No source? No problem!

High speed binary fuzzing

Nspace & @gannimo

EPFL hexhive

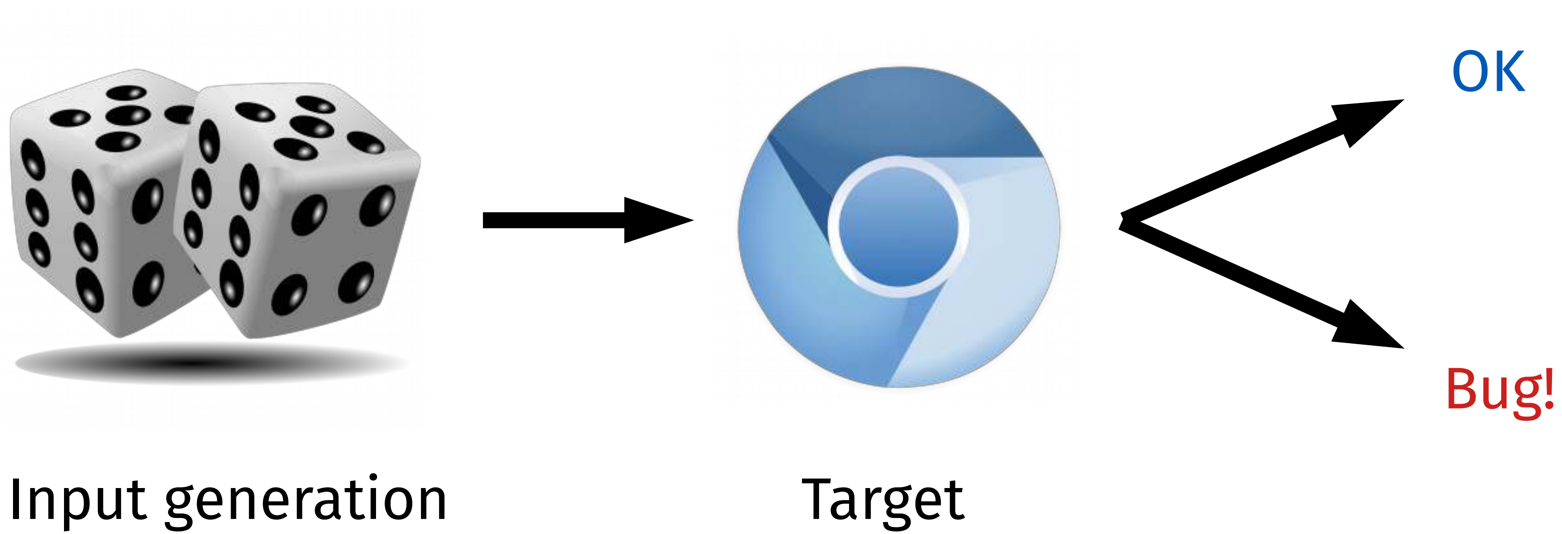
About this talk

- Fuzzing binaries is hard!
  - Few tools, complex setup
- Fuzzing binaries in the kernel is even harder!
- New approach based on ***static*** rewriting

\+ Kernel
Libc
Desktop

≈ 100M LoC

Fuzzing 101

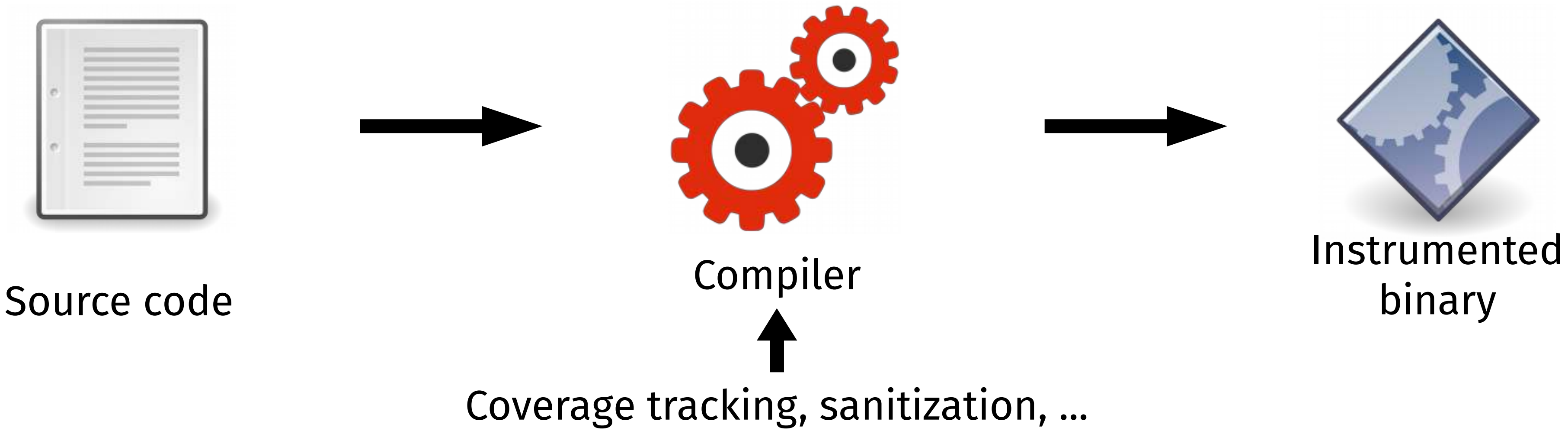


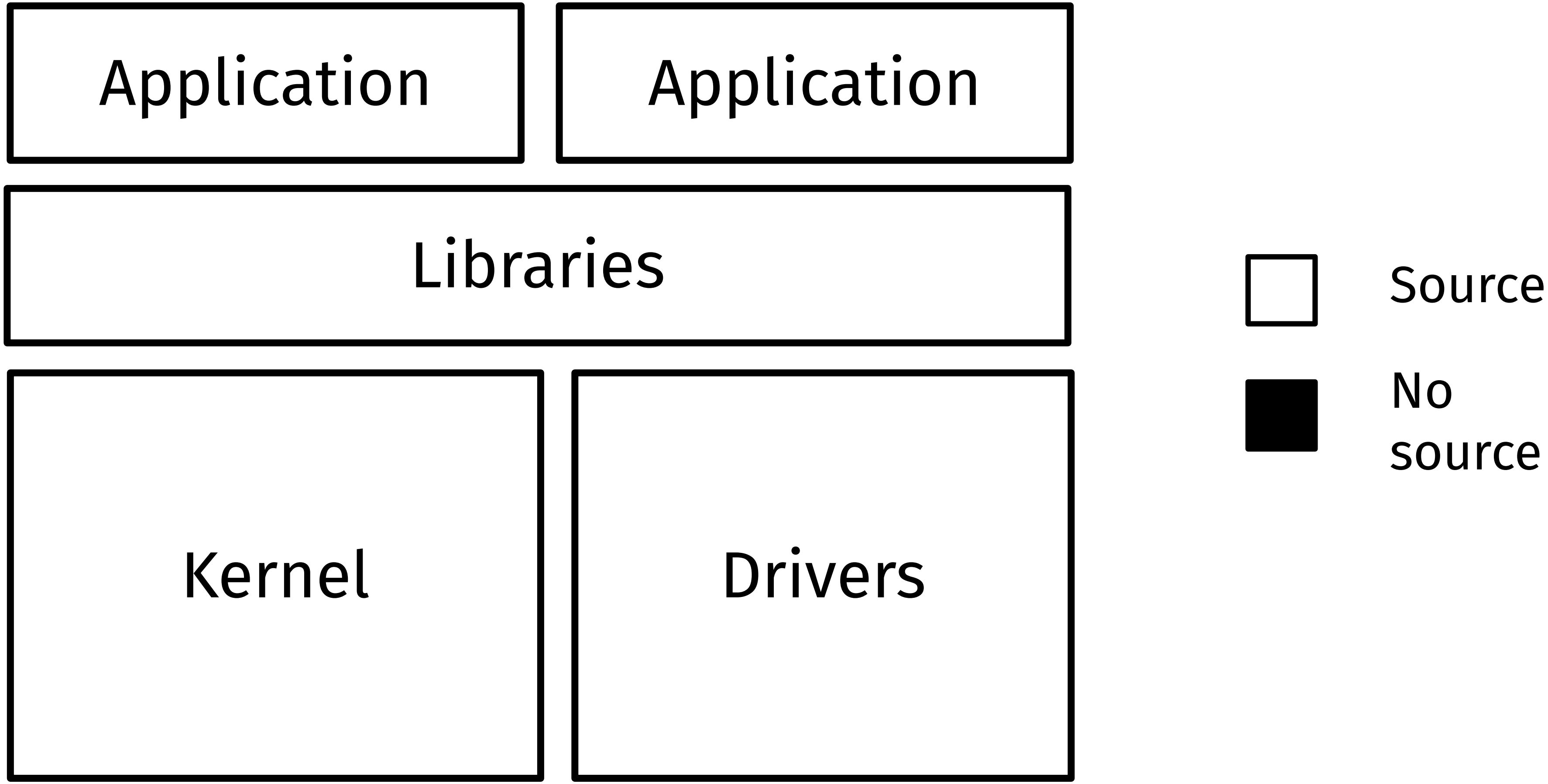
Effective fuzzing 101

- Test cases must ***trigger bugs***
    - Coverage-guided fuzzing
- The fuzzer must ***detect bugs***
    - Sanitization
- ***Speed*** is key (zero sum game)!

Fuzzing with source code

- Add ***instrumentation*** at compile time
  - Short snippets of code for coverage tracking, sanitization, …




Application
Application
Libraries
Kernel
Drivers
Source
No source

Rewriting binaries

- Approach 0: black box fuzzing
- Approach 1: rewrite ***dynamically***
  - Translate target at runtime
  - ***Terrible performance (10-100x slower)***
- Approach 2: rewrite ***statically***
  - More complex analysis
  - …but much better performance!

Static rewriting challenges

- Simply adding code breaks the target

```
mov [rax + rbx*8], rdi
dec rbx
jnz -7
```

```
mov [rax + rbx*8], rdi
<new code>
dec rbx
jnz -7
```

- Need to find ***all*** references and ***adjust*** them

Static rewriting challenges

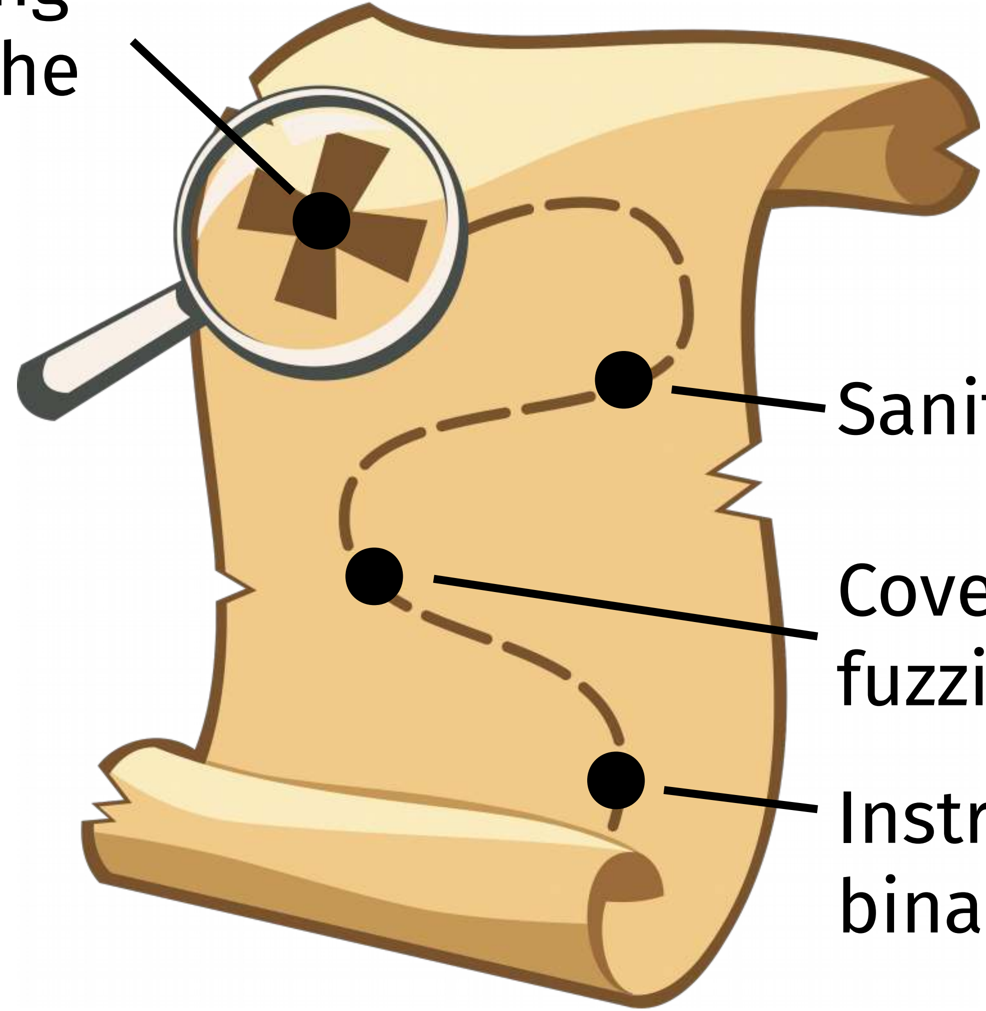
- Scalars and references are indistinguishable
  - Getting it wrong breaks the target

```
mov [rbp-0x8], 0x400aae
```

?

```
long (*foo)(long) = &bar;

long foo = 0x400aae;
```

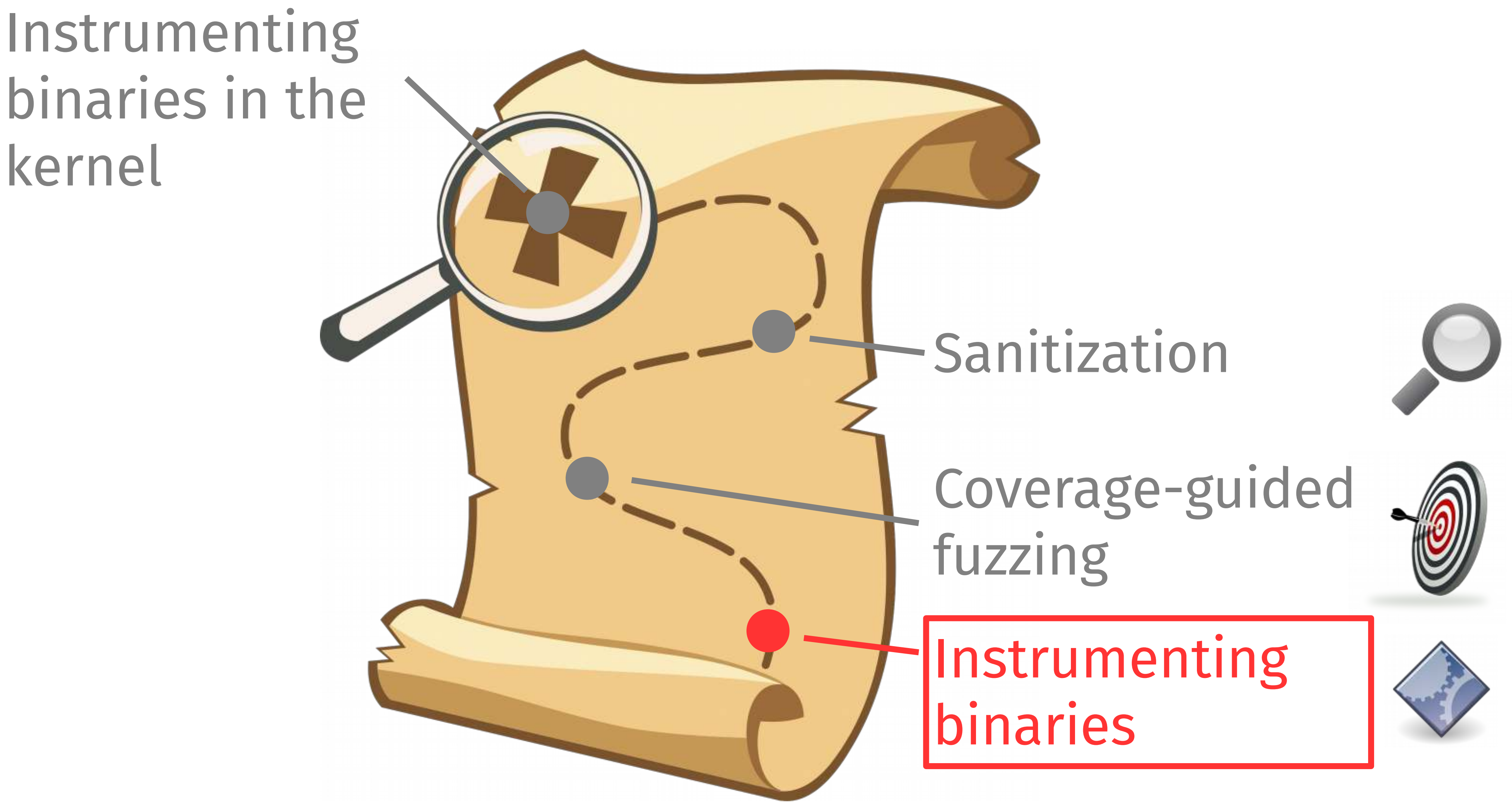
Instrumenting binaries in the kernel



Sanitization

Coverage-guided fuzzing

Instrumenting binaries


Instrumenting binaries in the kernel
Sanitization
Coverage-guided fuzzing
Instrumenting binaries

RetroWrite [Oakland '20]

- System for static binary instrumentation
- Symbolized assembly files easy to instrument
- Implements coverage tracking and binary ASan

Position-independent code

- Code that can be loaded at any address
- Required for: ASLR, shared libraries
- ***Cannot use hardcoded static addresses***
    - Must use relative addressing instead

Position-independent code

- On x86_64, PIC leverages RIP-relative addressing
    - `lea rax, [rip + 0x1234]`

- ***Distinguish references from constants in PIE binaries***
    - RIP-relative = reference, everything else = constant

Symbolization

- Symbolization replaces references with assembler labels

```
lea rax, [rip + 0x1234]
call 0x1337
dec rcx
jnz -15
```

Symbolization

- Symbolization replaces references with assembler labels

1) Relative jumps/calls

```
loop1:
    lea rax, [rip + 0x1234]
    call func1
    dec rcx
    jnz loop1
```

Symbolization

- Symbolization replaces references with assembler labels

1) Relative jumps/calls
2) PC-relative addresses

```
loop1:
   lea rax, [data1]
   call func1
   dec rcx
   jnz loop1
```

Symbolization

- Symbolization replaces references with assembler labels

  1) Relative jumps/calls
  2) PC-relative addresses
  3) Data relocations

```
loop1:
   lea rax, [data1]
   call func1
   dec rcx
   jnz loop1
```

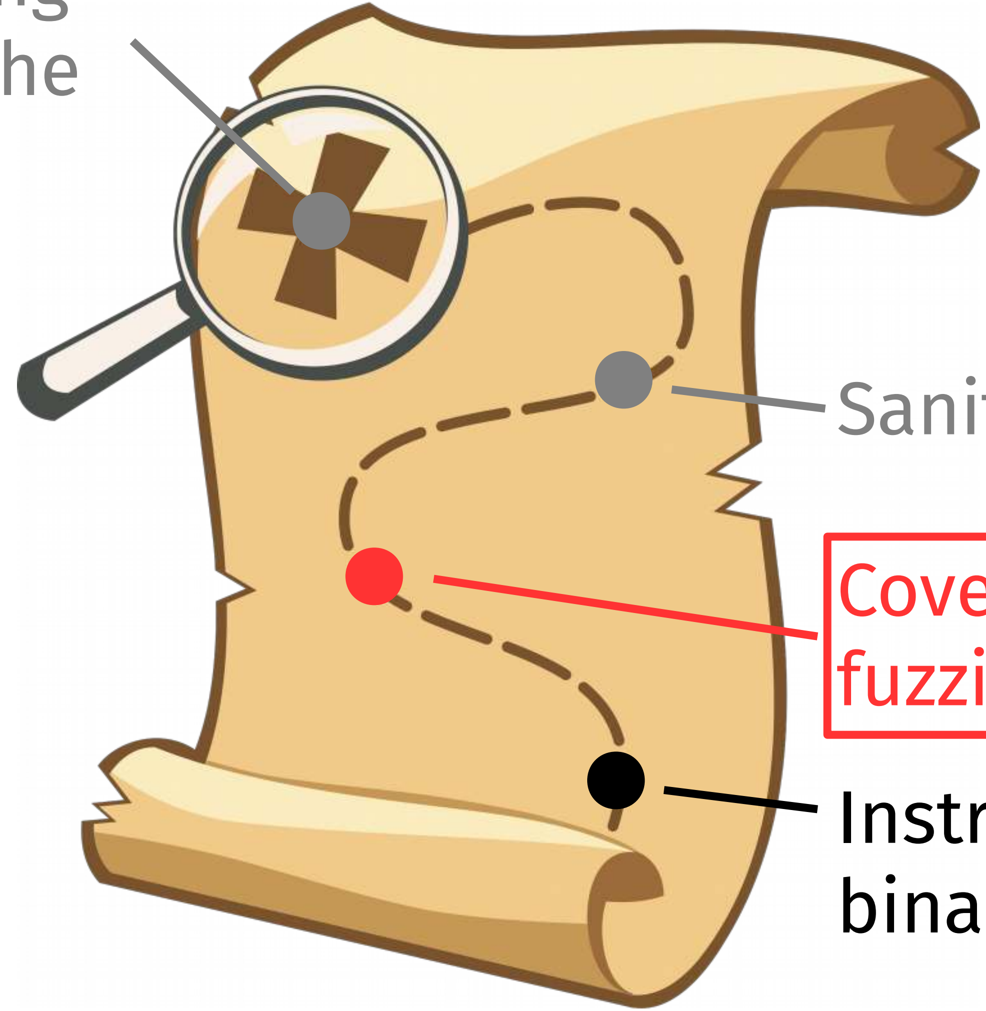
Symbolization

- Symbolization replaces references with assembler labels

1) Relative jumps/calls
2) PC-relative addresses
3) Data relocations

```
loop1:
   lea rax, [data1]
   <new code>
   call func1
   dec rcx
   jnz loop1
```

Instrumenting binaries in the kernel



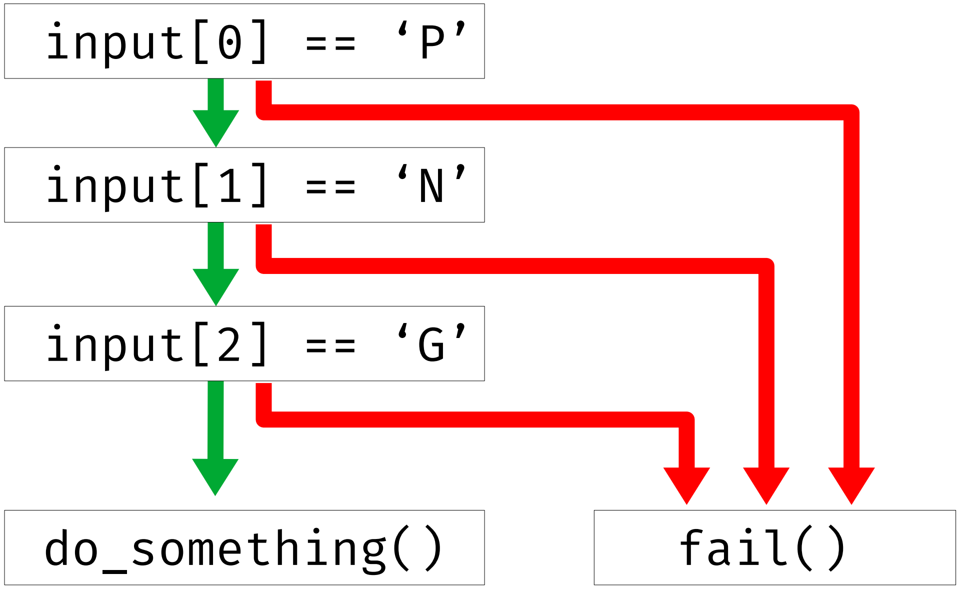
Sanitization

Coverage-guided fuzzing

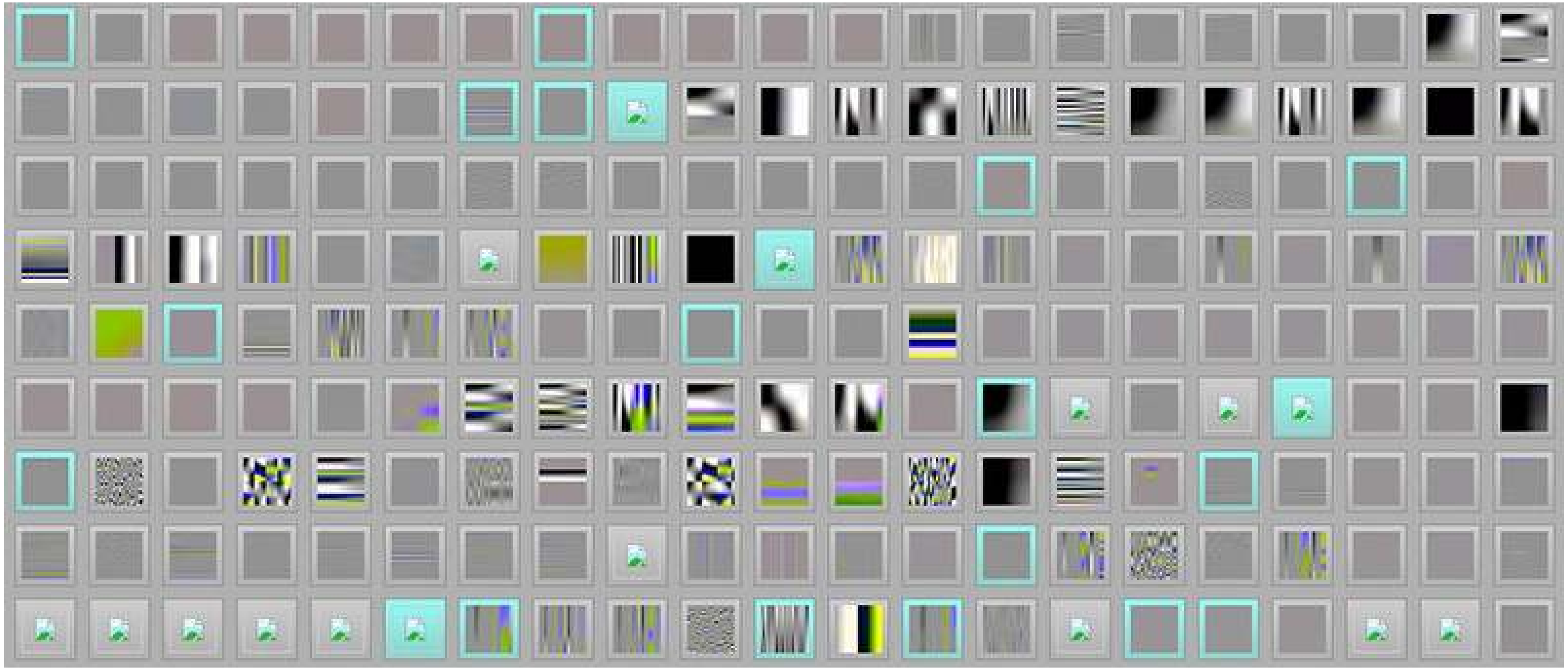
Instrumenting binaries

Coverage-guided fuzzing

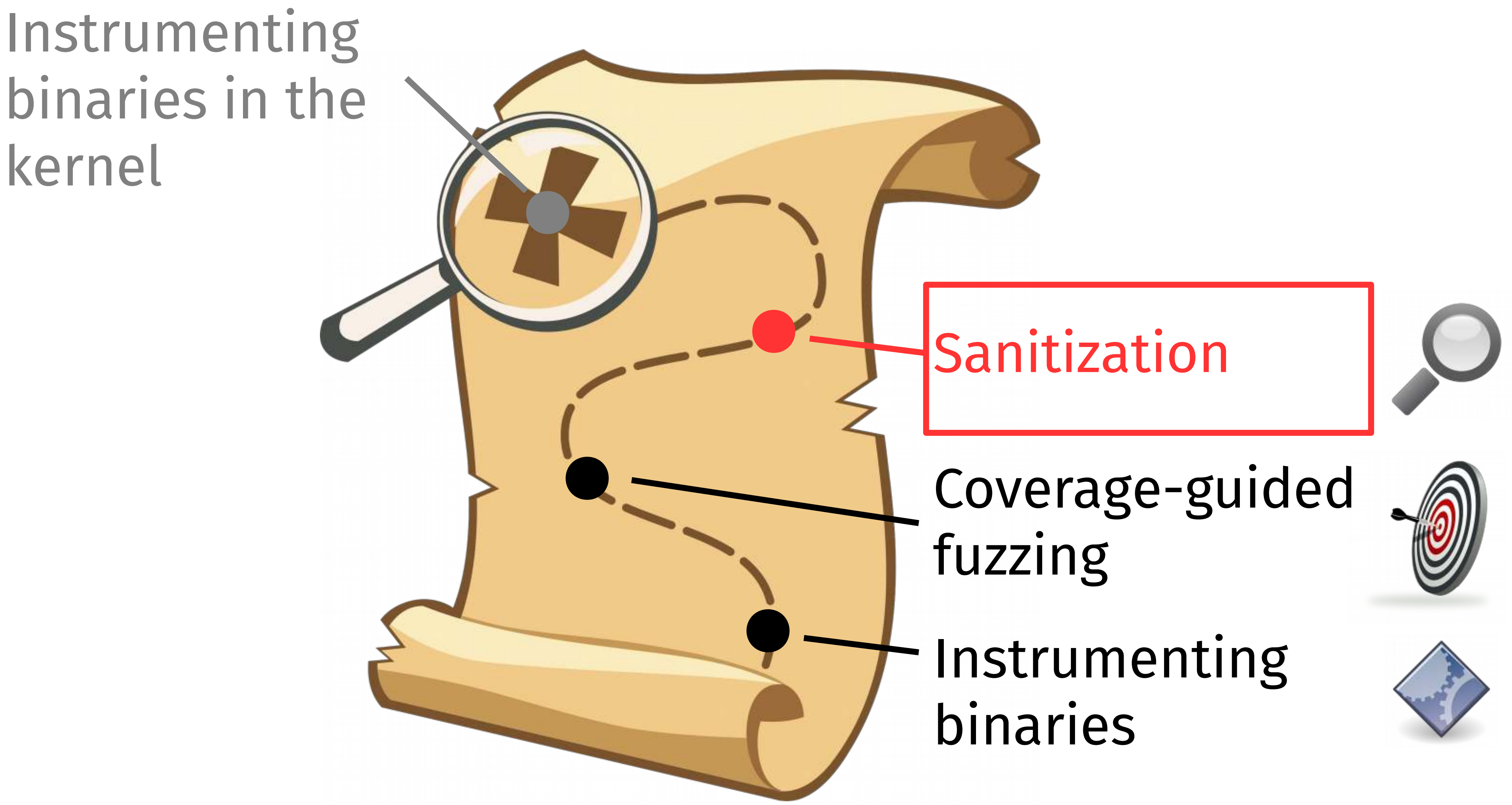
- Record test coverage (e.g. with instrumentation)
- Inputs that trigger new paths are “interesting”
- Mutate interesting inputs to discover new paths



Coverage-guided fuzzing



https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html

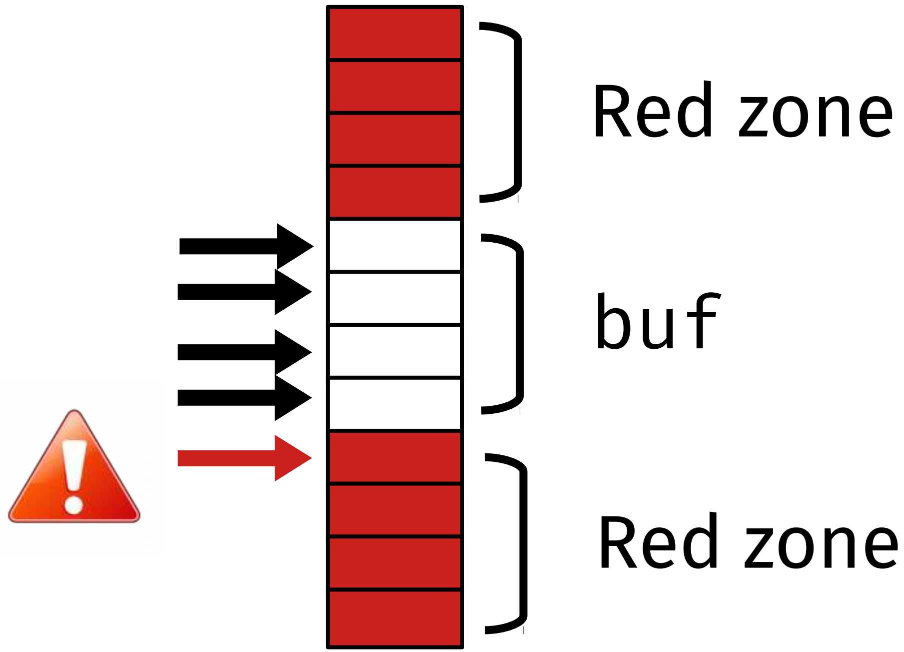

Instrumenting binaries in the kernel
Sanitization
Coverage-guided fuzzing
Instrumenting binaries

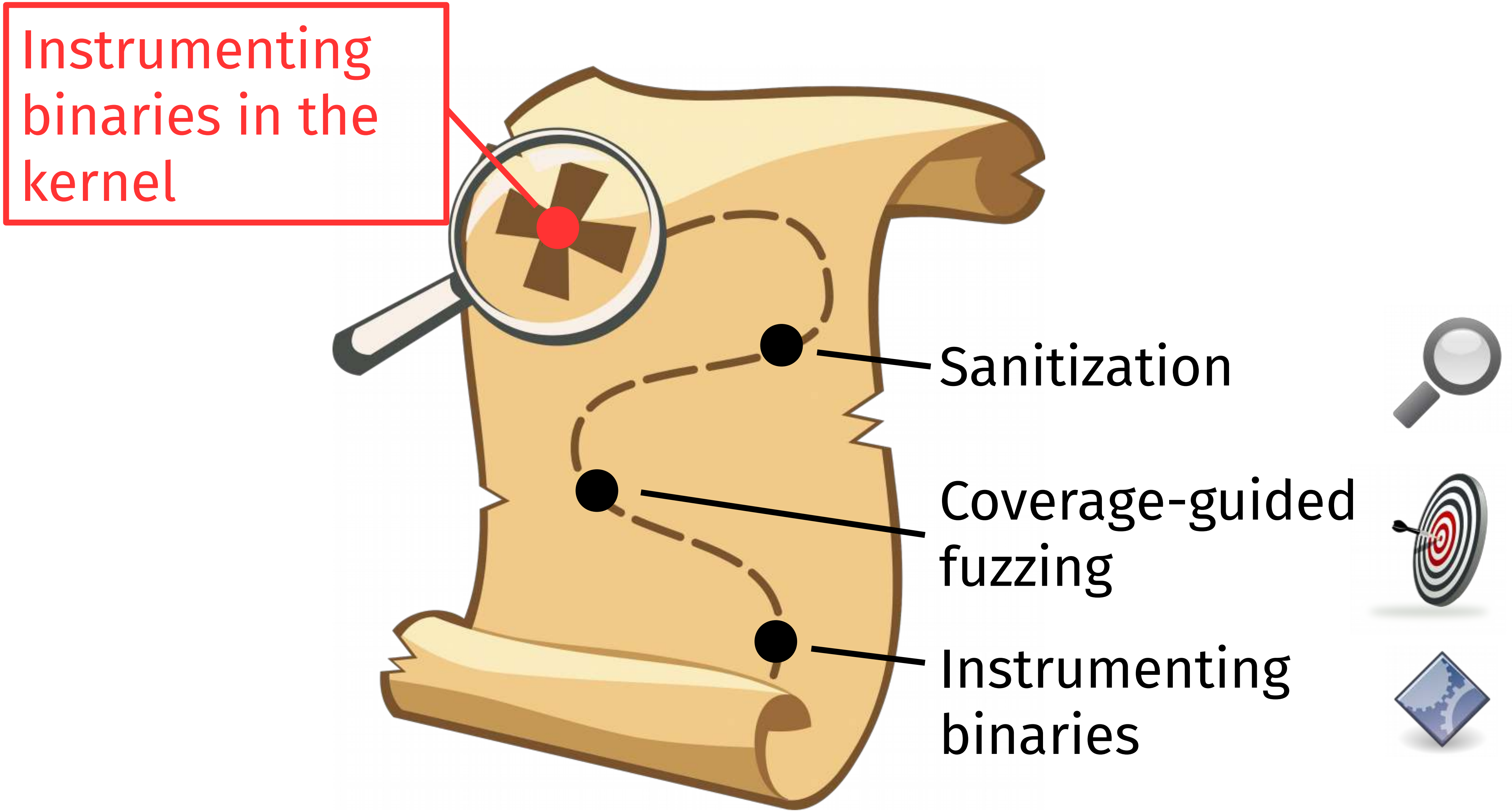
Address Sanitizer (ASan)

- Instrumentation catches memory corruption at runtime
  - Arguably most dangerous class of bugs
- Very popular sanitizer
  - Thousands of bugs in Chrome and Linux
- About 2x slowdown

ASan red zones

```
char buf[4];

strcpy(buf, “AAAAA”);
```




Instrumenting binaries in the kernel
Sanitization
Coverage-guided fuzzing
Instrumenting binaries

RetroWrite instrumentation

- Coverage tracking: instrument basic block starts

- Binary ASan: instrument all memory accesses, link with libASan

Kernel vs. userspace fuzzing

	Crash handling	Tooling	Determinism
Userspace	OS handles crashes gracefully	Easy to use and widely available	Single-threaded code usually deterministic
Kernel	Need VM to keep the system stable	More complex setup, fewer tools	Interrupts, many concurrent threads

Kernel binary fuzzing

- Approach 0: black box fuzzing
- Approach 1: dynamic translation
  - ***Slow! (10x +)***
  - No sanitization like ASan
- Approach 2: Intel Processor Trace (or similar)
  - Requires hardware support
  - Still no sanitization
- Approach 3: static rewriting

kRetroWrite

- Apply RetroWrite to the kernel
- Implemented so far: support for Linux modules
- Demonstrates that RetroWrite applies to the kernel

kRetroWrite

- Kernel modules are always position-independent

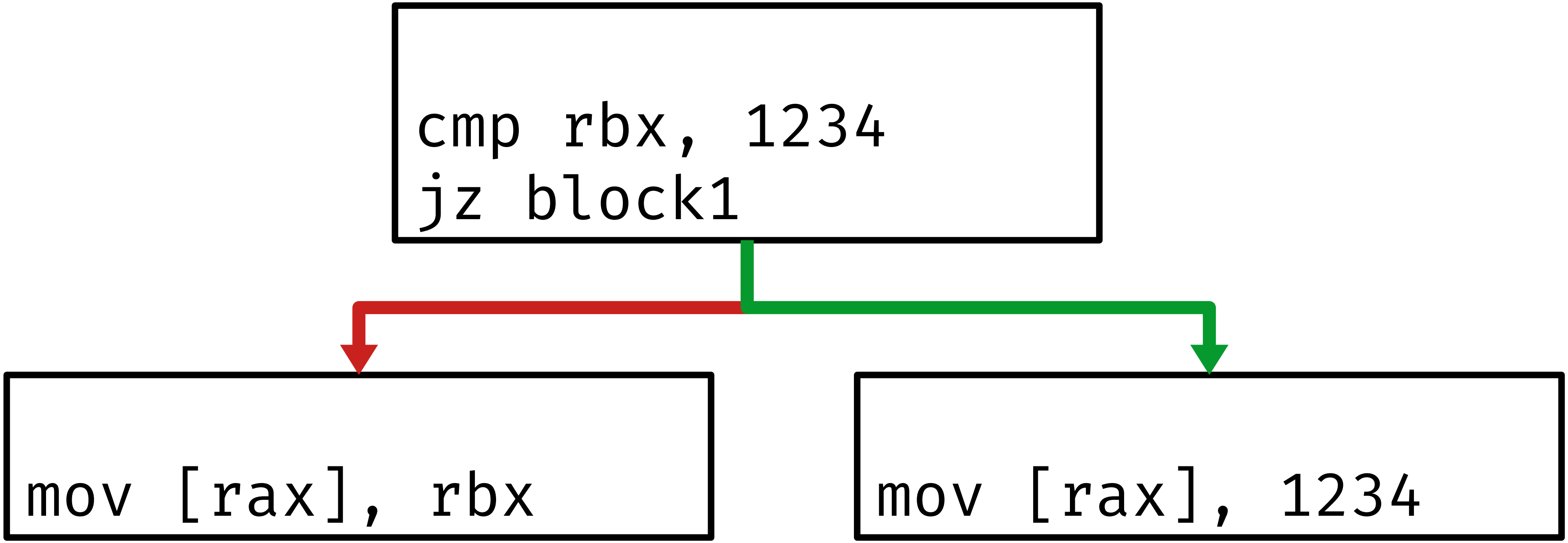
- Linux modules are ELF files
  - Reuse RetroWrite's symbolizer

- Implemented code coverage and binary ASan

kRetroWrite coverage

- Idea: use kCov infrastructure
  - Can interoperate with source-based kCov
- Call coverage collector at the start of each basic block
- Integrates with, e.g., syzkaller, or debugfs

kRetroWrite coverage



kRetroWrite coverage

```
call trace_pc
cmp rbx, 1234
jz block1
```

```
call trace_pc
mov [rax], rbx
```

```
call trace_pc
mov [rax], 1234
```

kRetroWrite binary ASan

- In userspace: link with libASan
- In kernel: build kernel with KASan (kernel ASan)
- Reuse modified userspace instrumentation pass

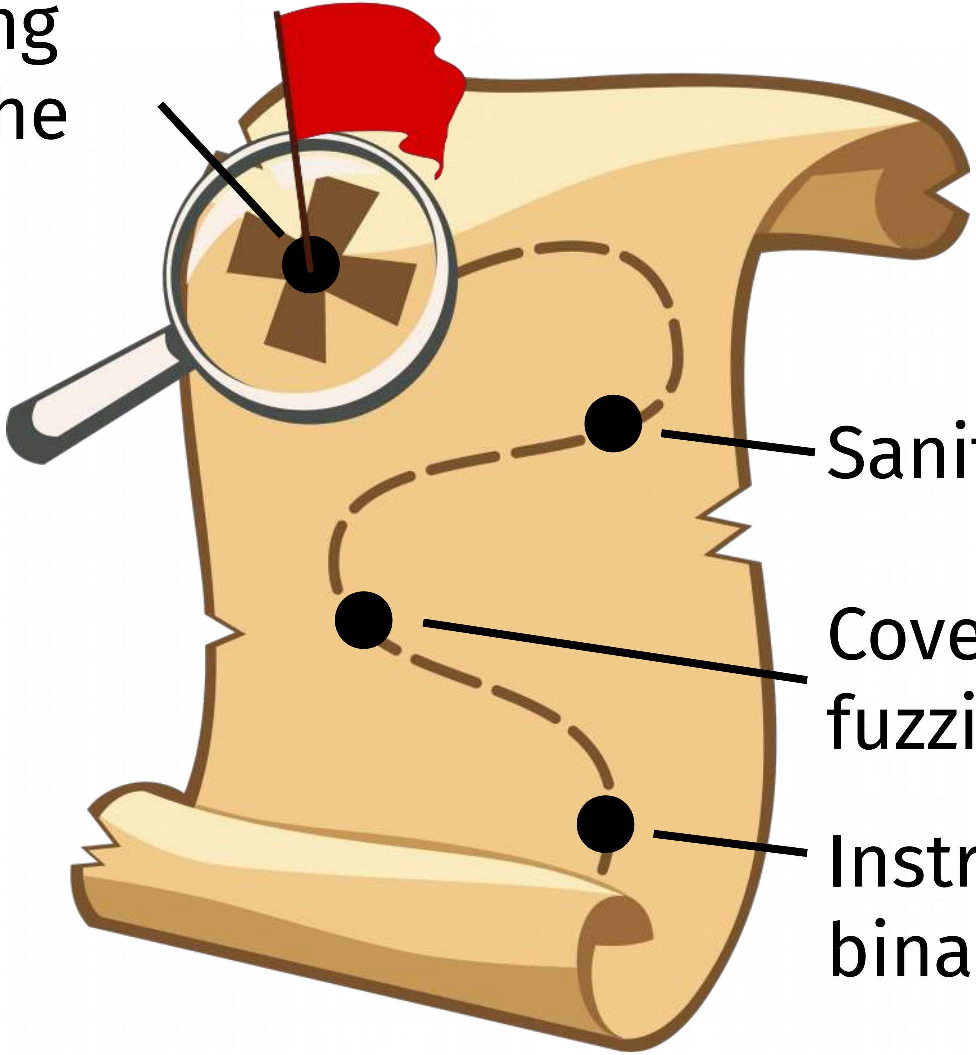
kRetroWrite binary ASan

- Instrument each memory access with a check
- Failed checks print a bug report
- Compatible with source-based kASan

Fuzzing with kRetroWrite

- Rewritten modules can be loaded and fuzzed with standard kernel fuzzers
- So far: tested with syzkaller

Instrumenting binaries in the kernel



Sanitization

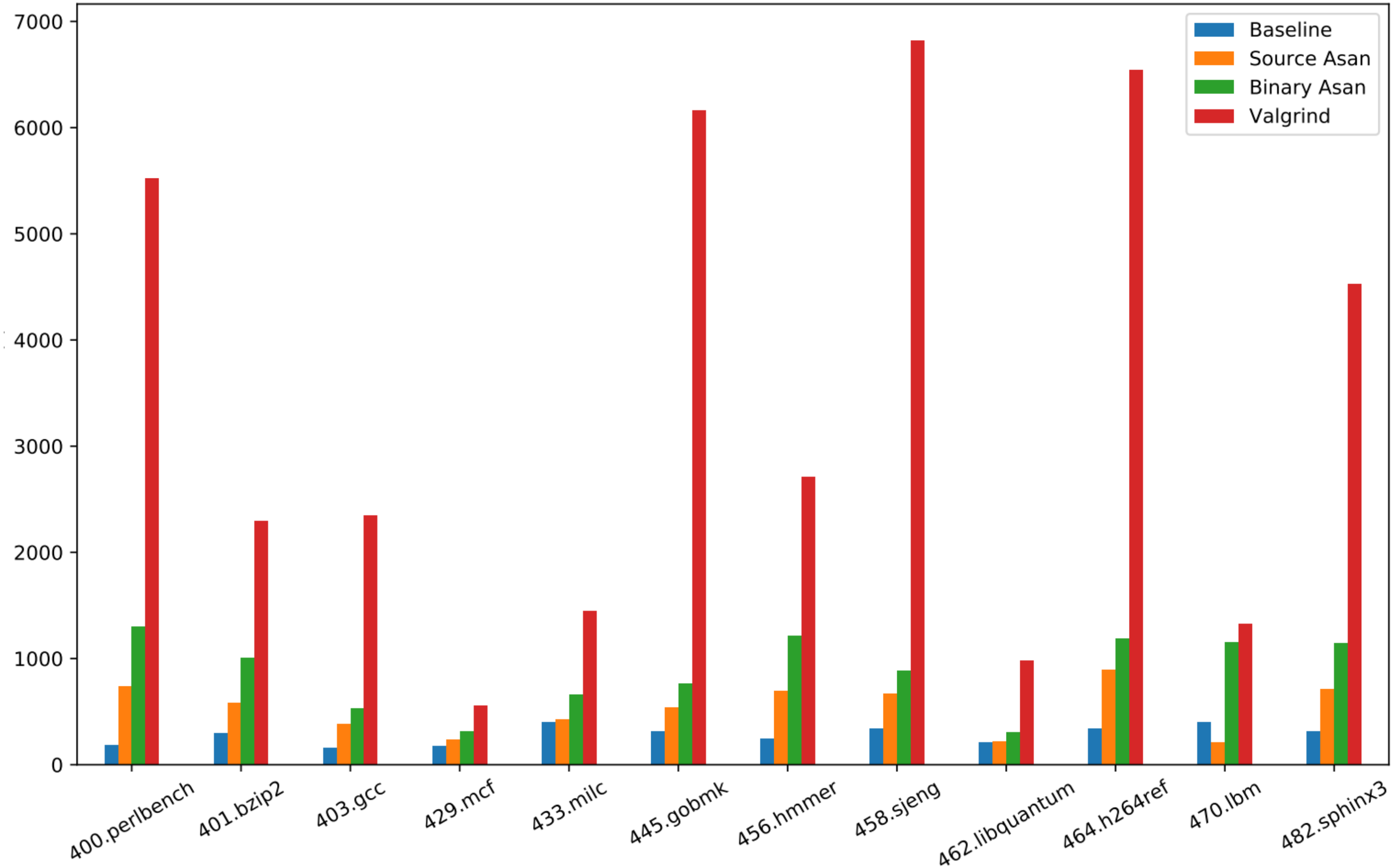
Coverage-guided fuzzing

Instrumenting binaries

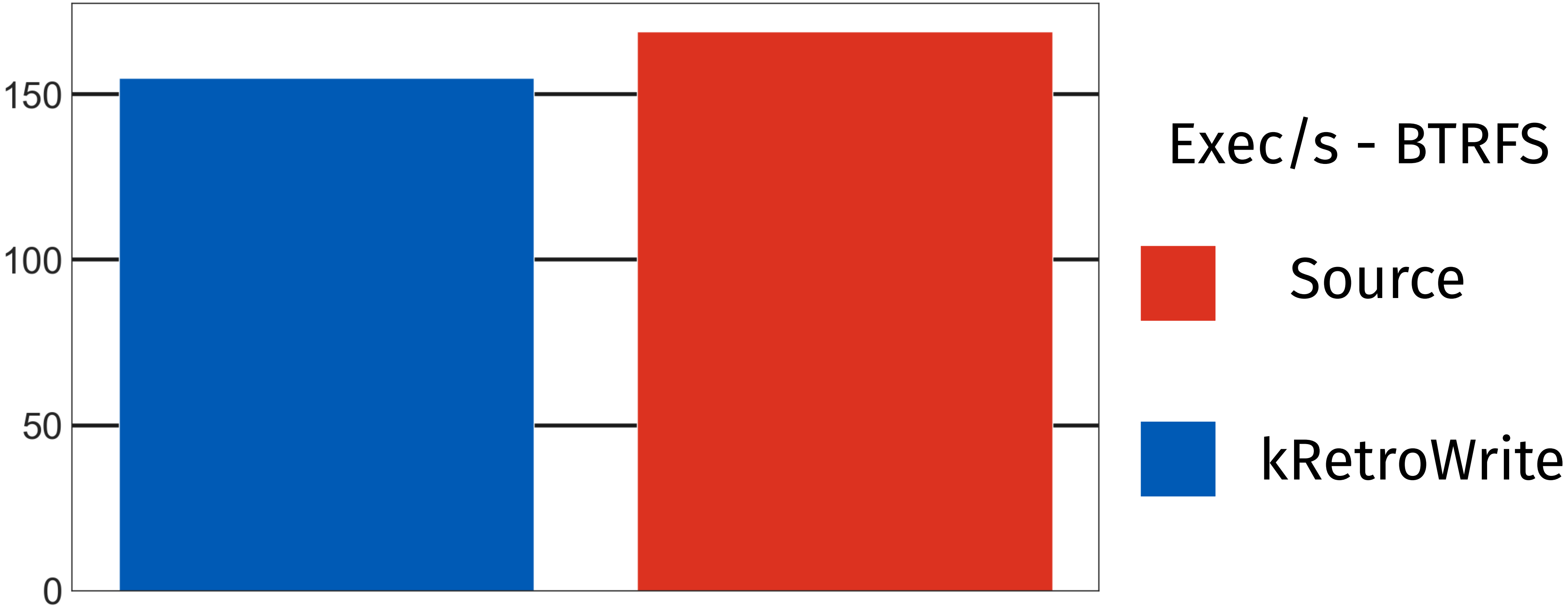
Our experiments

- Userspace: SPEC2006 runtime performance
  - RetroWrite ASan
  - Source ASan
  - Valgrind memcheck
- Kernel: fuzz filesystems/drivers with syzkaller
  - Source KASan + kCov
  - kRetroWrite KASan + kCov

Results - Userspace



Preliminary results - kernel



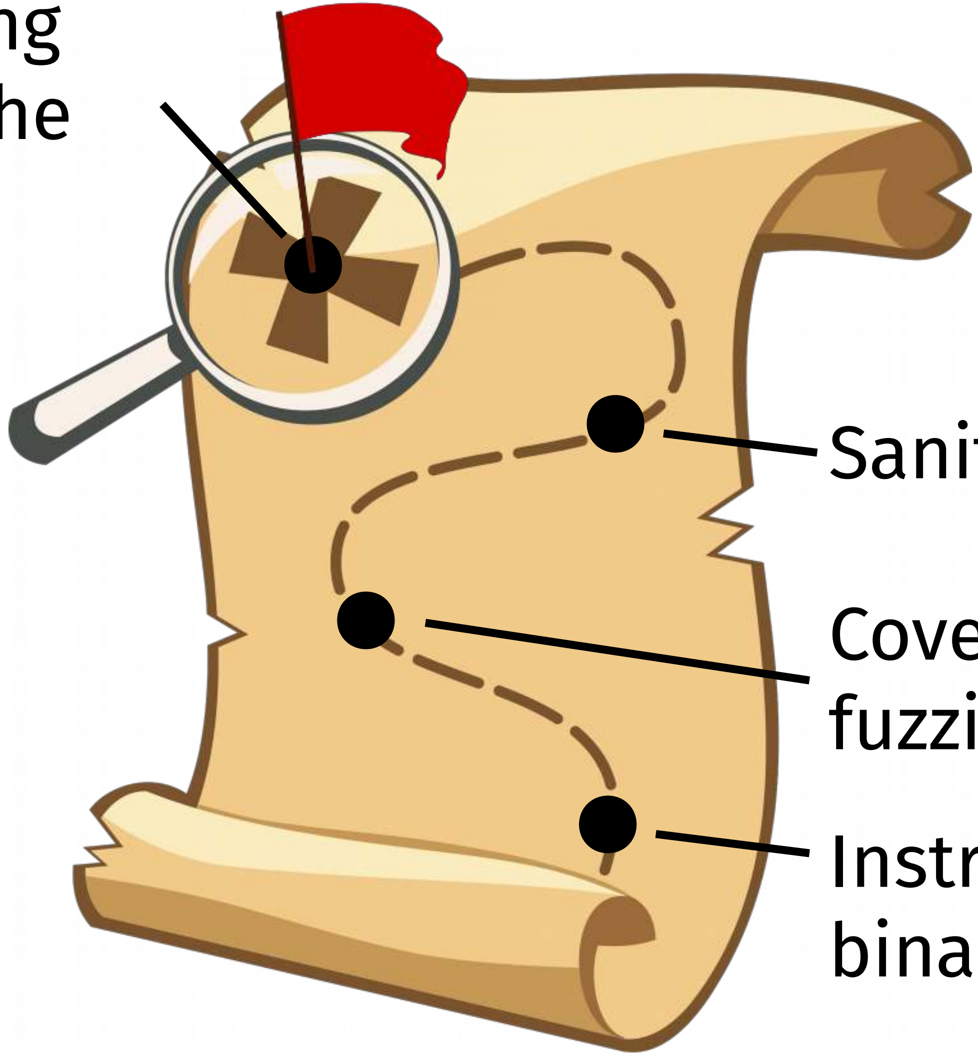
Demo

```
==================================================================
BUG: KASAN: slab-out-of-bounds in
Write of size 2 at addr ffff888000100800 by task syz-executor.7/12792

CPU: 1 PID: 12792 Comm: syz-executor.7 Not tainted 5.5.0-rc1 #3
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.12.0-1 04/01/2014
Call Trace:

 do_syscall_64+0x9c/0x390 arch/x86/entry/common.c:294
 entry_SYSCALL_64_after_hwframe+0x44/0xa9
RIP:
Code:
RSP:
RAX:
RDX:
RBP:
R10:
R13:

Allocated by task 3600:

 do_syscall_64+0x9c/0x390 arch/x86/entry/common.c:294
 entry_SYSCALL_64_after_hwframe+0x44/0xa9

Freed by task 3418:

 do_syscall_64+0x2bb/0x390 arch/x86/entry/common.c:304
 entry_SYSCALL_64_after_hwframe+0x44/0xa9

The buggy address belongs to the object at ffff888000100000
 which belongs to the cache kmalloc-2k of size 2048
The buggy address is located 0 bytes to the right of
```

```
==================================================================
BUG: KASAN: slab-out-of-bounds in
Read of size 2 at addr ffff8880001070c0 by task syz-executor.2/11419

CPU: 0 PID: 11419 Comm: syz-executor.2 Not tainted 5.5.0-rc1 #3
Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.12.0-1 04/01/2014
Call Trace:

 do_syscall_64+0x9c/0x390 arch/x86/entry/common.c:294
 entry_SYSCALL_64_after_hwframe+0x44/0xa9
RIP:
Code:
RSP:
RAX:
RDX:
RBP:
R10:
R13:

Allocated by task 1484:

 do_syscall_64+0x9c/0x390 arch/x86/entry/common.c:294
 entry_SYSCALL_64_after_hwframe+0x44/0xa9

Freed by task 1088:
```

```
==================================================================
r-free in
lr ffff888000100000 by task syz-executor.7/21927

nm: syz-executor.7 Not tainted 5.5.0-rc1 #3
Standard PC (i440FX + PIIX, 1996), BIOS 1.12.0-1 04/01/2014

0x390 arch/x86/entry/common.c:294
ter_hwframe+0x44/0xa9

longs to the page:
) refcount:0 mapcount:-128 mapping:0000000000000000 index:0x0
) ffff88807f887300 ffff88807f887300 0000000000000000
) 0000000000000008 00000000ffffff7f 0000000000000000
 kasan: bad access detected

the buggy address:
) 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
) 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
f ff ff ff ff ff ff ff ff ff ff ff ff ff ff

f ff ff ff ff ff ff ff ff ff ff ff ff ff ff
f ff ff ff ff ff ff ff ff ff ff ff ff ff ff
==================================================================
```

Instrumenting binaries in the kernel



Sanitization

Coverage-guided fuzzing

Instrumenting binaries

Conclusions

- Instrument real-world binaries for fuzzing
  - Coverage tracking for fast fuzzing
  - Memory checking to detect bugs
- Static rewriting at zero instrumentation cost
  - Limited to position independent code
  - Symbolize without heuristics
- More? https://github.com/HexHive/retrowrite
  - User-space now, kernel in ~2-3 weeks