FOSDEM 2020

SaBRE
Load-time selective binary rewriting

Paul-Antoine Arras, Anastasios Andronidis, Luís Pina, Karolis Mituzas,
Qianyi Shu, Daniel Grumberg, Cristian Cadar

Software Reliability Group, Imperial College London

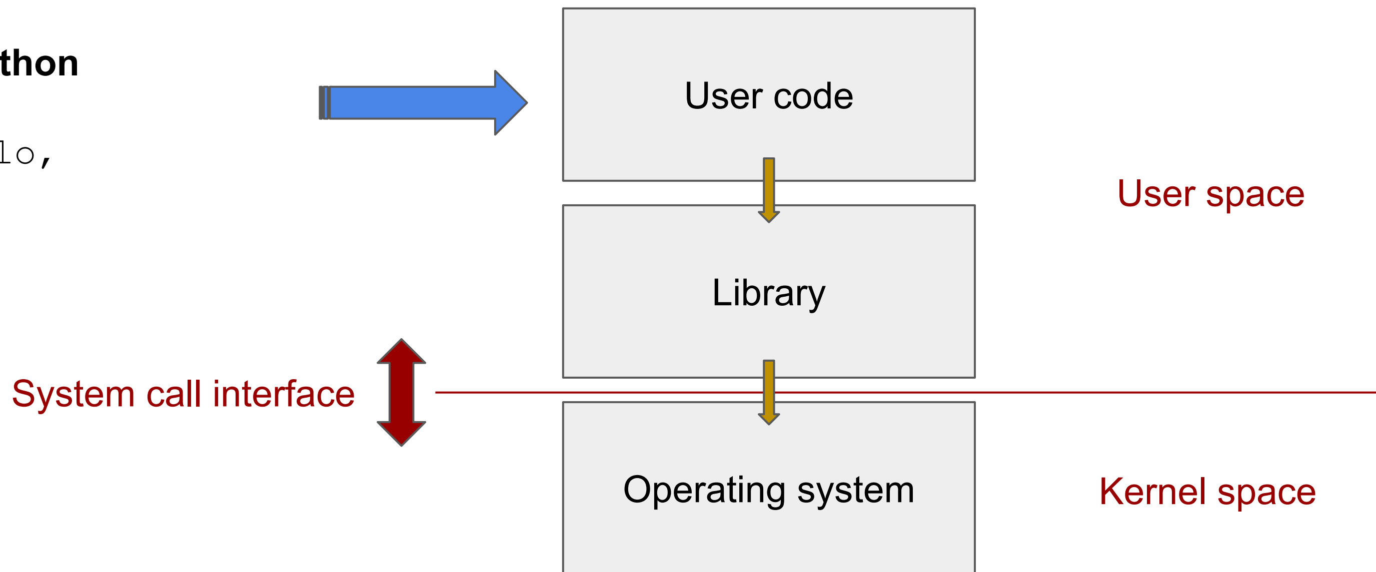
How resilient is my software?

- Assess fault tolerance
- E.g. disk full, memory exhausted
- Hard to reproduce on real system
- Can we simulate a fault?
- Yes, but...
- Kernel hacking is dangerous
- Tinkering with libraries can also be painful
- What’s in between?

Hello world

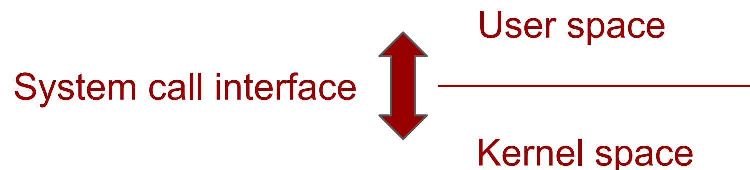
Python

```
print('Hello,
world!')
```



System call interface

- Set of low-level operations
- Request a service
- Very similar to function call
    - Several arguments
    - One result



Python

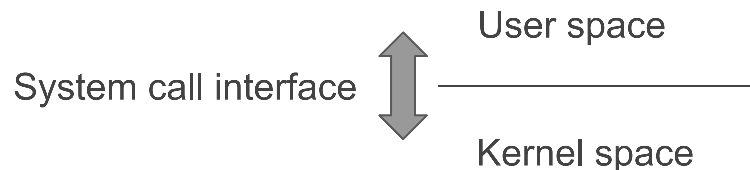
```
print('Hello, world!')
```

System call (C syntax)

```
write(1, "Hello, world!", 13)
```

System call errors

- Return value
  - ≥ 0 → success
  - < 0 → failure
- `write`
  - Size written
  - E.g. permission denied (EPERM), disk full (ENOSPC)



Python

```
file = open("/tmp/hello", "w")
file.write("Hello!")
```

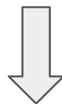
System call

```
open("/tmp/hello", "w") = 8
write(8, "Hello!", 6) = 6
write(8, "Hello!", 6) = EPERM
                        < 0
```

Fault injection

- How to simulate e.g. a permission error at system call level?
- Swap return value with error code

```
write(8, “Hello!”, 6) =    6
```



```
write(8, “Hello!”, 6) =
EPERM
```

How to achieve that?

Binary rewriting

What is binary rewriting?

- Modify program at machine code level
- No source code needed
- Does not require recompilation
- Only requirement: *disassembling*
    - Break program into sequence of instructions
    - Assume done for now

0 1 0 0 1 0 0 1 0 0 1

Binary

```
push R0
load 0x14,R0
call fnct
or 0x67,R2
```

Disassembly

What is binary rewriting?

- Modify program at machine code level
- No source code needed
- Does not require recompilation
- Only requirement: *disassembling*
    - Break program into sequence of instructions
    - Assume done for now

0 1 0 0 1 0 0 1 0 0 1

Binary

```
push R0
load 0x14,R0
call fnct
or 0x67,R2
```

Disassembly

Disassembly

Offset	Size in bytes	Human-readable instruction (mnemonic + operands)
0x00	1	push R0
0x01	2	load 0x14,R0
0x03	5	call fnct
0x08	3	or 0x67,R2
0x0b	2	jump L1
0x0d	3	and 0x45,R2
0x10	5	jump L2
…		

Operations on instructions

- Remove
- Replace
- Add

Remove

Pad with `nop`s

```
0x00    1    push R0
0x01    2    load 0x14,R0
0x03    5    call fnct
0x08    3    or 0x67,R2
```

```
0x00    1    push R0
0x01    1    nop
0x02    1    nop
0x03    5    call fnct
0x08    3    or 0x67,R2
```

Replace

Call → jump

```
0x00      1    push R0
0x01      2    load 0x14,R0
0x03      5    call fnct
0x08      3    or 0x67,R2
```

```
0x00      1    push R0
0x01      2    load 0x14,R0
0x03      ?    jump
0x??      3    or 0x67,R2
```

Size matters

- Shifting instructions is impractical
    - Jumps become invalid
    - Addresses have to be recomputed
- Do rewritten instructions fit?
- Compare instruction sizes
    - Original S(O)
    - Rewritten S(R)

Replace

Call → jump

```
0x00      1     push R0
0x01      2     load 0x14,R0
0x03      5     call fnct
0x08      3     or 0x67,R2
```

```
0x00      1     push R0
0x01      2     load 0x14,R0
0x03      ?     jump
0x??      3     or 0x67,R2
```

S(O) = 5

S(R) = ?

Replace

Call → jump

```
0x00    1    push R0
0x01    2    load 0x14,R0
0x03    5    call fnct
0x08    3    or 0x67,R2
```

```
0x00    1    push R0
0x01    2    load 0x14,R0
0x03    5    jump
0x08    3    or 0x67,R2
```

S(O) = S(R)

Replace

Call → jump

```
0x00    1    push R0
0x01    2    load 0x14,R0
0x03    5    call fnct
0x08    3    or 0x67,R2
```

```
0x00    1    push R0
0x01    2    load 0x14,R0
0x03    3    jump
0x06    1    nop
0x07    1    nop
0x08    3    or 0x67,R2
```

$S(O) \leq S(R)$

Replace depends on relative sizes

- If S(R) = S(O) → just replace
- If S(R) < S(O) → pad with `nop`s
- If S(R) > S(O) → ???

Problem

How to fit larger instructions?

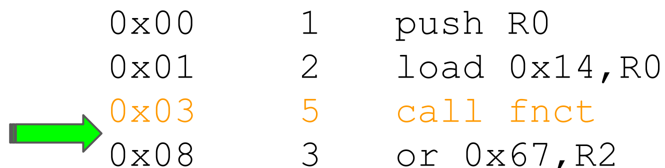
Detour

- Problem: S(R) ≥ S(O)
- Shifting instructions still not an option
- Solution: relocate instructions to out-of-line scratch space

1. Allocate memory
2. Move instructions
3. Add jumps into and out of moved instructions

Add with detour

Insert a jump to rewritten instructions

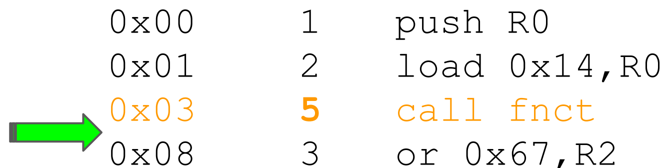


```
0x00     1    push R0
0x01     2    load 0x14,R0
0x03     5    jump D0
L0:
0x08     3    or 0x67,R2
…
D0:
0xffec   5    call fnct
…     // added instructions
0xfffd   5    jump L0
```

Out-of-line scratch space (D0 … jump L0)

Add with detour

Insert a jump to rewritten instructions



```
0x00     1    push R0
0x01     2    load 0x14,R0
0x03     5    jump D0
L0:
0x08     3    or 0x67,R2
…
D0:
0xffec   5    call fnct
…    // added instructions
0xfffd   5    jump L0
```

Out-of-line scratch space

S(O) = S(J)

Replace with detour

- If S(J) ≤ S(O) → replace and pad with `nop`s
- Otherwise, relocate neighbouring instructions

```
0x00      1    push R0
0x01      2    load 0x14,R0
0x03      5    call fnct
0x08      3    or 0x67,R2
```

```
0x00      1    push R0
0x01      5    jump D0
0x06      1    nop
0x07      1    nop
L0:
0x08      3    or 0x67,R2
…
D0:
…     // substitute instructions
0xfff8    5    call fnct
0xfffd    5    jump L0
```

Out-of-line scratch space

S(J) = 5
S(O) = 2

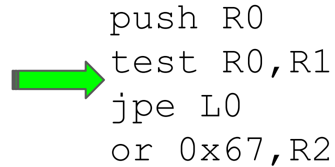
Replace depends on relative sizes

- S(R) = S(O) → replace O with R
- S(R) < S(O) → replace and pad with `nop`s
- S(R) > S(O) → detour with jump (J)
    - S(J) = S(O) → replace O with J
    - S(J) < S(O) → replace and pad with `nop`s
    - S(J) > S(O) → replace and **relocate surrounding instructions**

Can instructions always be relocated?

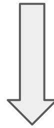
Side effects

Alter status flags



```
push R0
test R0,R1
jpe L0
or 0x67,R2
```

set parity flag
jump if parity even

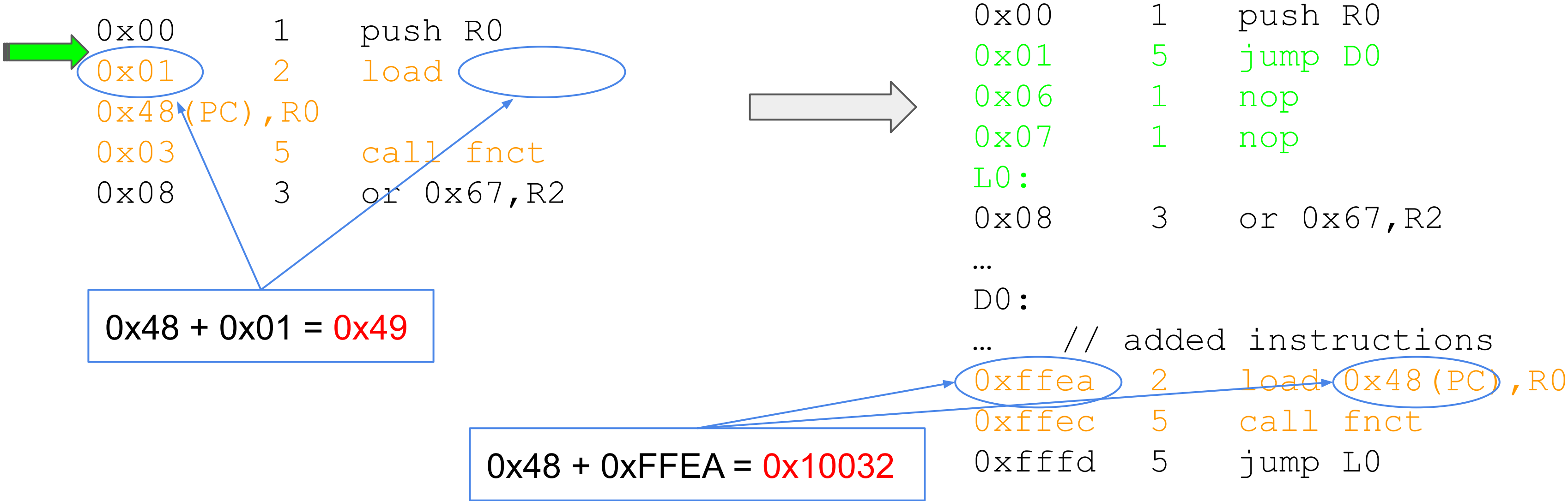


```
push R0
test R0,R1
add R2,R3
jpe L0
or 0x67,R2
```

Solution

Whitelist of instructions known to be safe to relocate

PC-relative addressing



PC-relative addressing

Solution

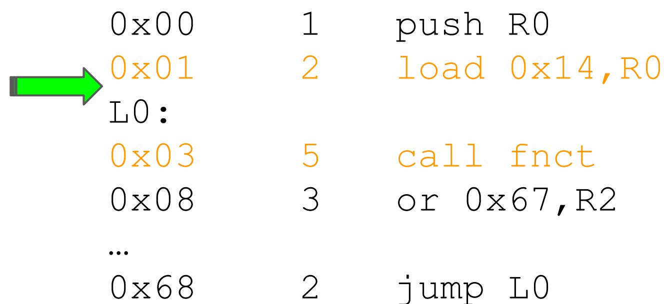
Fixup displacement
in relocated instruction

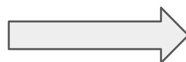
```
0x00       1    push R0
0x01       2    load
0x48(PC),R0
0x03       5    call fnct
0x08       3    or 0x67,R2
```

0x49 - 0xFFE6 = -0xFF9D

```
0x00       1    push R0
0x01       5    jump D0
0x06       1    nop
0x07       1    nop
L0:
0x08       3    or 0x67,R2
…
D0:
…     // added instructions
0xffe6     6    load -0xff9d(PC),R0
0xffec     5    call fnct
0xfffd     5    jump L0
```

Branch target





```
0x00      1    push R0
0x01      5    jump D0
0x06      1    nop
0x07      1    nop
L1:
0x08      3    or 0x67,R2
…
0x68      2    jump L0
…
D0:
0xffec    2    load 0x14,R0
…    // added instructions
0xfff8    5    call fnct
0xfffd    5    jump L1
```

Solution

Record branch target locations before rewriting

Problematic instructions

- Branch targets → do not rewrite
- PC-relative addressing → fixup displacement
- Side effects → only rewrite white-listed instructions

What if not enough instructions can be relocated?

Cannot relocate instructions

- Cannot accommodate jump
- Detour cannot be used
- Instead, insert short illegal instruction
- Setup signal handler to catch SIGILL
- Put added instructions into handler
- Significant overhead but extremely rare

Disassembling

What is disassembling?

Break binary code into sequence of instructions

0 1 0 0 1 0 0 1 0 0 1

Binary

Disassembling

```
push R0
load 0x14,R0
call fnct
or 0x67,R2
```

Disassembly

0 1 | 0 0 1 | 0 0 1 0 | 0 1

Binary

Disassembling

```
push R0
load 0x14,R0
call fnct
or 0x67,R2
```

Disassembly

Disassembler types

- Dynamic
  - Actually run program
  - Decode instructions just in time
  - Runtime penalty
  - E.g. Dyninst, DynamoRIO, Pin
- Static
  - Program is not run
  - Binary scanned according to algorithm
  - No runtime penalty
  - E.g. Multiverse

Static disassembler

Linear Sweep

```
0x00    1    push R0
0x01    2    load 0x14,R0
0x03    5    call fnct
0x08    3    or 0x67,R2
0x0b    2    jump L1
0x0d    3    and 0x45,R2
0x10    5    jump L2
0x15    ?    bad // garbage
```

Recursive Traversal

```
0x00    1    push R0
0x01    2    load 0x14,R0
0x03    5    call fnct
0x08    3    or 0x67,R2
0x0b    2    jump L1
…    // skipped instructions

L1:
0x6c    4    move R0,R1
```

Disassembly challenges

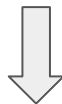
- Code discovery or content classification problem
  - Mixed code and data
  - Halting problem
- Instruction overlapping
  - Variable-length ISA
  - One byte encodes several instructions
  - Obfuscation technique

SaBRe:
Load-time selective binary rewriting for system calls and function prologues

Fault injection

- How to simulate e.g. a permission error at system call level?
- Swap return value with error code

```
write(8, “Hello!”, 6) =    6
```



```
write(8, “Hello!”, 6) =
EPERM
```

How to achieve that?

Intercepting system calls

- Problem: syscalls are in libraries, not user programs
- How to ensure all syscalls are intercepted?
- Rewriting on disk ahead of time is impractical
- Rewriting in memory just in time incurs overhead
- Solution: “just ahead of time”
    - At load time
    - Intercept dynamic linker
    - Rewrite libraries when mapped
    - In process memory

Application Programming Interface

- Instrumentation through user-defined plugins
  - E.g. fault injection plugin
  - Plugin implements hook functions
- Hook function for syscalls:

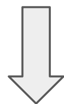
```
long (*sbr_sc_handler_fn)
     (long, long, long, long, long, long, long)
```

  - Receives syscall number and 6 args
  - Responsible for actually issuing syscall
  - May alter parameters and return value

Fault injection

- How to simulate e.g. a permission error at system call level?
- Swap return value with error code

```
write(8, "Hello!", 6) =     6
```



```
write(8, "Hello!", 6) =
EPERM
```

How to achieve that?

Basic fault injection plugin

```
long handle_syscall (long sc_no, long arg1, long arg2, long
arg3, long arg4, long arg5, long arg6) {

    long ret = real_syscall(sc_no, arg1, arg2, arg3, arg4,
arg5, arg6);

    if (sc_no == SYS_WRITE)

        ret = -EPERM;

    return ret;

}
```

Available plugins

- Identity
  - Dummy plugin
  - Passes syscalls on unchanged
  - Testing and benchmarking
- Fault injector
  - Test application resiliency
  - Simulate syscall failures
  - User sets failure probability by syscall category
- Tracer
  - `strace` replacement
  - Does not use ptrace
  - Much more efficient

Supported archs

- x86_64
  - Main dev target
  - Continuous integration
- RISC-V
  - Open ISA
  - Devroom yesterday
  - QEMU Linux support

Current implementation

- Target OS: Linux
- Languages: mainly C + ASM snippets
- No third-party library
- LoC
  - Backbone: 2108
  - x86_64: 1333
    - C: 963
    - ASM: 410
  - API + support code: 2016
- Program size
  - x86_64: 47 KiB
  - RISC-V: 40 KiB

Evaluation

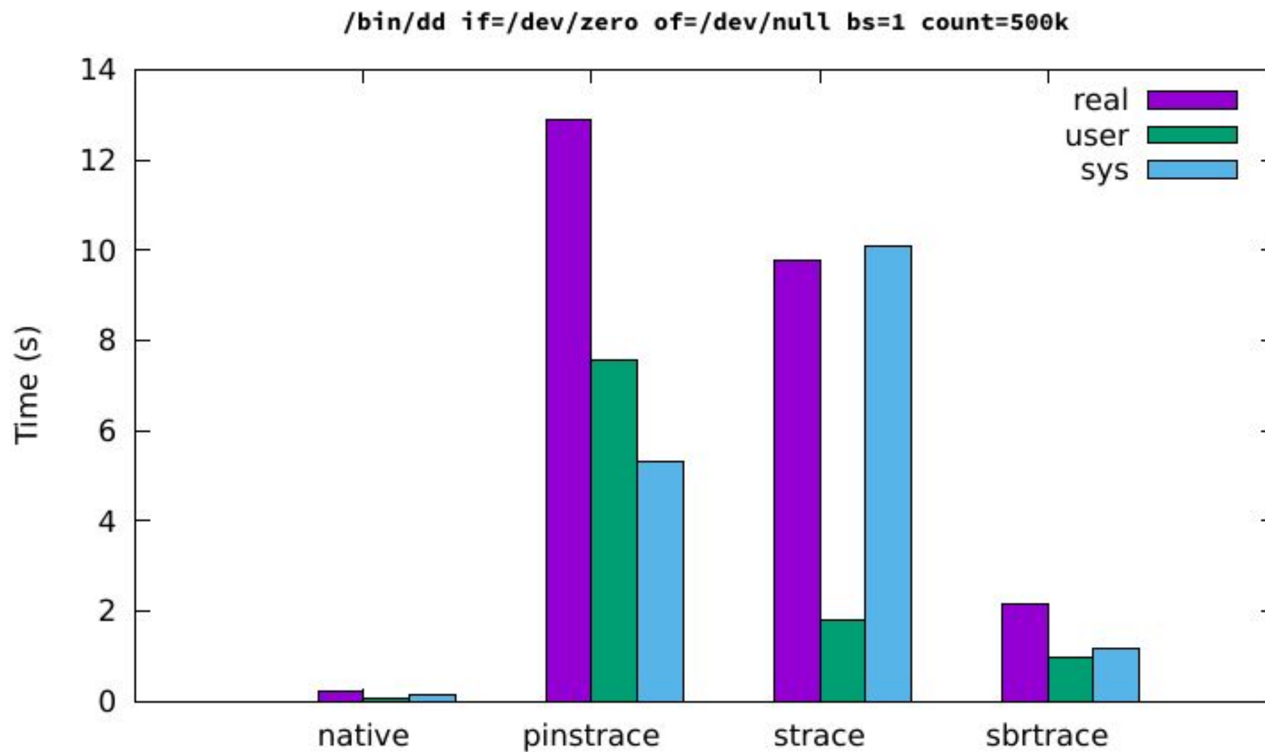
Worst-case overhead

- Experimental setup
  - Plugin: identity
  - Applications: nginx, lighttpd, redis, memcached
  - Glibc 2.28
  - 3-minute execution, millions of requests
  - CPU utilisation: 100%
- Results
  - 404 syscalls detoured
  - **Load-time overhead: 60ms**
  - **Run-time overhead: ≤ 3%**

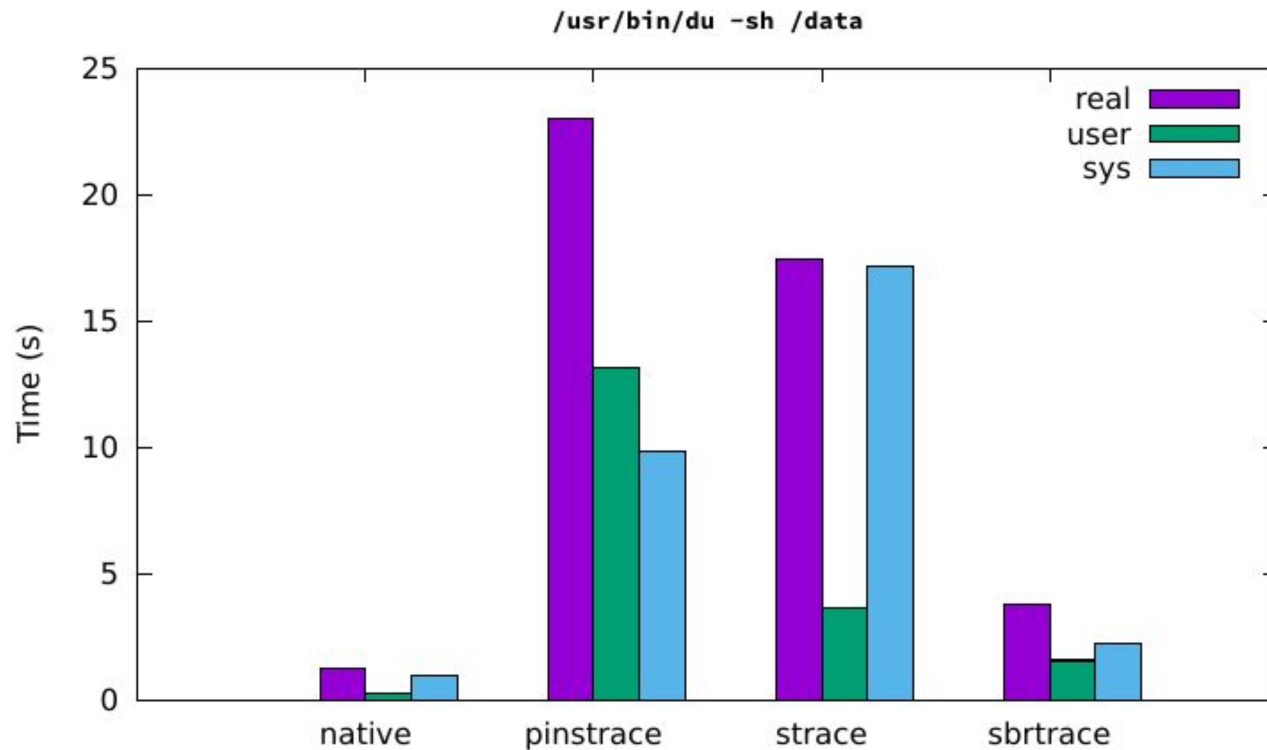
System call tracer

- Plugin mimics `strace`'s output
- Not using `ptrace`
- Compare with `strace` and equivalent Pin plugin
- Issue read and write with `dd`
  - ~1M syscalls issued
- Estimate disk usage of 150GB tree with `du`

`dd` results



`du` results



Fault injector

- Applications: GNU coreutils
- Busybox testsuite
- Bugs found
    - Cryptic error messages
    - Lack of resiliency
    - Crashes

SaBRe in a nutshell

- Selective binary rewriting
  - Syscalls and vDSO
  - Function prologues
  - x86_64-specific: RDTSC
- At load time, in process memory
- Low overhead, suitable for embedded devices
- Simple API to build plugins
- Available plugins
  - Fault injector
  - Tracer
- GPLv3

Availability

- GitHub: https://github.com/srg-imperial/SaBRe
- Licence: GPLv3
  - Plugins also under GPLv3
- PR and bug reports welcome!