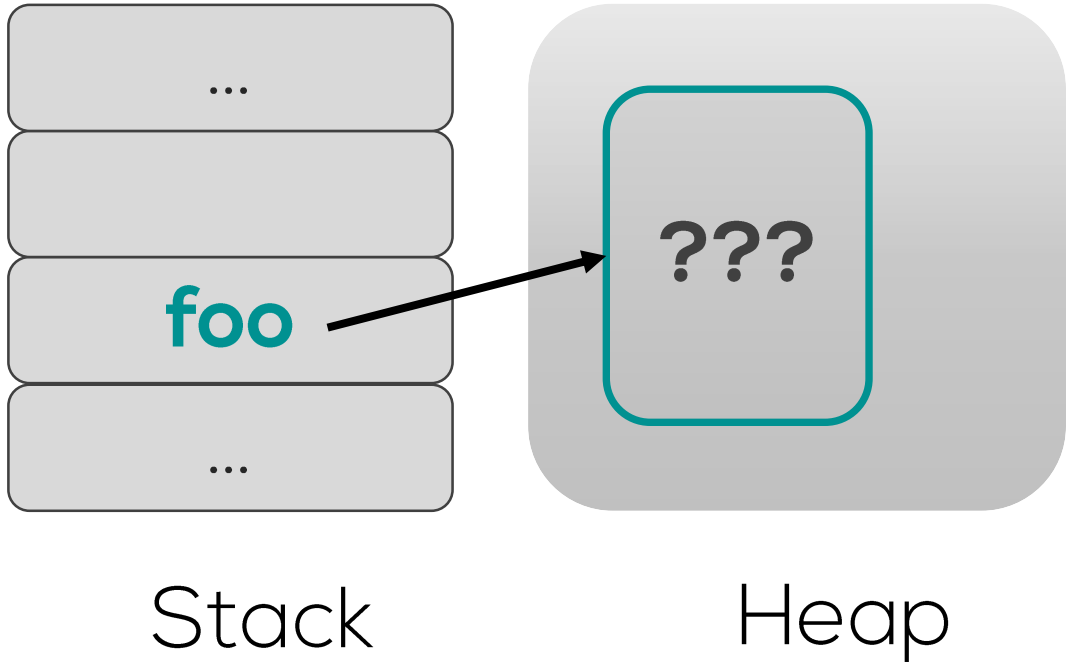
Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying

Kangjie Lu, Marie-Therese Walter, **David Pfaff**,
Stefan Nürnberger, Wenke Lee, and Michael Backes

Georgia Tech, CISPA, Saarland University, MPI-SWS, DFKI

Starting with the famous brother: Use-after-free

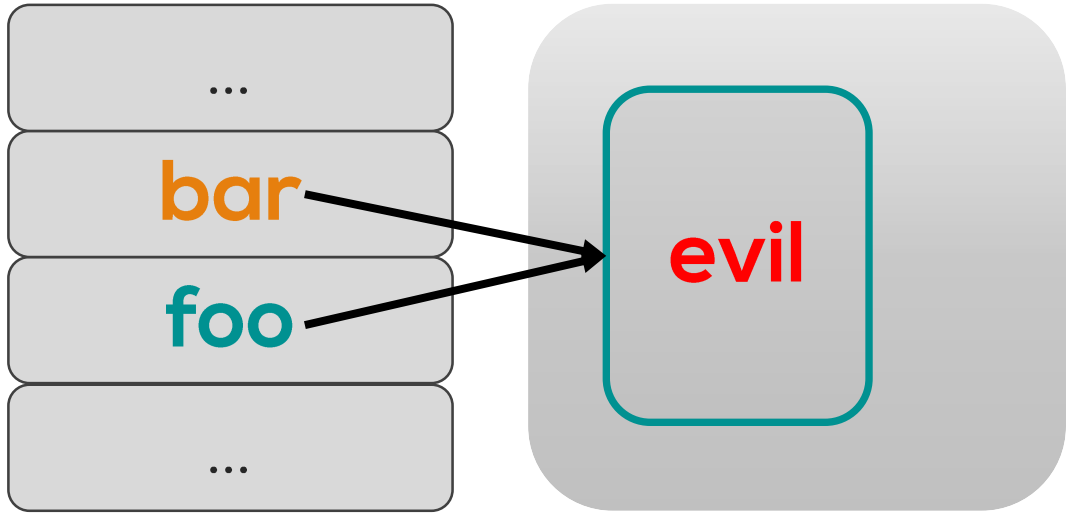
```
char* foo = (char*) malloc (100);
*foo = "abc";
free (foo);


printf(%s, *foo);
```



*foo = **???**

Starting with the famous brother: Use-after-free

```
char* foo = (char*) malloc (100);
*foo = "abc";
free (foo);
char* bar = (char*) malloc (100);
*bar = "evil";
printf(%s, *foo);
```



*foo = **"evil"**

Uninitialized use vulnerabilities on the stack

Vulnerable Function:

```
struct A *foo;
...
foo->complete();
```

Uninitialized use vulnerabilities on the stack

Setup function:

```
int buf[50];
for (i=0; i<50; i++)
{
      buf[i]=evil;
}
```

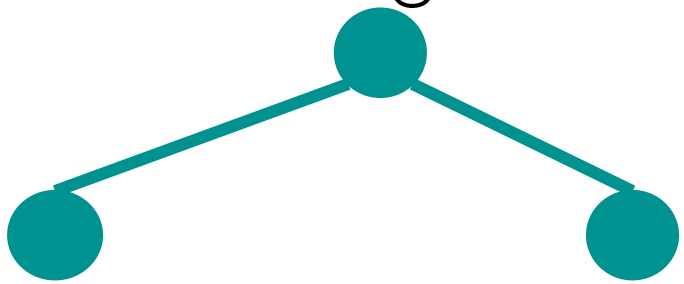
Vulnerable Function:

```
struct A *foo;

...
foo->complete();
```

evil->complete()

Uninitialized uses pose critical security risks

Overlapping Memory



1

Setup Function:
Control uninitialized variable

2

Vulnerable Function:
Uninitialized Use

Arbitrary read, write, and execution

In reality, uninitialized-use problems are overlooked

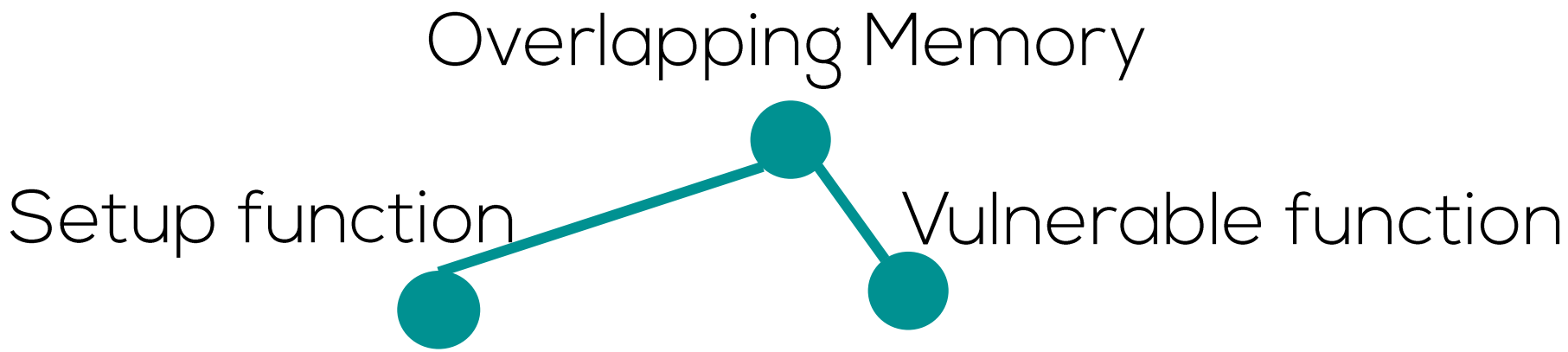
- Uninitialized uses were regarded as undefined behaviors

2015-2016: **16** Linux kernel patches, **1** CVE

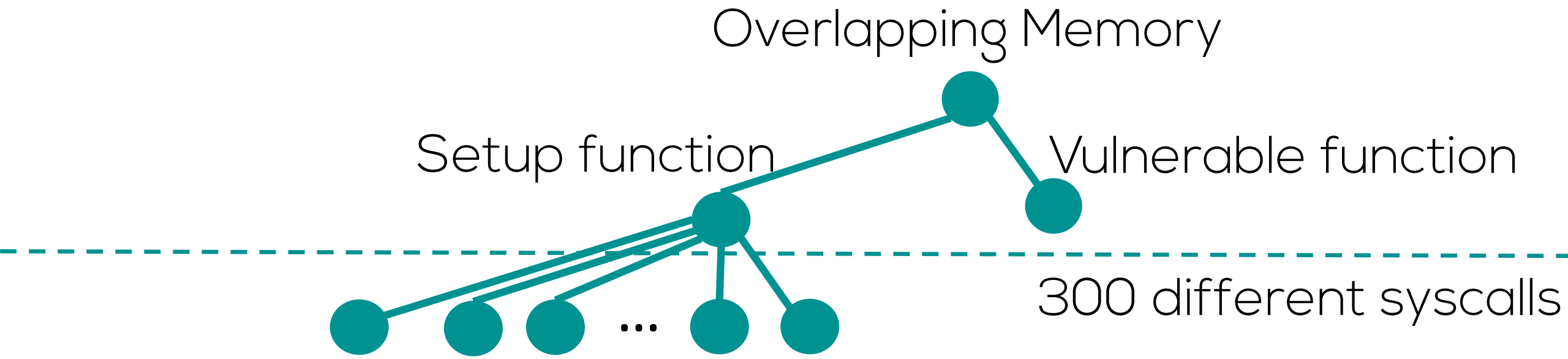
- Full memory safety techniques exclude uninitialized uses as a target

- Widespread belief: uninitialized memory is uncontrollable

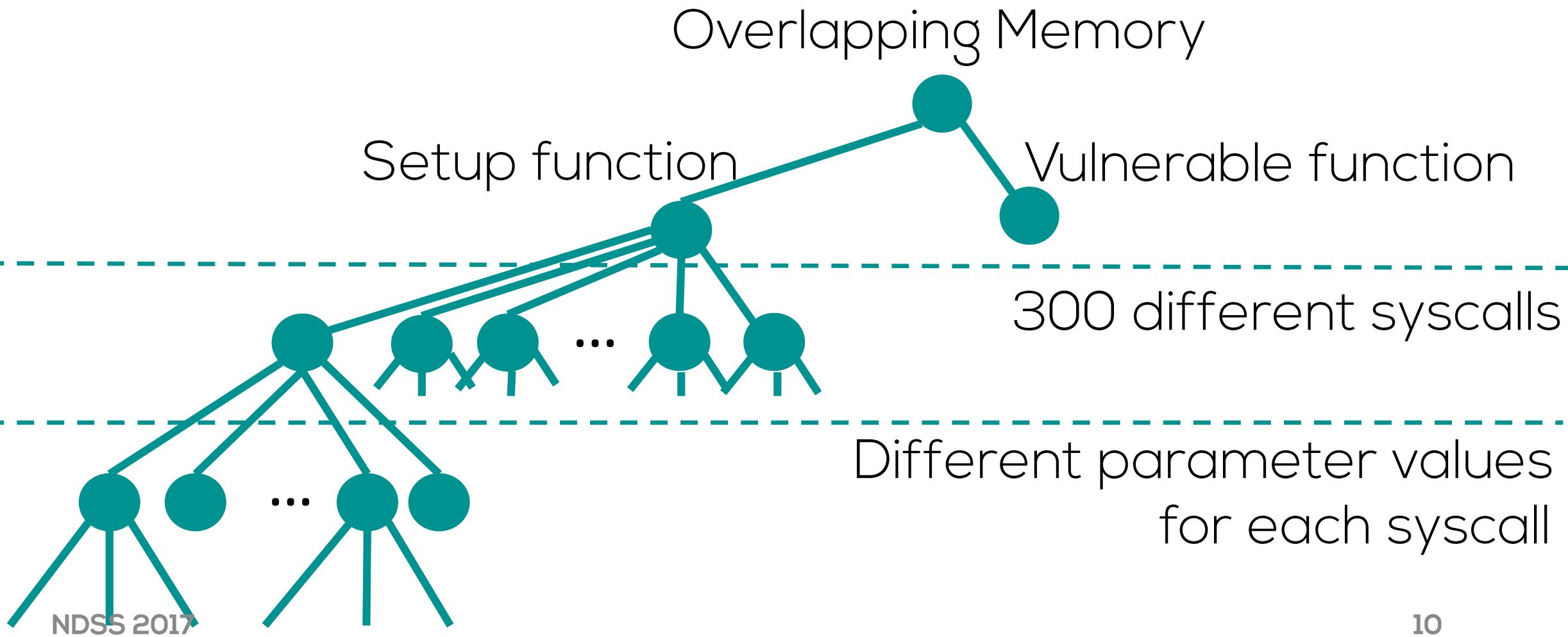
Manually exploiting unitialized uses in the Linux kernel stack is difficult



Manually exploiting unitialized uses in the Linux kernel stack is difficult



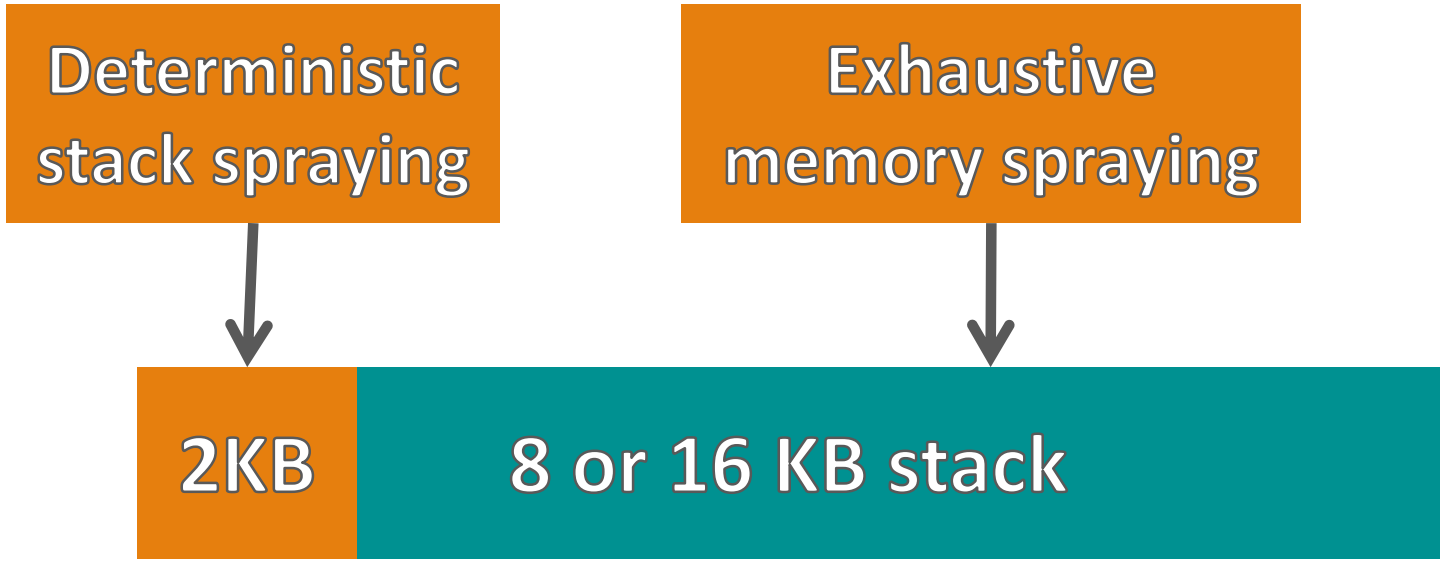
Manually exploiting unitialized uses in the Linux kernel stack is difficult



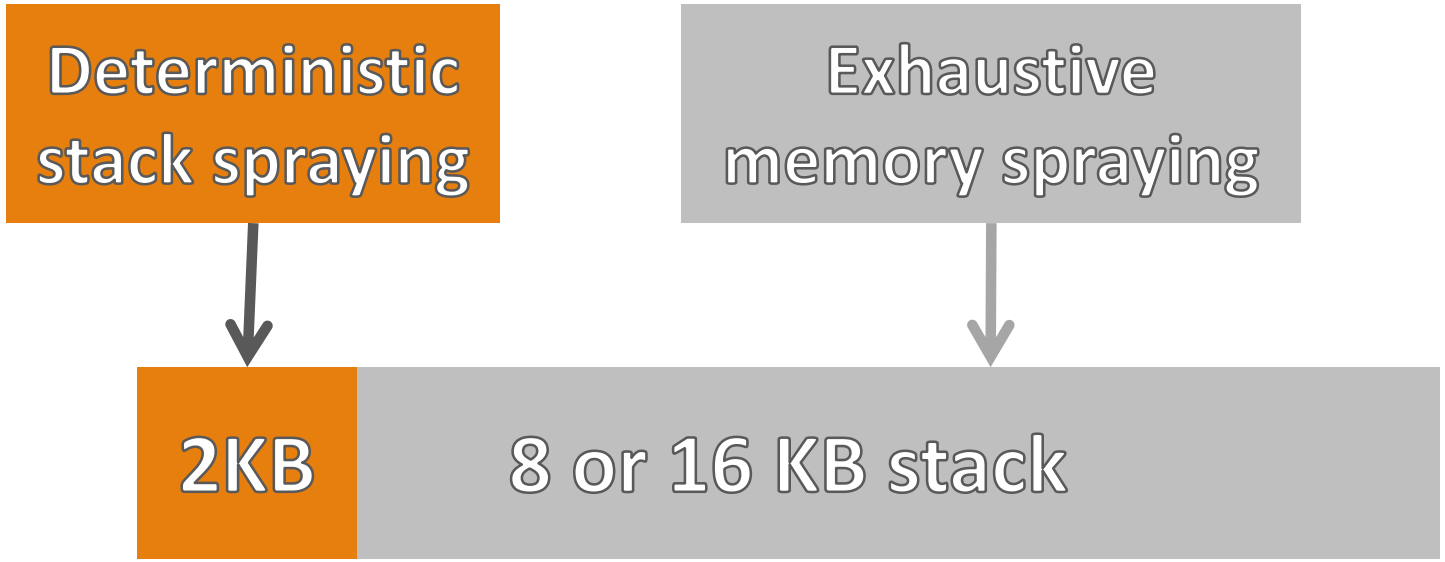
Primary kernel stack usage

- 90% of all system calls only use the first 2KB of the kernel stack
- Most interesting region to target

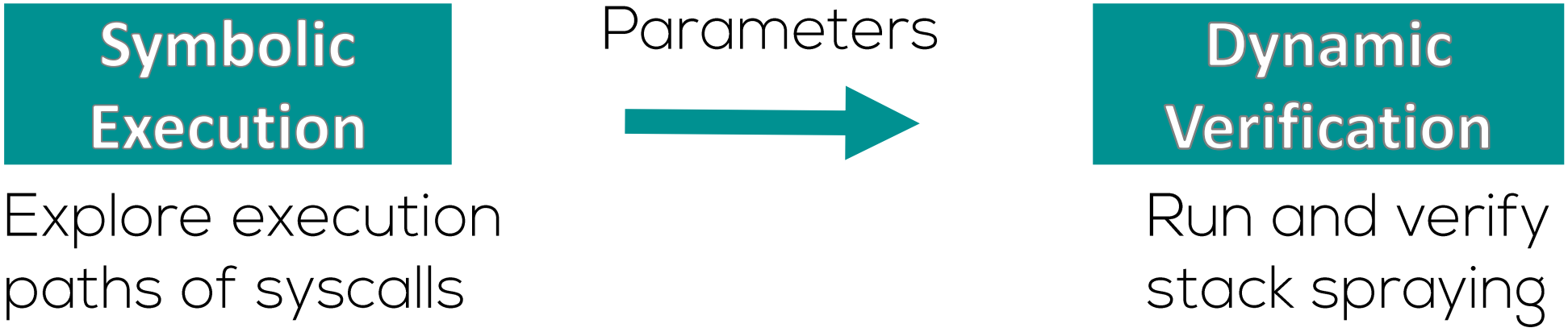
Targeted stack spraying



Targeted stack spraying

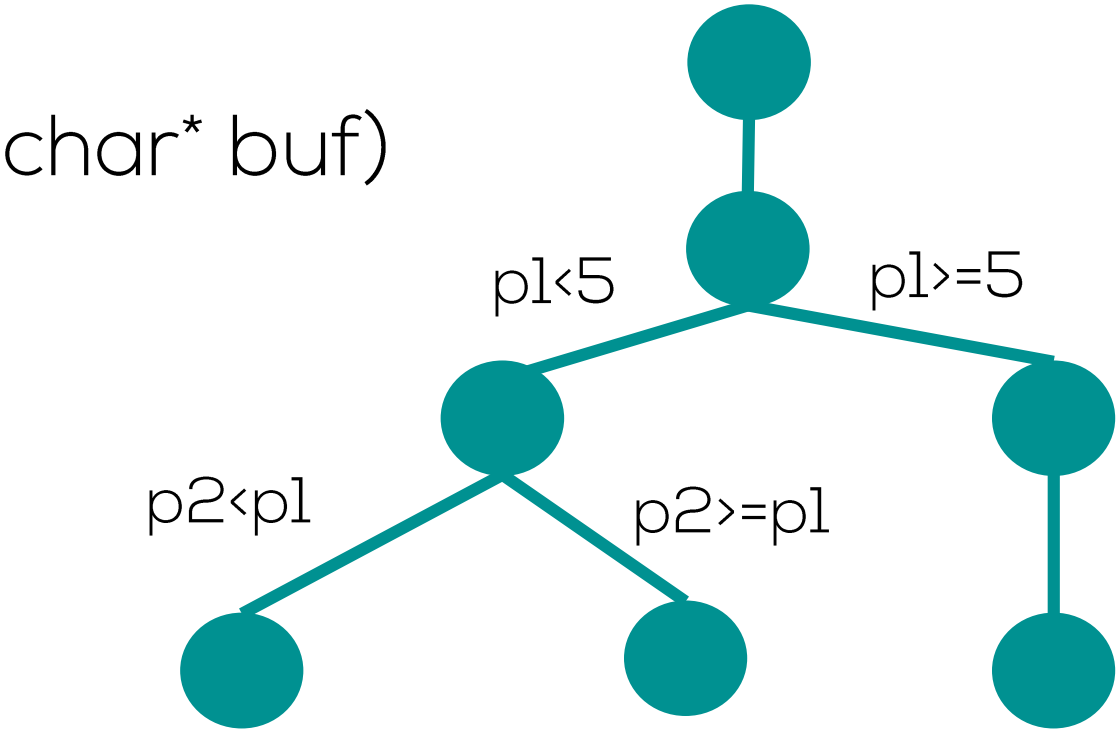


Deterministic stack spraying overview



Path exploration using Symbolic Execution

syscall(int p1, int p2, char* buf)

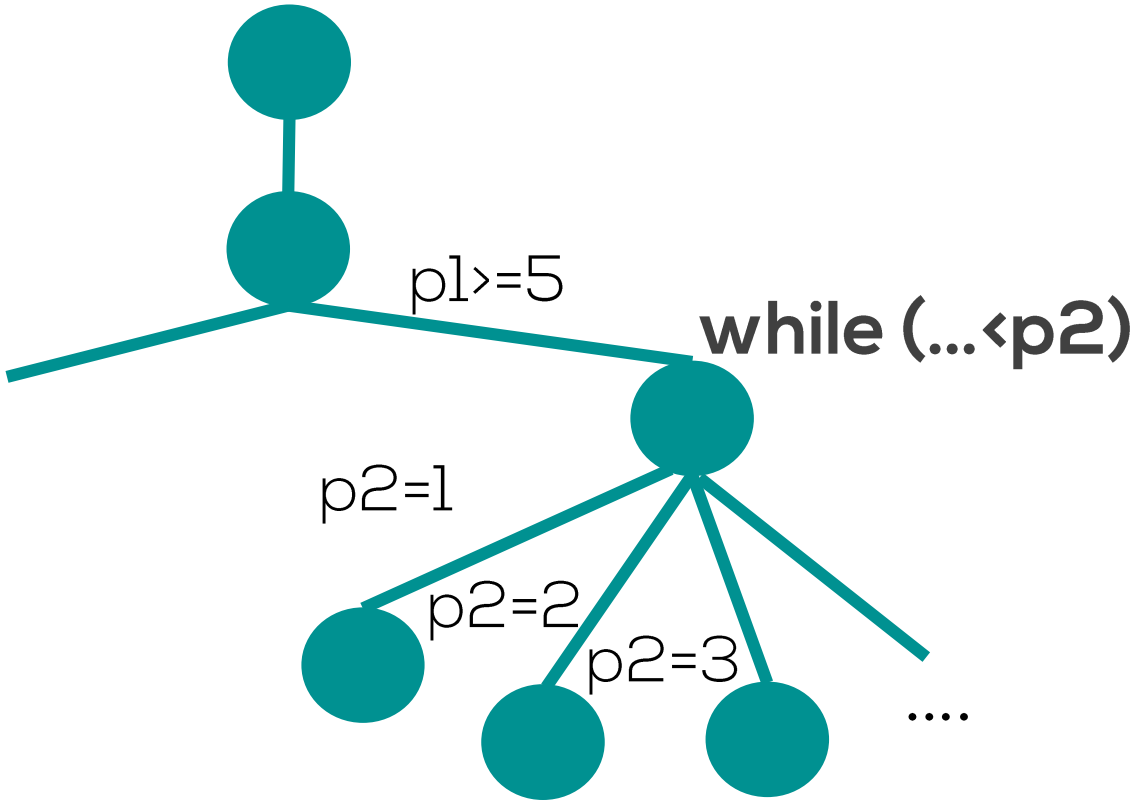


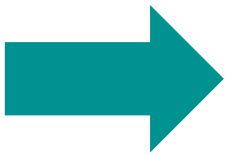
Path 1: p1<5; p2<p1

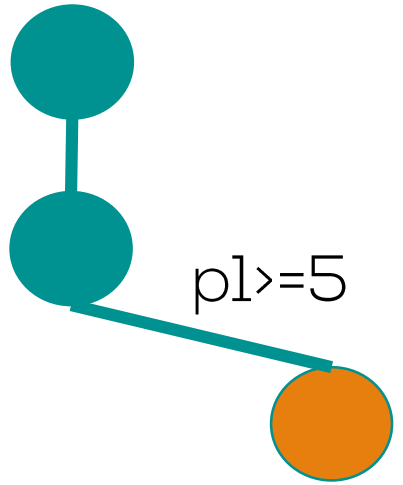
Path 2: p1<5; p2>=p1

Path 3: p1>=5

SE: handling path explosion due to unbounded loops



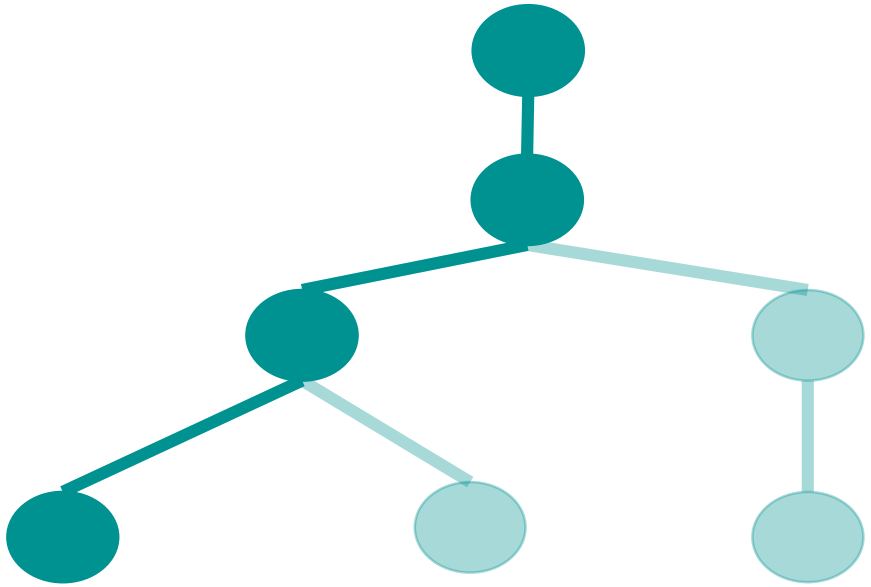




Path 3: p1>=5
LOOP(p2)

Fuzz loop variable (p2) in dynamic verification phase

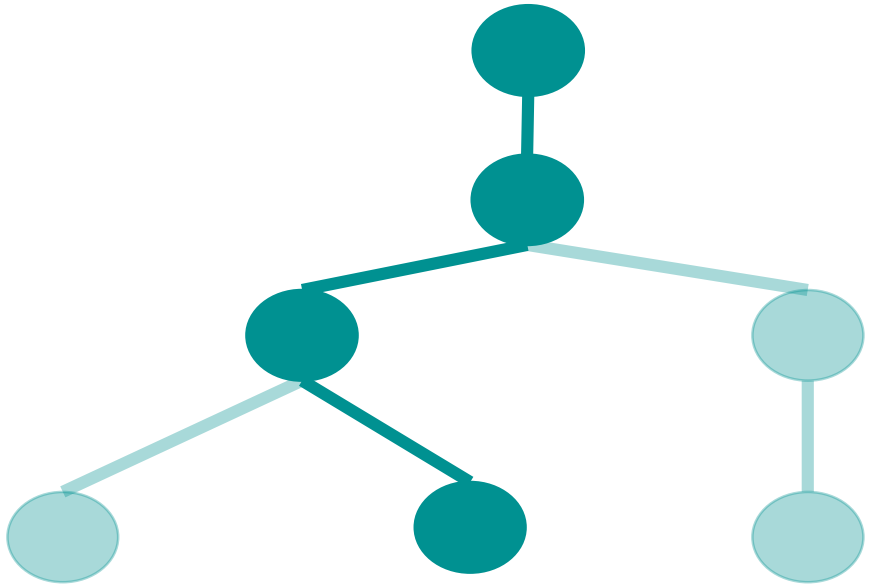
Verify stack spraying by executing paths



Path 1: p1<5; p2<p1

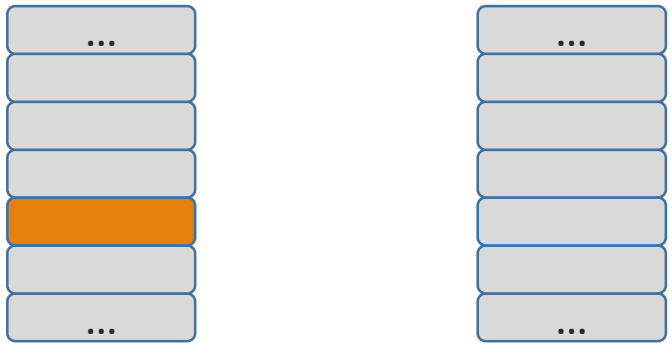
syscall(p1=3, p2=2, buf="UUID")

Verify stack spraying by executing paths

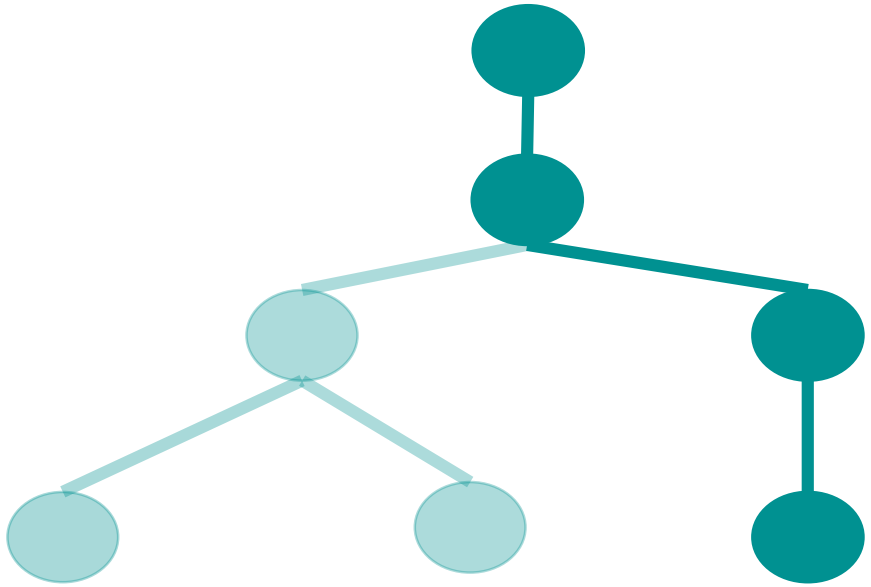


Path 2: p1<5; p2>=p1

syscall(p1=3, p2=3, buf="UUID")

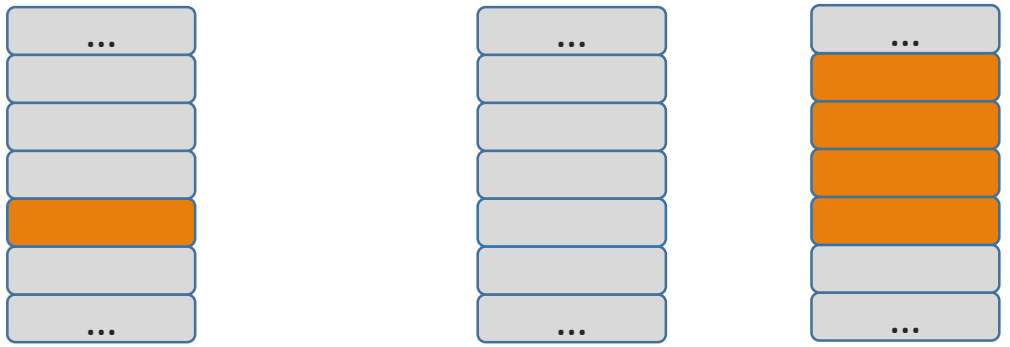


Verify stack spraying by executing paths

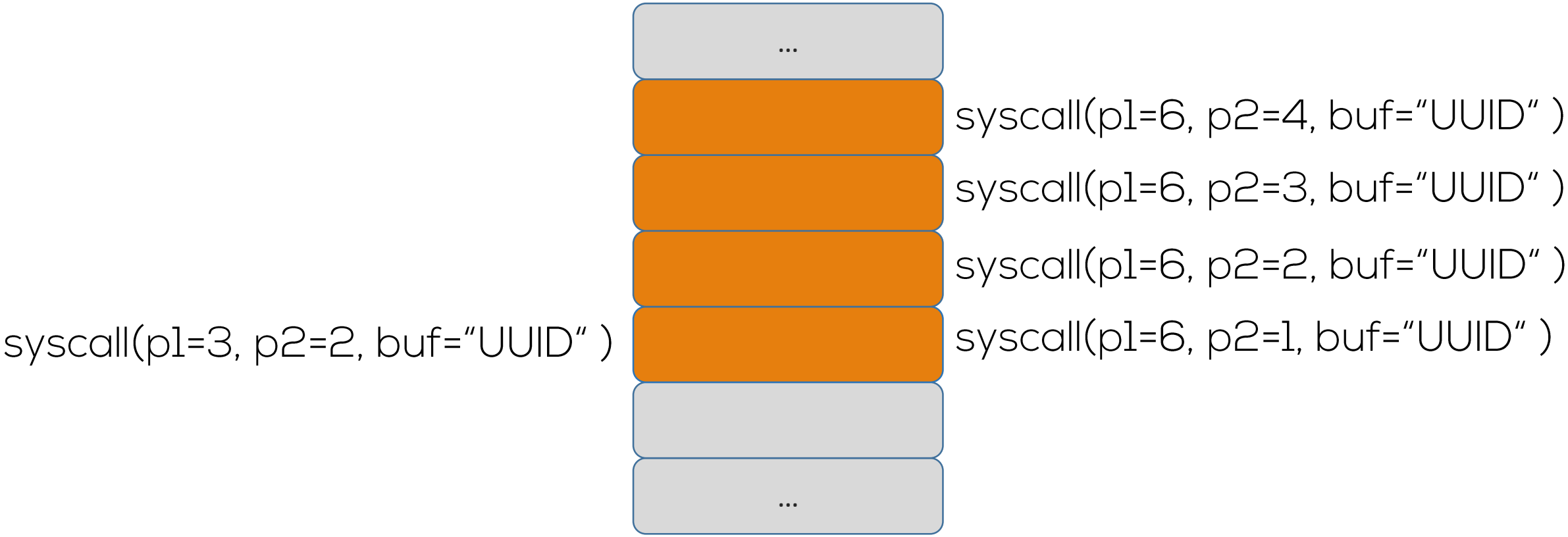


Path 3: p1>=5

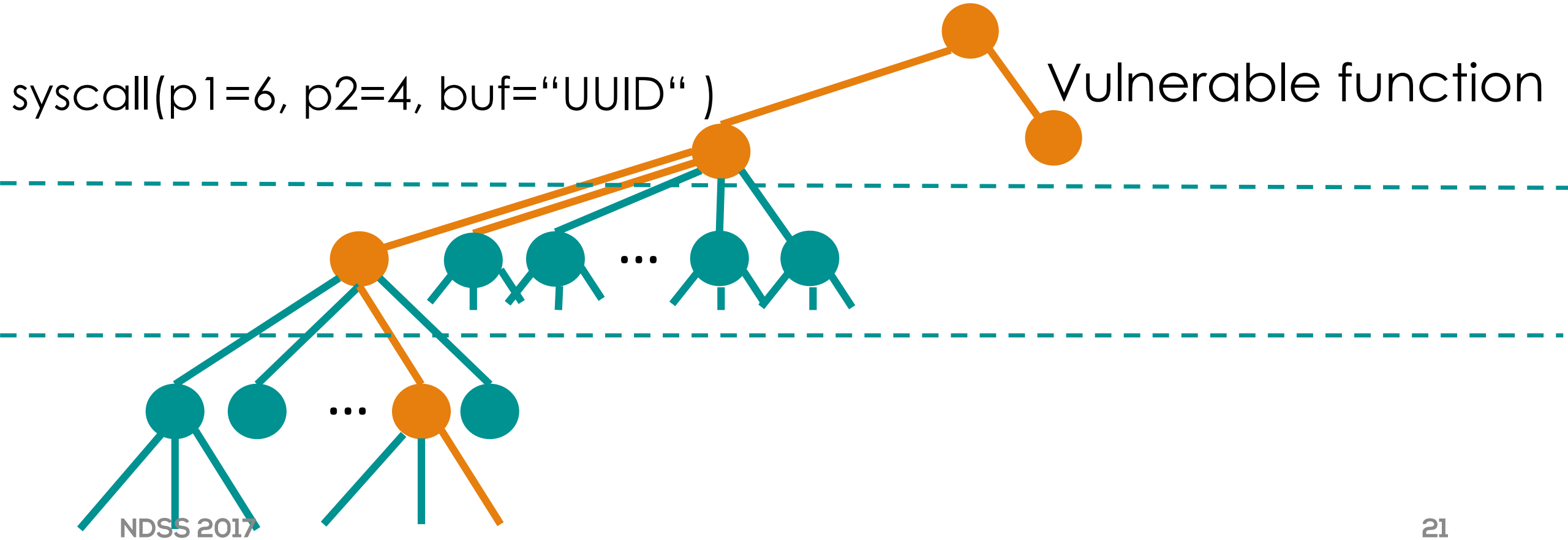
syscall(p1=6, FUZZ(p2), buf="UUID")



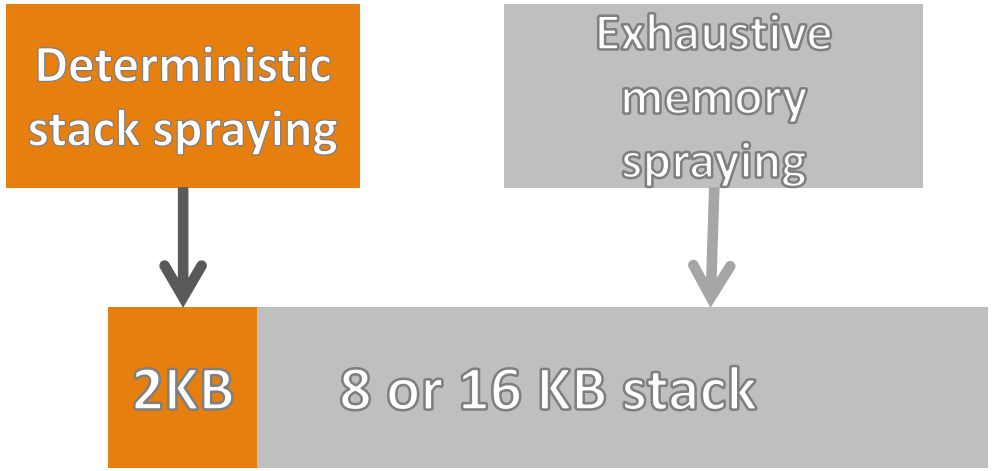
Mapping syscalls and parameters to memory locations



Automatically exploiting uninitialized uses in the Linux kernel is possible!



Achieved Linux kernel stack coverage



Deterministic Stack Spraying: **39%** of 2KB

Deterministc Stack Spraying + Exhaustive Memory Spraying: **91%** of full stack

Real world case study: Linux Kernel Privilege escalation CVE-2010-2963

- CVE-2010-2963: Uninitialized pointer used for write
- Found by Kees Cook

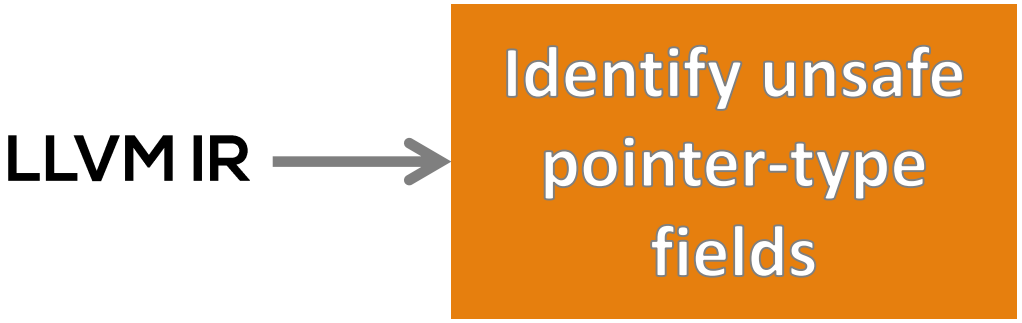
- Setup: get_video_tuner32 → Vuln: get_microcode32

- We automatically found **27** syscalls that can control the uninitialized pointer

Efficient mitigation by zero-initialization

```
struct A *foo;
…
foo->complete();
```

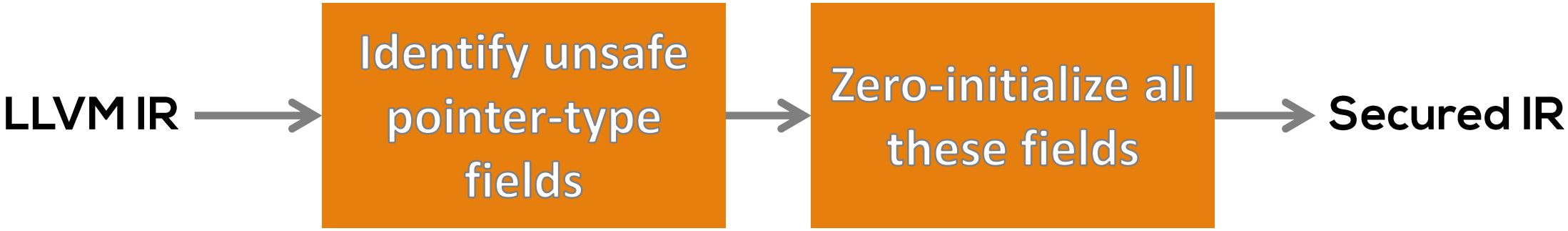
```
struct A *foo;
…
bar(foo);
```



Efficient mitigation by zero-initialization

```
struct A *foo = 0;
…
foo->complete();
```

```
struct A *foo = 0;
…
bar(foo);
```



Mitigation performance overhead

- Syscall performance overhead with LMBench
  - Average: **1.95%**

- User program performance overhead with SPEC benchmarks
  - Average: **0.47%**

Conclusions

- Uninitialized stack variables can be reliably controlled
- Uninitialized use is a critical attack vector
- Memory-safety techniques should include uninitialized use as a prevention target