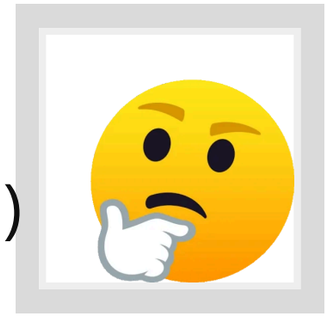
benefits and drawbacks of syscall hooks

Hajime Tazaki, Kenichi Yasukata (iijlab)

Linux netdev conference 0x18 (2024)

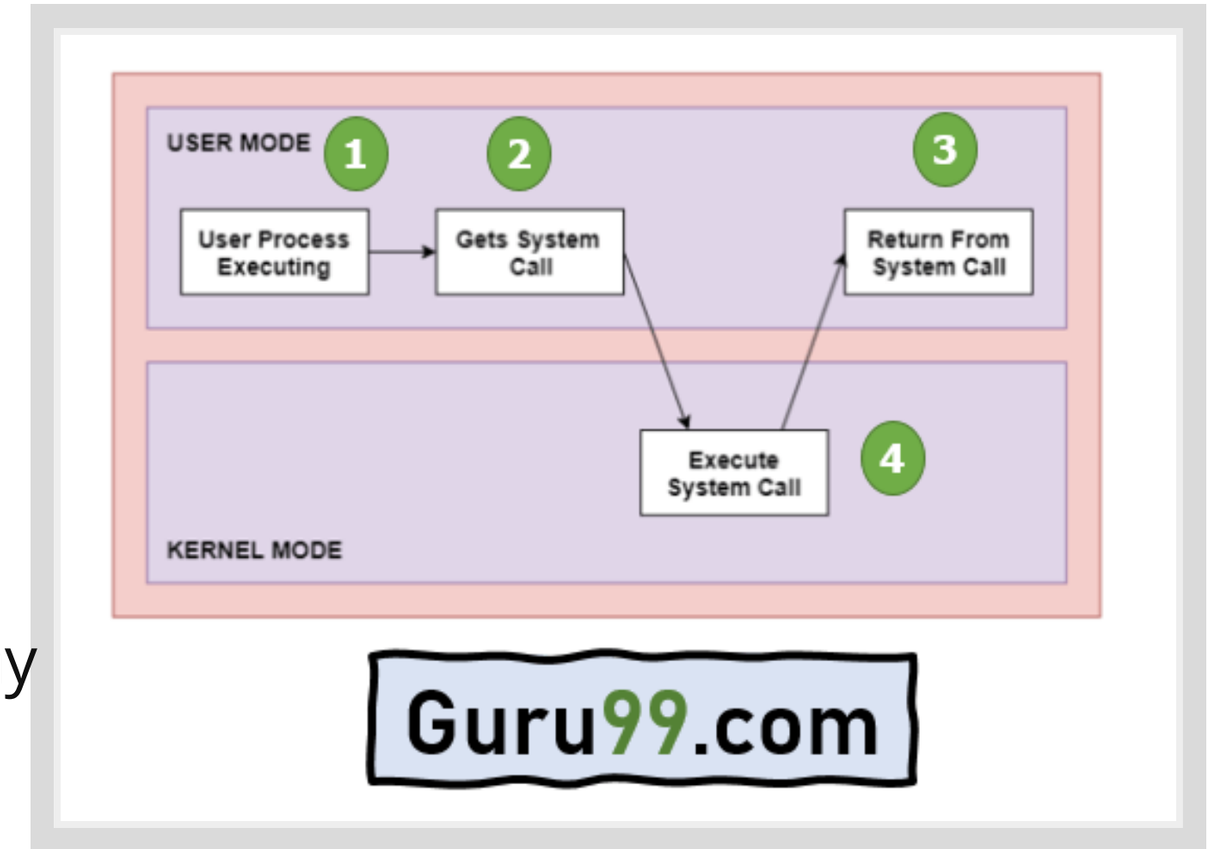
introduction

- what is syscall hook ? (definition)
- why do we need it ? (motivation)
- how is it useful ? (problem/solution)
- when do you use it ? (usecase)



what is syscall and why syscall hook ?

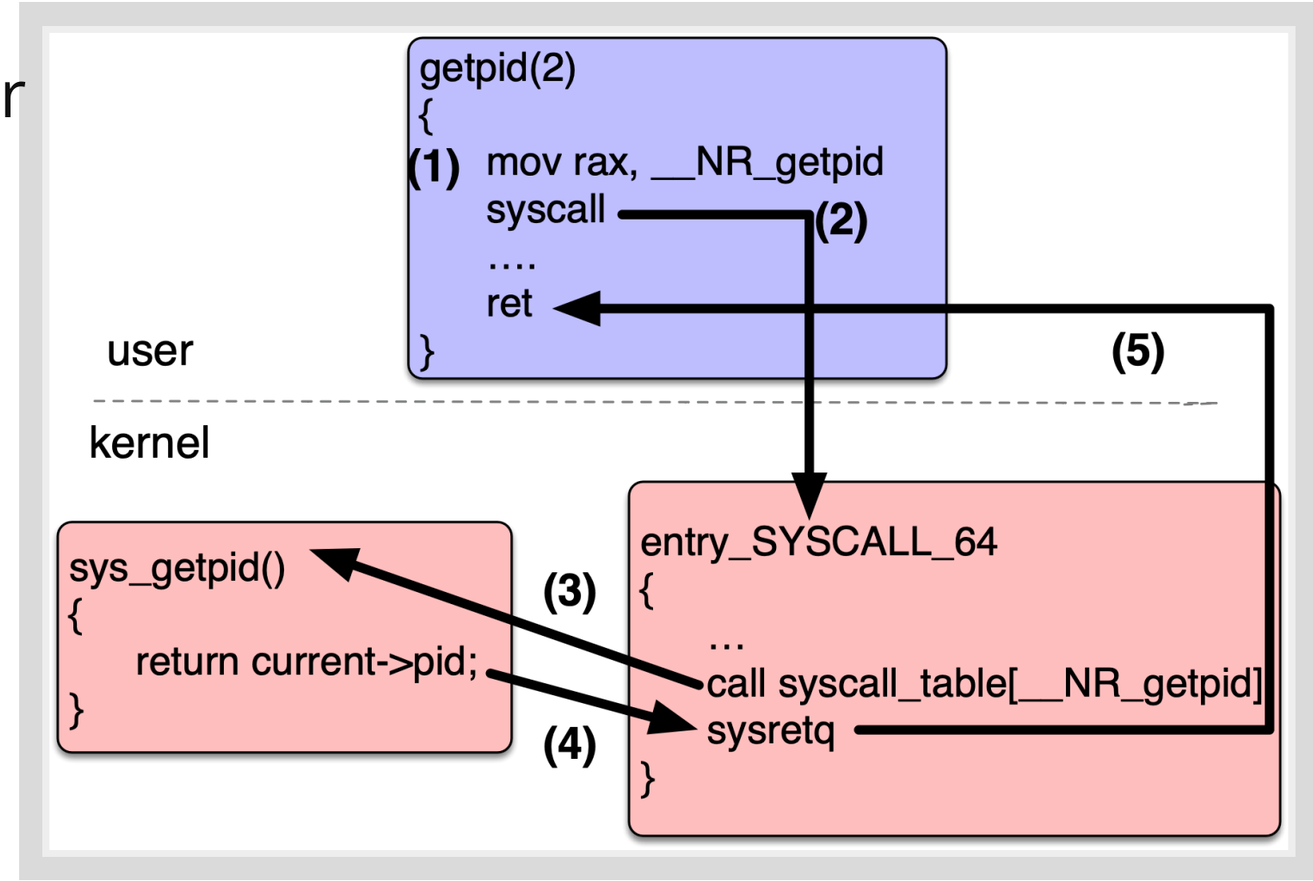
- trace code path on the fly
- use different TCP instead of kernel
- use your own filesystem
- run your programs on different environment
  - **without porting**
- syscall hook (and interposing) is a way
  - instead of kernel updates
  - userspace program updates



https://www.guru99.com/system-call-operating-system.html

recap: how syscall works ?

1. set syscall number to %rax register
2. exec **syscall** instruction
   - jump syscall entry point
3. call a handler in *syscall_table[nr]*
4. the handler return with a value
5. return w/ **sysretq** instruction to caller

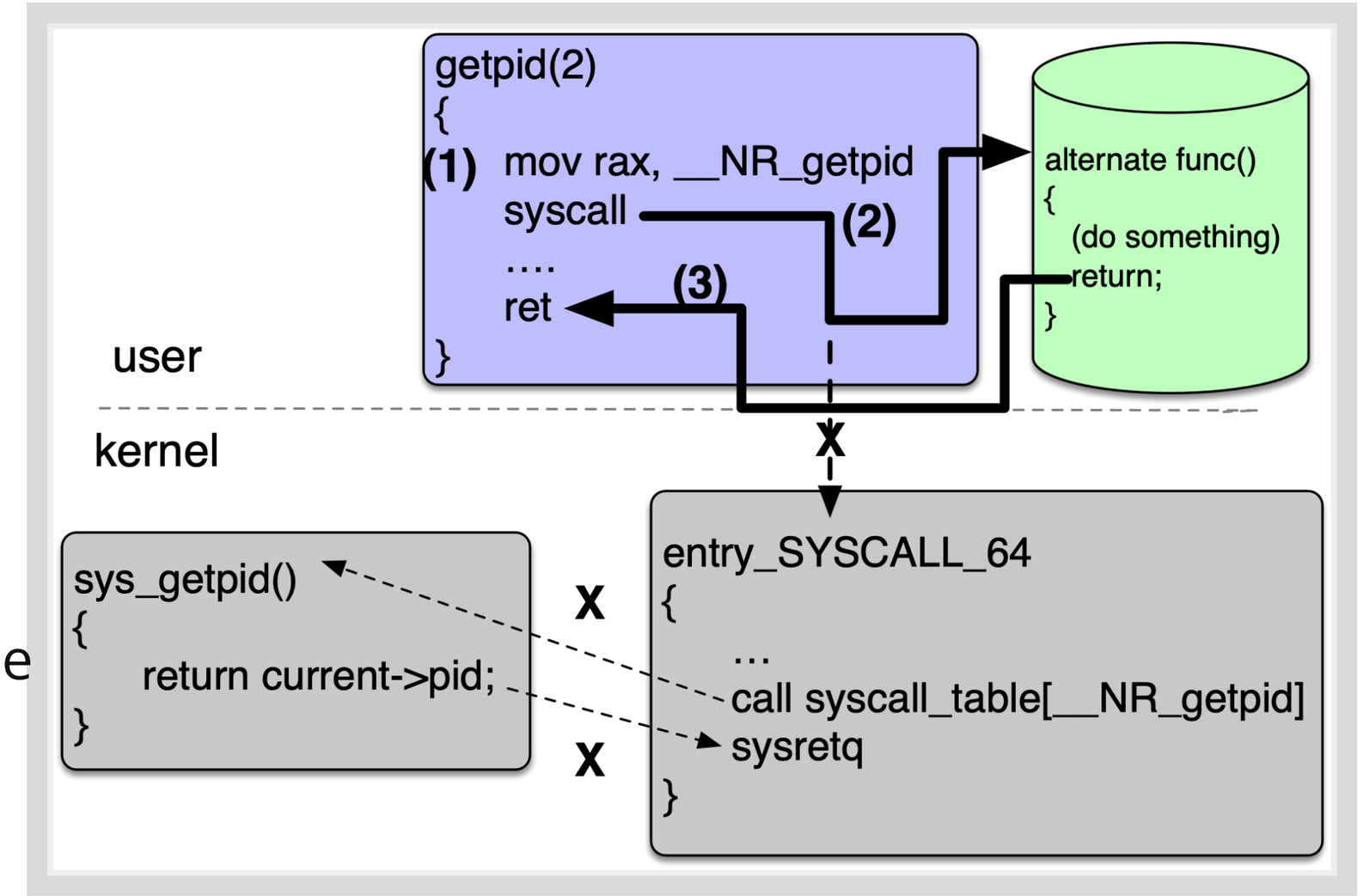


change the behavior of syscall ?

1. change kernel code,

   or

2. change userspace code

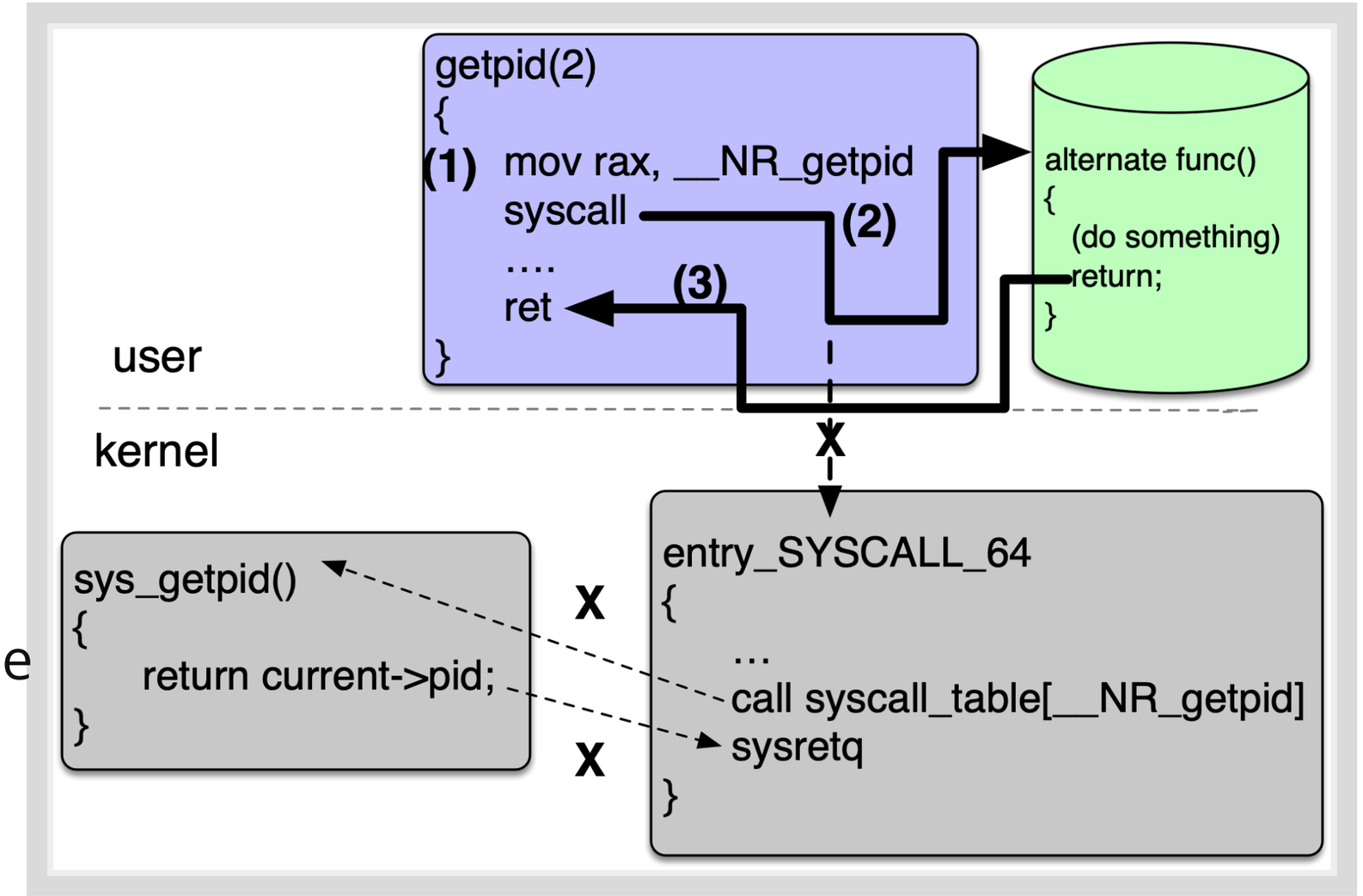


change the behavior of syscall ?

1. change kernel code,

   or

2. change userspace code



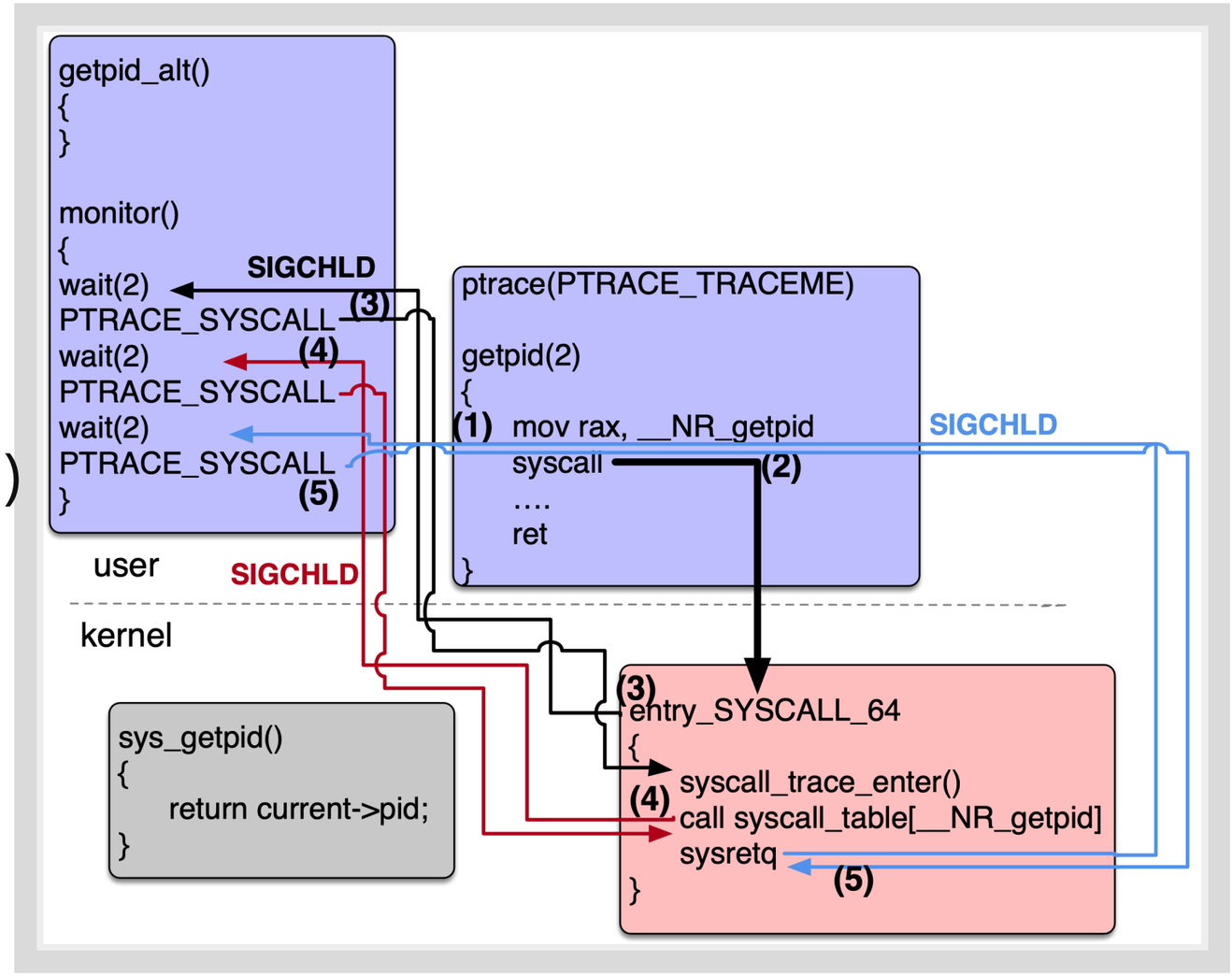
3. 1) hook syscalls and 2) call different syscall/function (interposition)
   - so that don't touch existing code

variants

- ptrace
- signal handler based hooks (int3, seccomp, SUD)
- symbol replacement (i.e. LD_PRELOAD)
  - libc replacement
- binary rewrites (hermitux, x-container, etc)
- syscall table hijack (in-kernel)

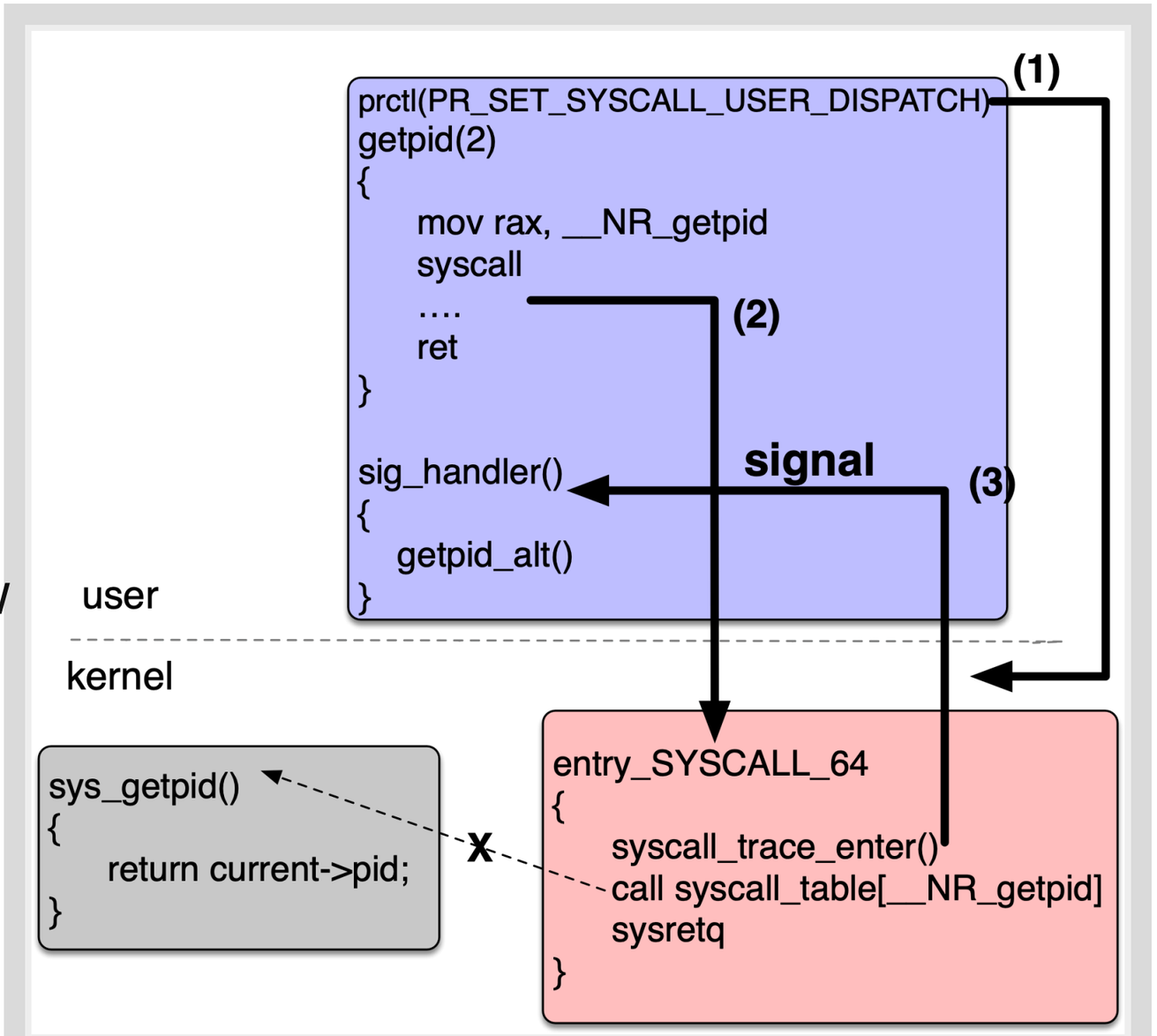
ptrace

- **monitor process events with memory/register information**
- users: UML, gvisor
- pros
  - transparent (req, trap and call)
  - multiple arch/platforms
- cons
  - slow (3 ctx switches / syscall)



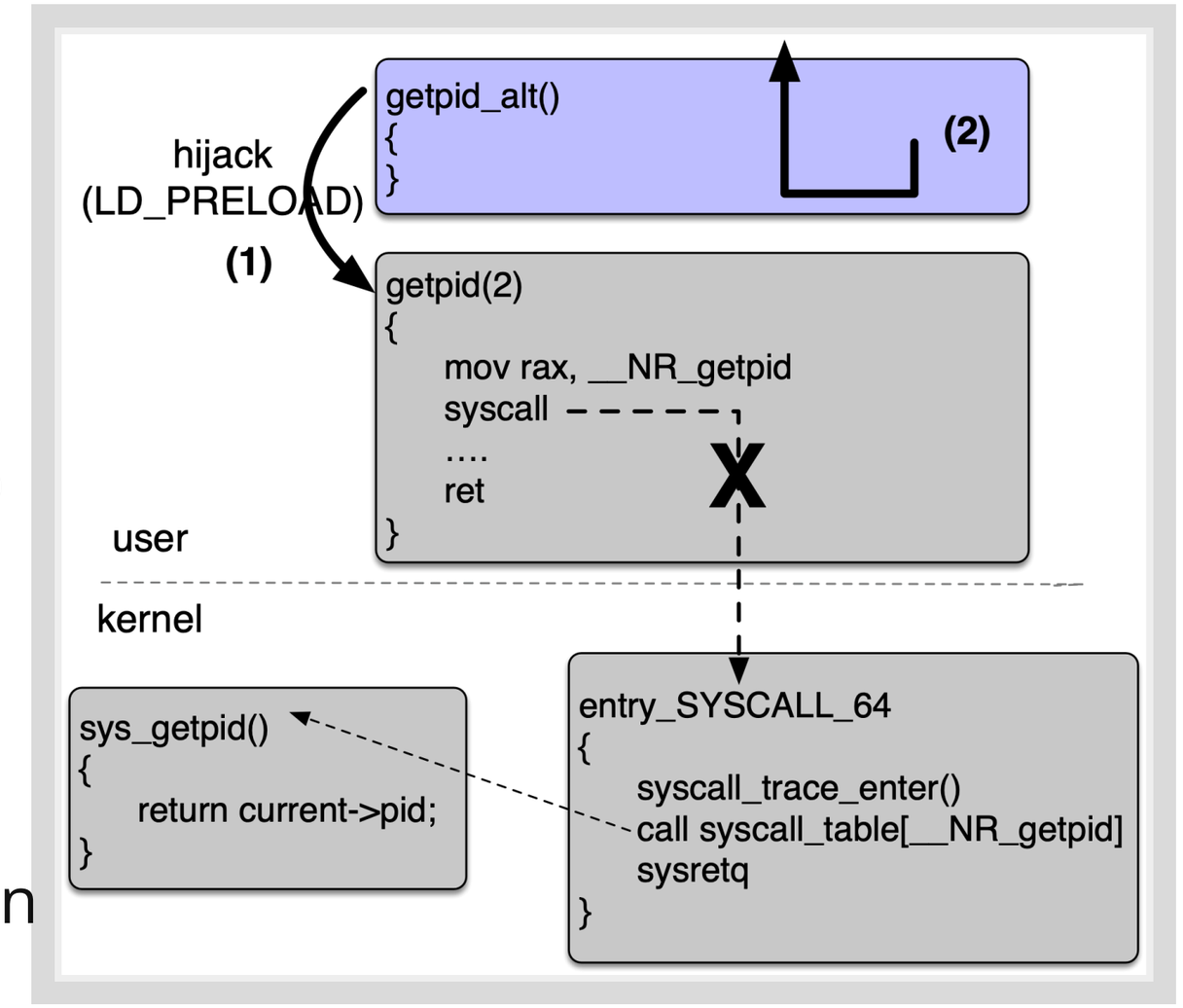
SUD (syscall user dispatch)

- **syscall filter within a process via signal**
- usage: Wine ?
- pros
  - no process switching (unlike ptrace)
- cons
  - cost of signal delivery isn't low
- alternative
  - seccomp(SCMP_ACT_TRAP) => gvisor
  - int3 trap (SIGTRAP)



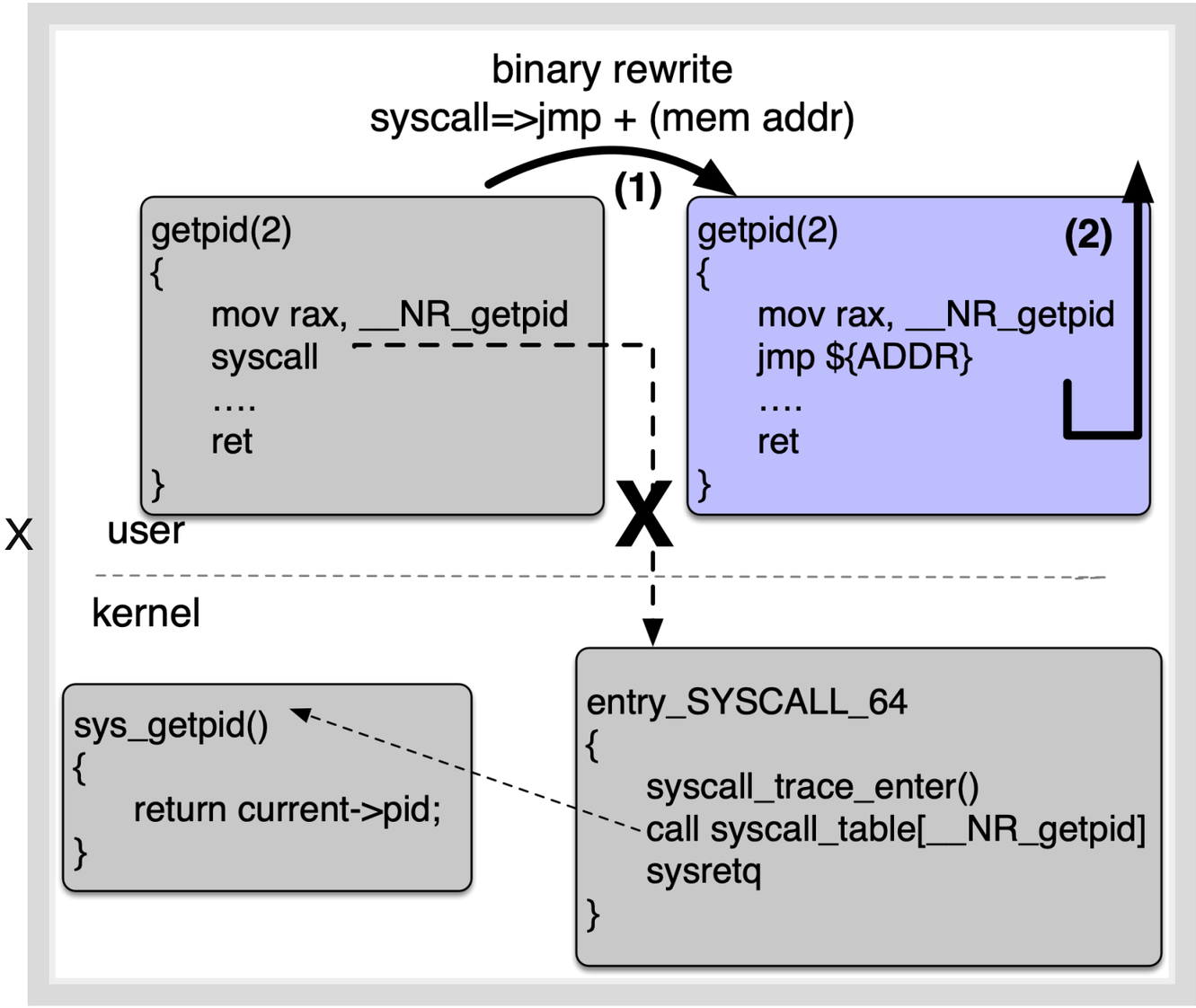
LD_PRELOAD

- **symbol replace before main()**
- usage: rdma socket (rsocket), userspace fs/net subsystem, etc
- pros
  - no runtime overhead
- cons
  - symbols are not always visible (thus not rewritable)
  - not effective after LD_PRELOAD (e.g. JIT)
  - before LD_PRELOAD either
  - doesn't work w/ static binaries
- alternative
  - libc replacement (resolve hidden symbol problem)

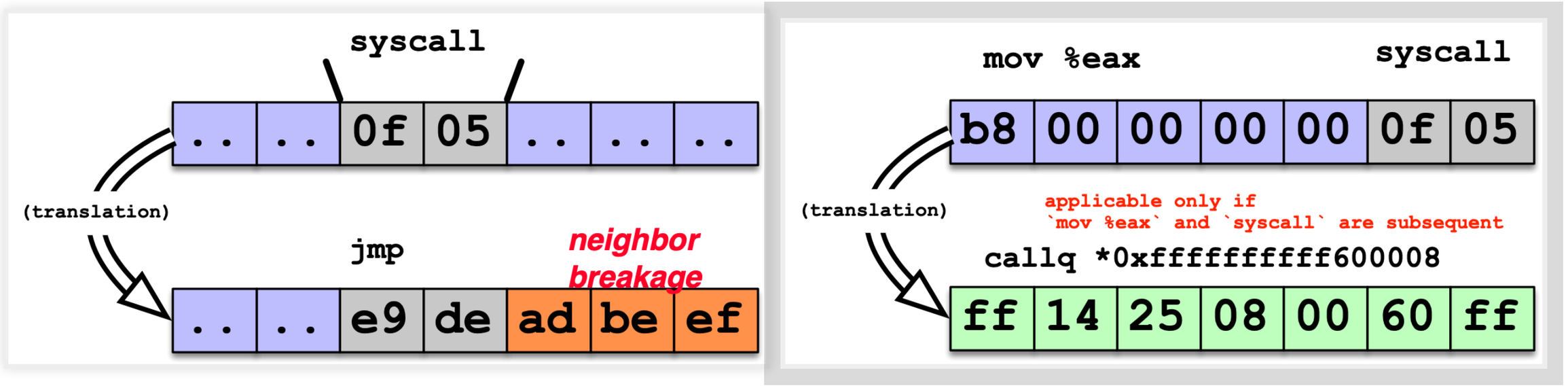


binary rewriting

- rewrite `syscall/sysenter` instruction in binary
  - either offline/online
- usage: gvisor (usertrap), HermiTux
- pros
  - no runtime overhead
- cons
  - break neighbor instructions



binary rewriting (cont'd)



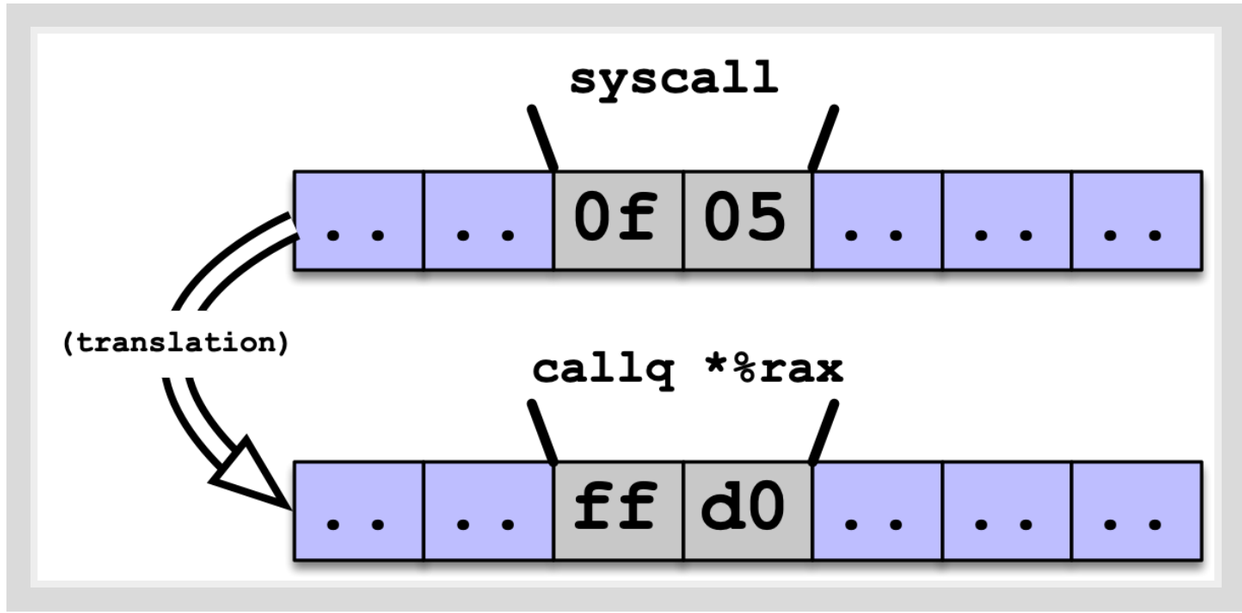
0. suppose wish to convert
   - `syscall` => `jmp 0xdeadbeef` (jmp rel32)
1. no sufficient room to rewrite 5 bytes instructions
   - `syscall` == `0f 05` (2 bytes)
   - `jmp 0xdeafbeef` == `e9 de ad be af` (5 bytes)
   - destroy neighbor instructions
2. depends on the order of instructions
   - `mov sysno %rax; syscall` (7 bytes)
   - `callq ${addr of handler}` (7 bytes)
   - *not 100% binaries are like that.*

summary of existing syscall hooks

	type	speed	coverage	user
native	syscall	--	--	--
ptrace	signal	😢😢	🤱	uml, gvisor
sud	signal	😢	🤱	wine?
seccomp	signal	😢	🤱	gvisor
int3	signal	😢	🤱	?
ldpreload	symbol rewrite	🤱🤱	😢😢	rsocket
bin rewrite	instruction rewrite	🤱	😢	unikernels

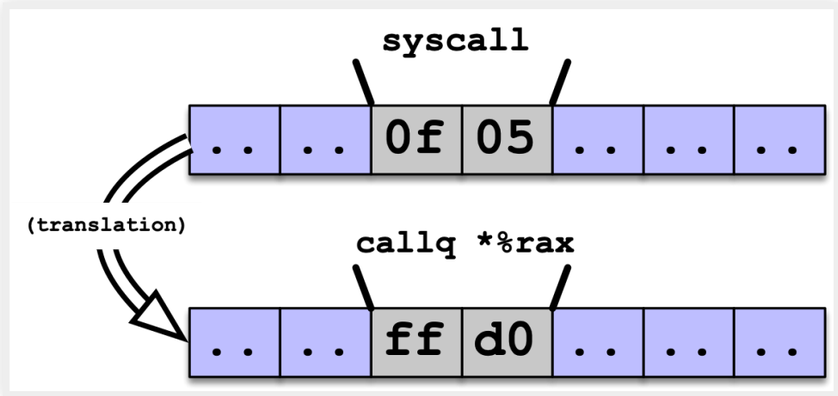
zpoline

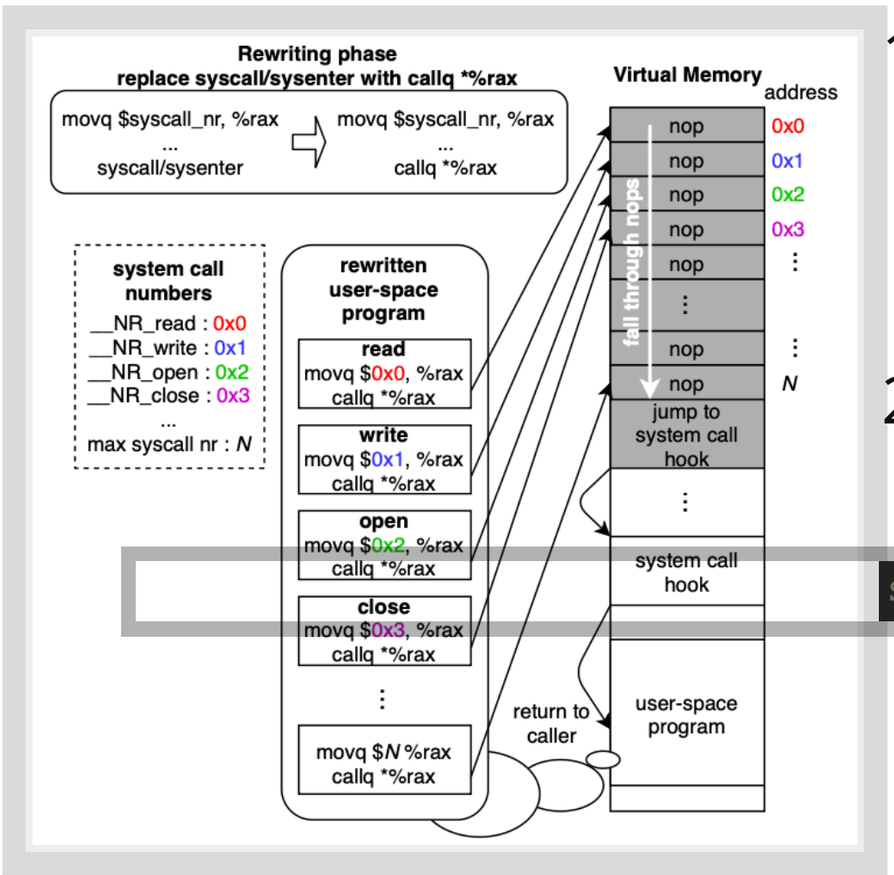
- hook by replacing `syscall` with `callq *%rax`
  - a binary rewriting approach
- pros
  - fast (no signal, func call)
  - exhaustive hook
  - no neighbor inst breakage
- cons
  - start-up delay
  - (but no runtime delay)



- `%rax` contains syscall number (0~500)
- so replaced instruction jumps to address 512

zpoline: how it works





How `callq *%rax` works ?

0. **rewrite all `syscall` instructions on ELF loading**
   - fill `nop` instruction from address 0x0
   - trampoline code at the end of `nop` segment (requires to turn the kernel writable)
1. **upon reached in the syscall (`callq *%rax`)**
   - `%rax` contains syscall number (0~500)
   - jump to the address (0 - 500)
   - nop slide until the trampoline code
2. **call user-defined hook program**
   - do whatever you want

```
$ echo 0 > "/proc/sys/vm/mmap_min_addr"
```

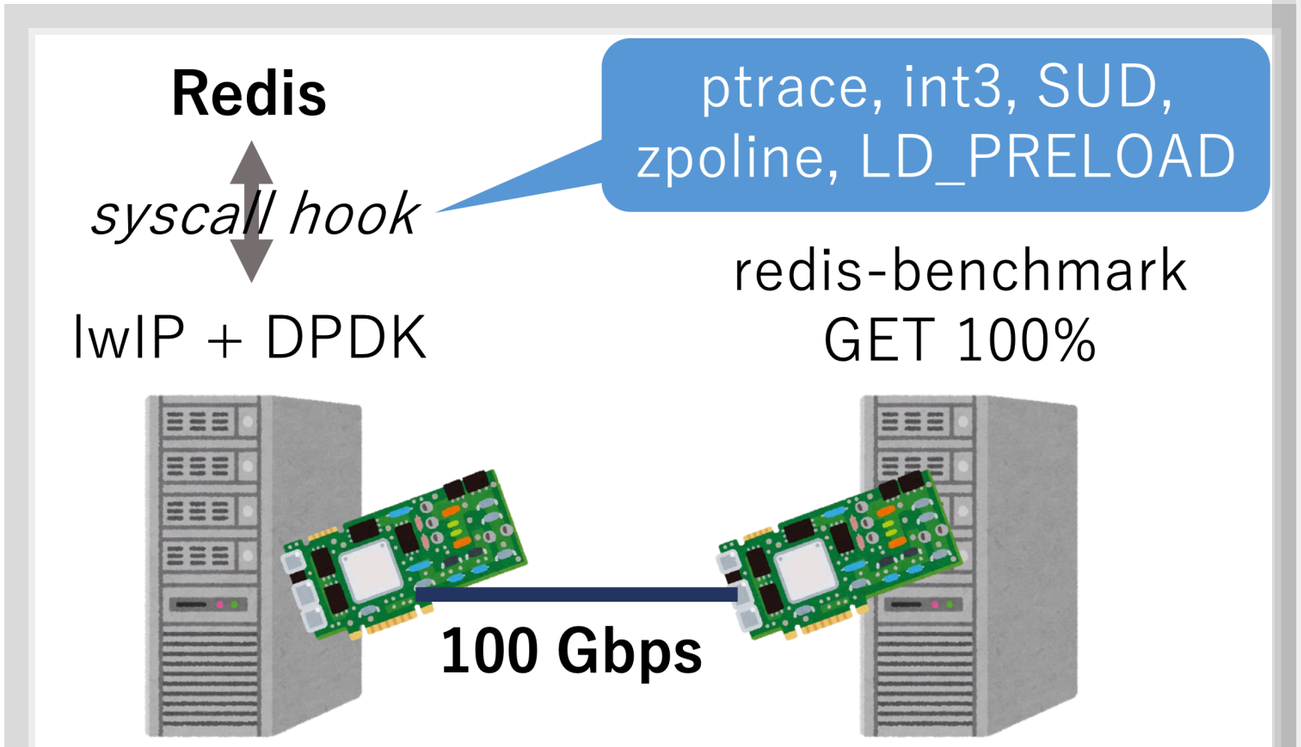
zpoline: how it behaves

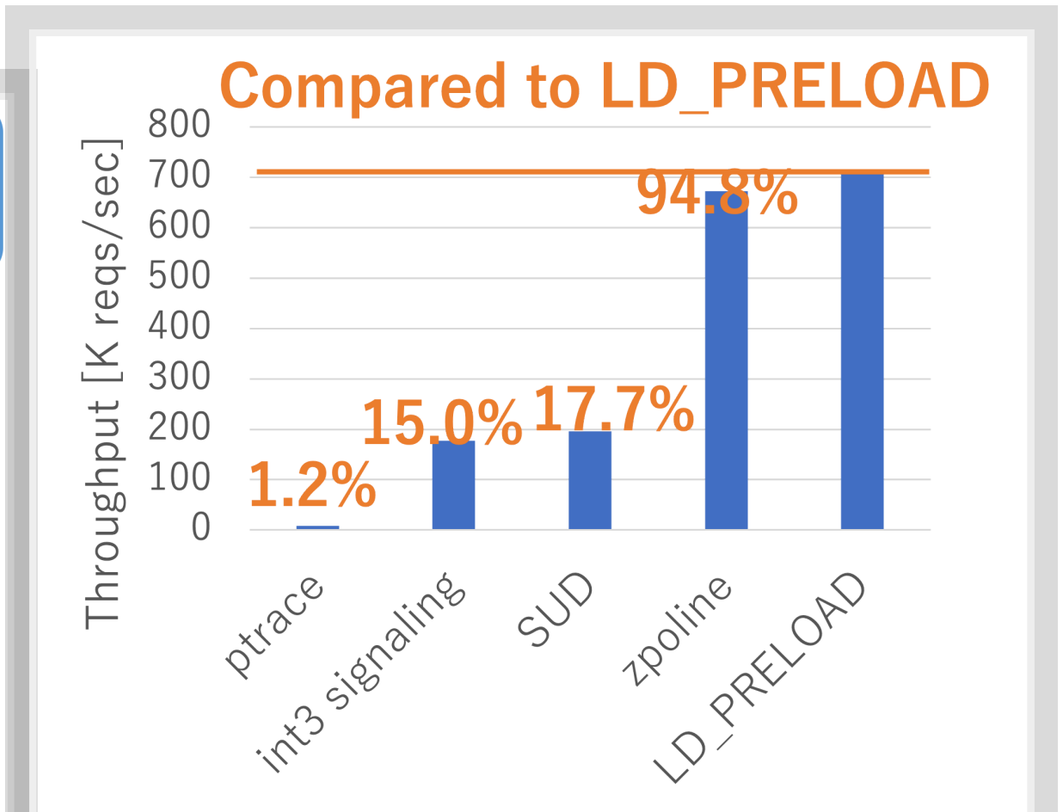
	getpid (nsec)
native	116
ptrace	17442
sud	1663
seccomp	1720
int3	1396
zpoline	31
ldpreload	2

- (micro) benchmark
  - getpid(2) 1000times
  - (except native) return a dummy value in hook funcftion
  - so that we can observe the overhead of hook mechanism

*actual syscall isn't called except the native case

zpoline how it behaves (cont'd)





- redis benchmark
  - load: GET 100%
  - run redis/lwip/DPDK
  - w/ different syscall hooks

zpoline: benefits

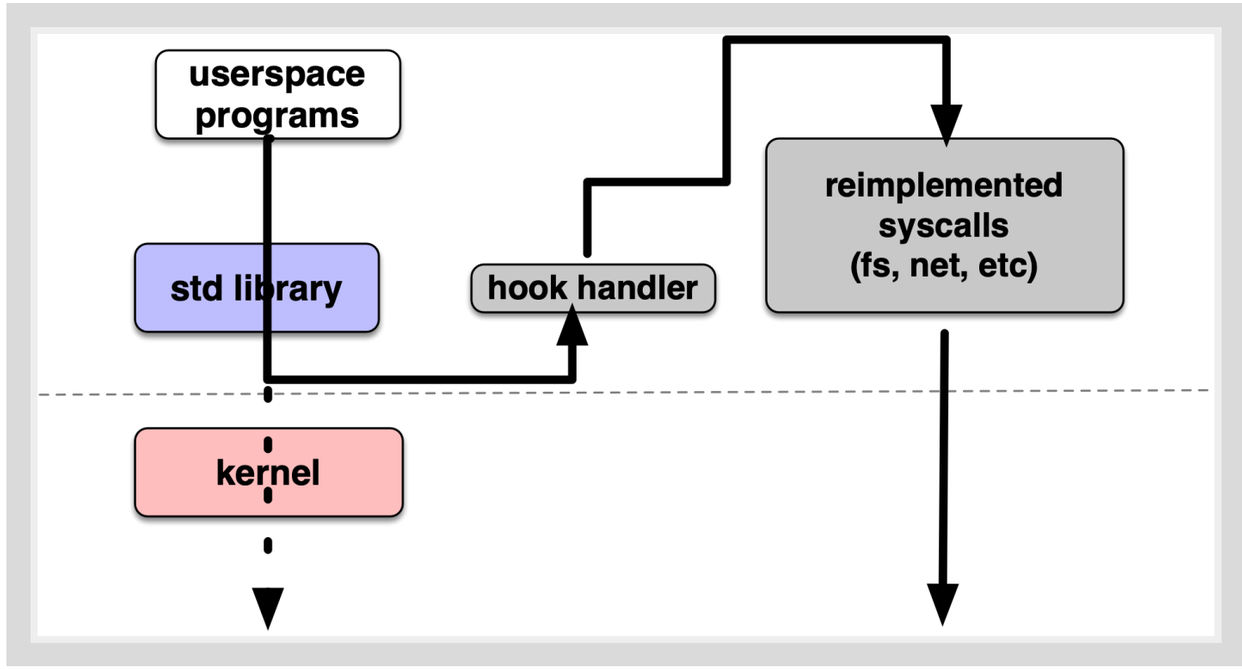
	type	speed	coverage	user
native	syscall	--	--	--
ptrace	signal	😢😢	🤱	uml, gvisor
sud	signal	😢	🤱	wine?
seccomp	signal	😢	🤱	gvisor
int3	signal	😢	🤱	?
ldpreload	symbol rewrite	🤱🤱	😢😢	rsocket
bin rewrite	instruction rewrite	🤱	😢	unikernels
zpoline	instruction rewrite	🤱	🤱	(welcome!)

- zpoline does the best
  - fast (no signal, func call)
  - exhaustive hook coverage
  - no neighbor instruction breakage

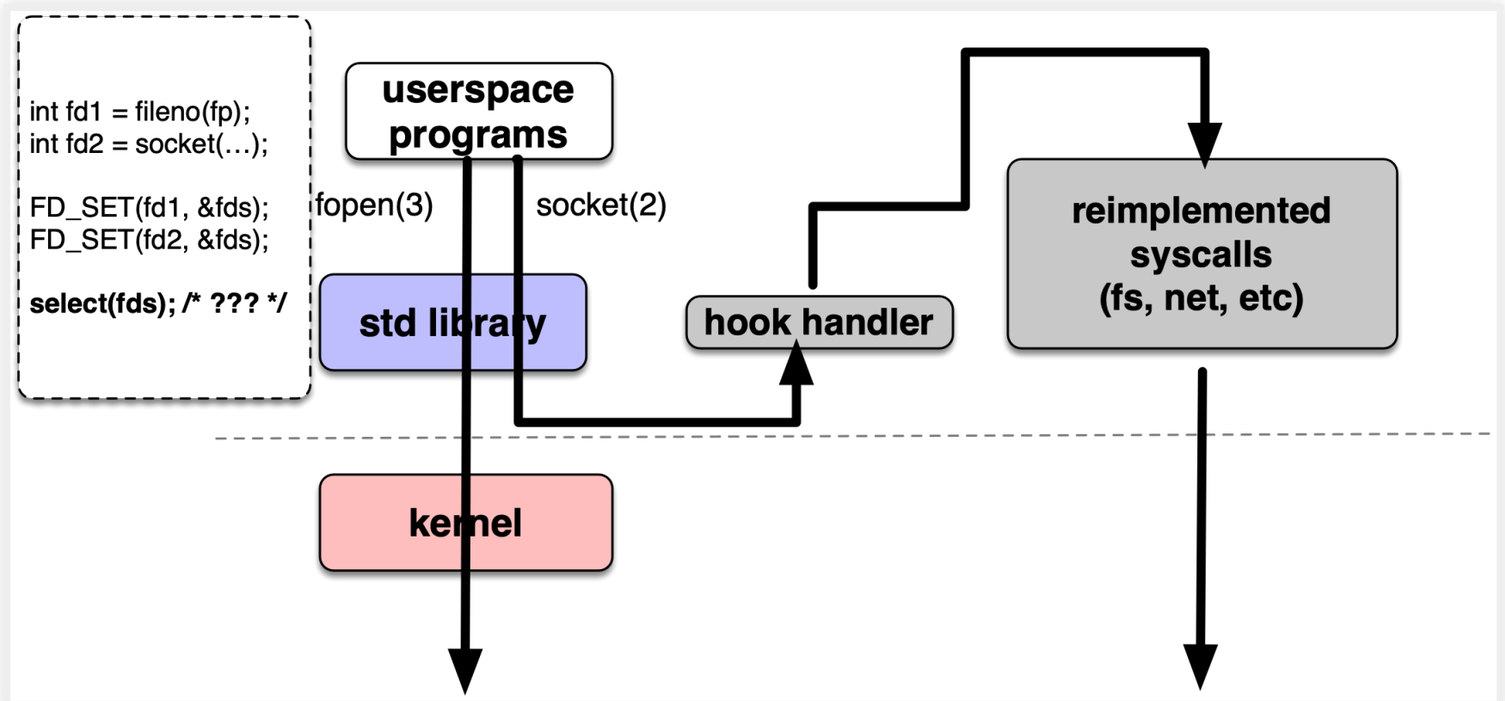
There ain't no such things as a free lunch.

pitfalls of syscall hooks

- type of syscall hook
  - partial hook
  - complete hook
- after you hooked a syscall,
  you need to do syscalls by yourself
  - userspace tcp,
  - filesystems,
  - kernel emulation
- not trivial at all..



handling two universes in partial hooks



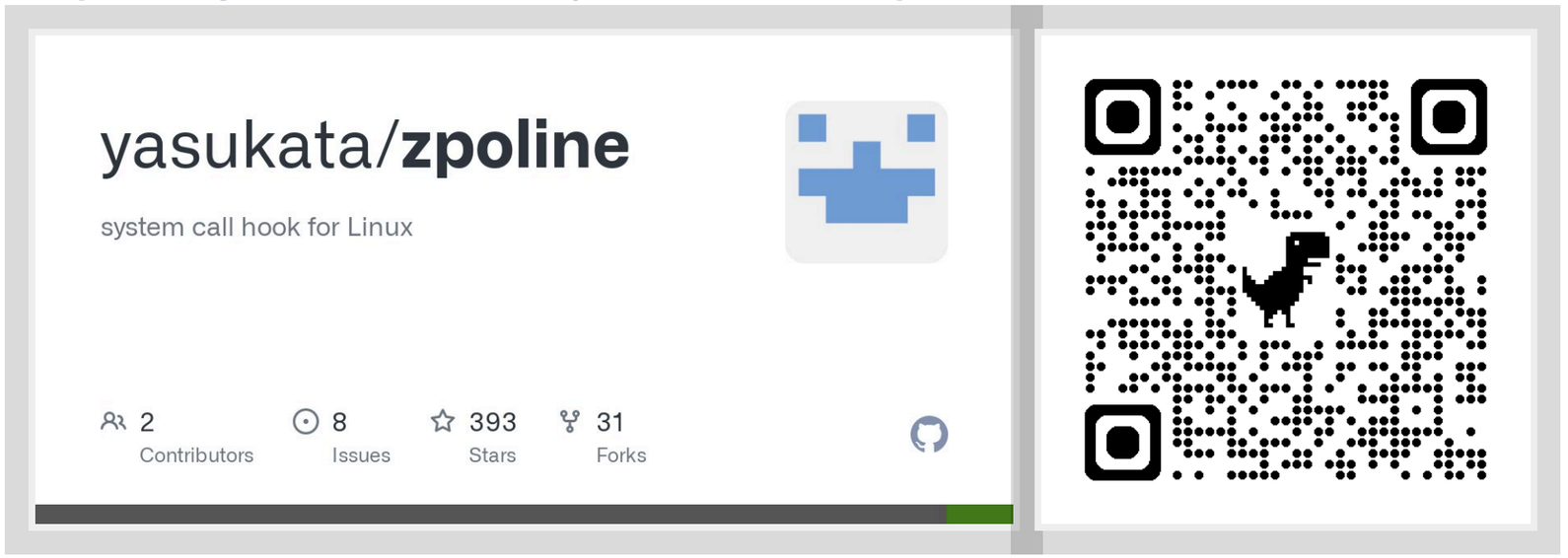
- need of caring multiple universes (host/guest)
- example: poll to host files and socket in userspace TCP
  - in generic kernel, both files/sockets are in the same kernel
  - can poll at once (e.g., select/epoll_wait)
- need to care in the middle

```
int hooked_select(pollfds[], nfds_t, int) {
  int host_fd = host_poll();
  int user_fd = user_poll();
  return (merge {host,user}_fd)
}
```

Summary

- review of existing syscall hook mechanisms
- zpoline: an approach to fix issues on LD_PRELOAD and bin-rewrites

- code: https://github.com/yasukata/zpoline



Backups

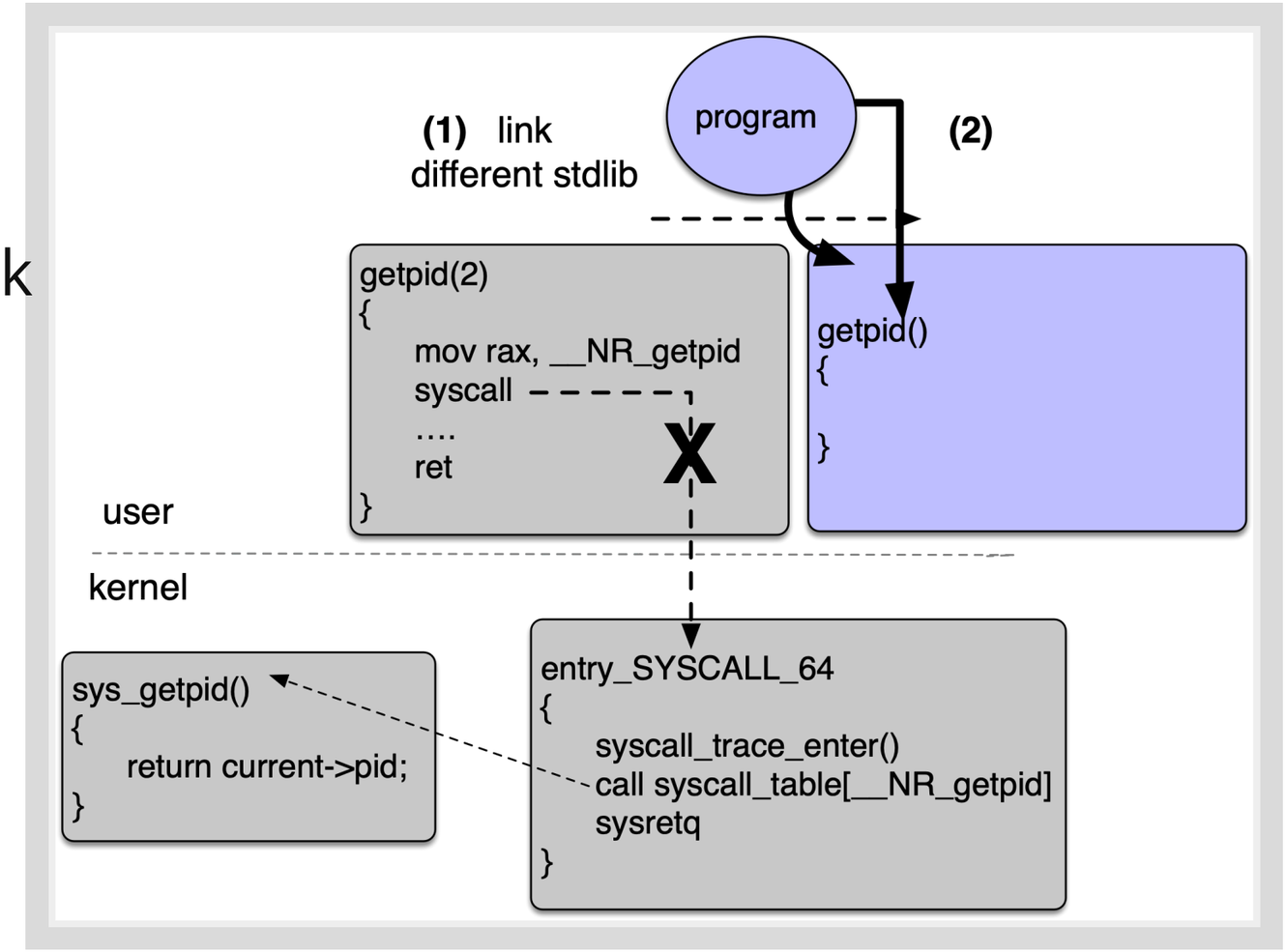
How zpoline is started ?

- use a launcher program
  1. load a binary
  2. replace instructions (`syscall`)=>`callq *%rax`)
  3. load (new) syscall handler
  4. call `main()` function
- we can do offline

libc replacement

(maybe backup?)

- replace std library and re-link to programs
- pros
  - no runtime overhead
- cons
  - maintenance burden (if out-of-tree)

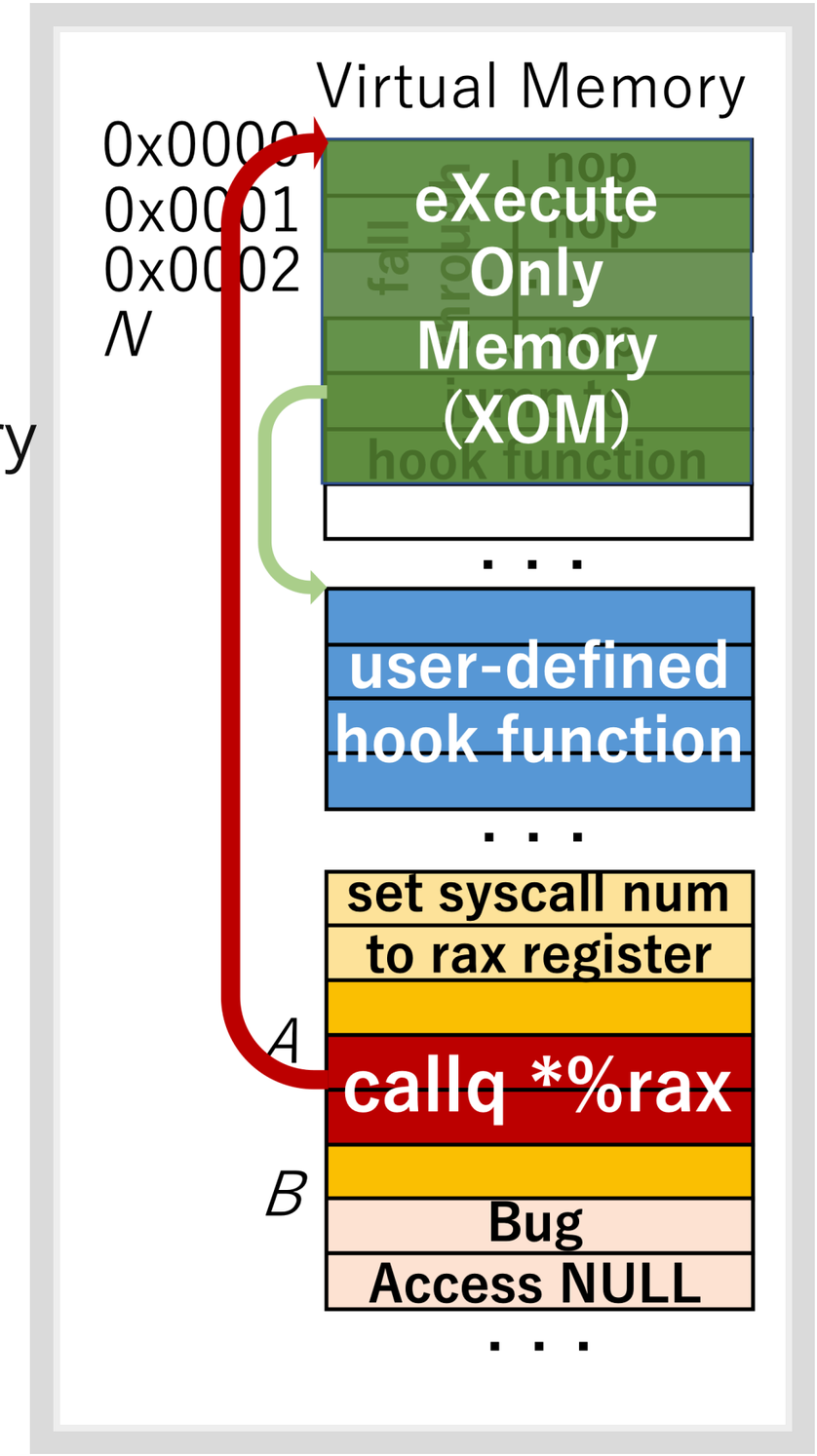


platform support

- Supported
  - OS : Linux, FreeBSD, NetBSD, DragonFly BSD
  - CPU : x86-64
- Unsupported
  - OS : Windows, macOS, OpenBSD
  - CPU : ARM, and so on

null access termination

- zpoline uses address **zer0**
  - thus no faults w/ accessing that memory
  - no signals upon program failure
- solution
  - r/w: XOM (eXecute only memory via `mprotect(2)`)
  - x: additional checks of caller address



References

- code: https://github.com/yasukata/zpoline
- paper: https://www.usenix.org/conference/atc23/presentation/yasukata