USENIX ATC 2023 – Boston, MA, USA – 10th July

zpoline: a system call hook mechanism based on binary rewriting

Kenichi Yasukata[1], Hajime Tazaki[1], Pierre-Louis Aublin[1], Kenta Ishiguro[2]

[1] IIJ Research Laboratory

[2] Hosei University

System Call

- System calls are the primary interface for user-space programs to communicate with OS kernels

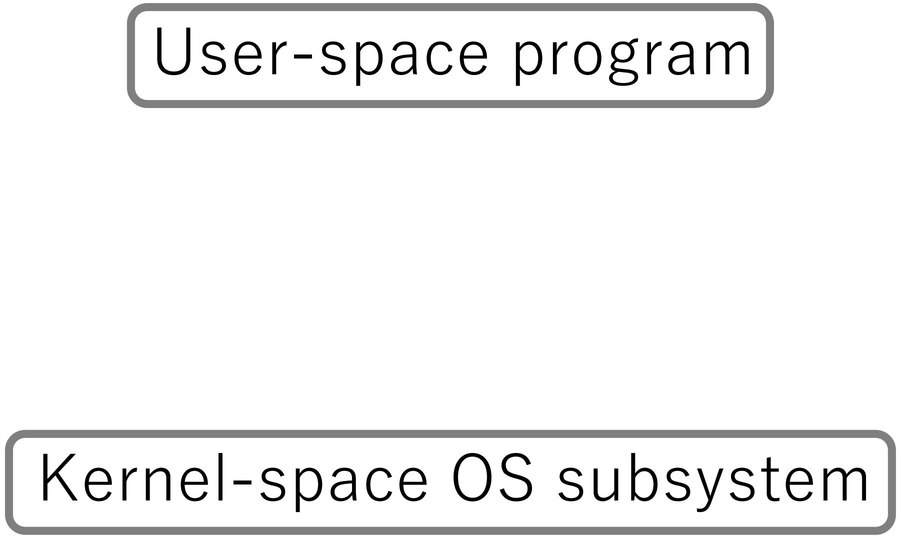
System Call

- System calls are the primary interface for user-space programs to communicate with OS kernels

User-space program

System Call

- System calls are the primary interface for user-space programs to communicate with OS kernels

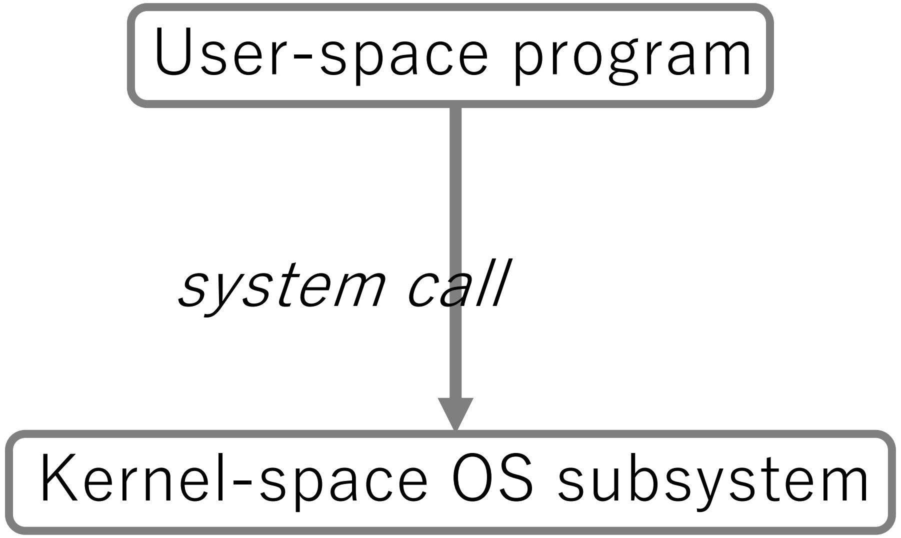


User-space program

Kernel-space OS subsystem

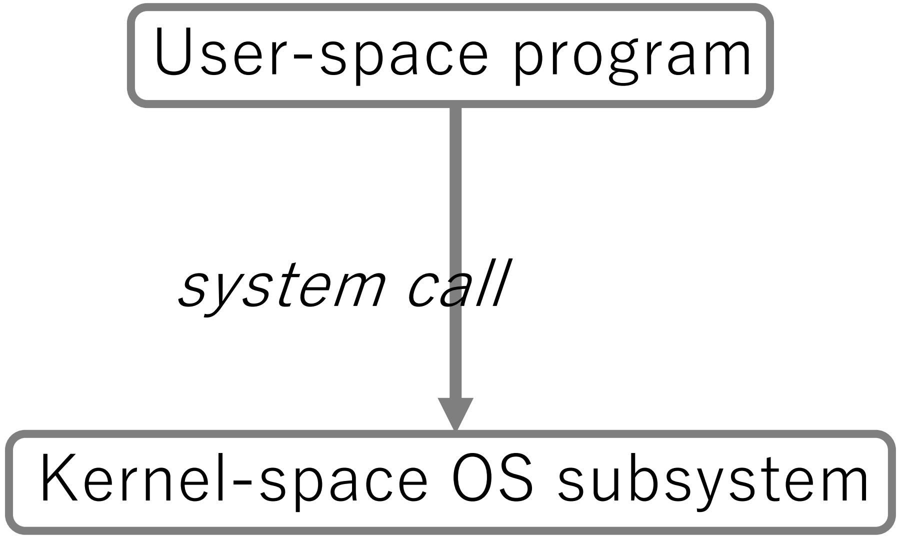
System Call

- System calls are the primary interface for user-space programs to communicate with OS kernels



System Call Hook

- System calls are the primary interface for user-space programs to communicate with OS kernels
- A system call hook mechanism intercepts a system call



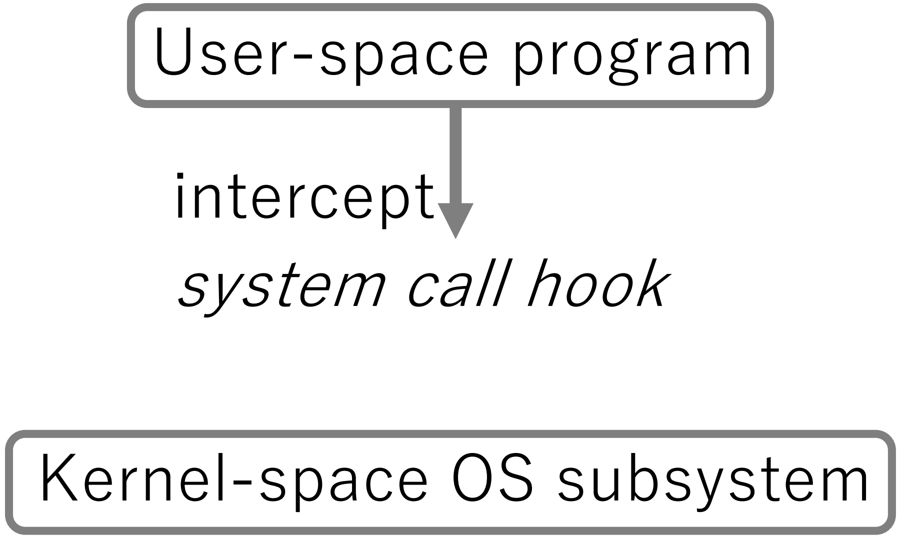
User-space program

system call

Kernel-space OS subsystem

System Call Hook

- System calls are the primary interface for user-space programs to communicate with OS kernels
- A system call hook mechanism intercepts a system call


User-space program

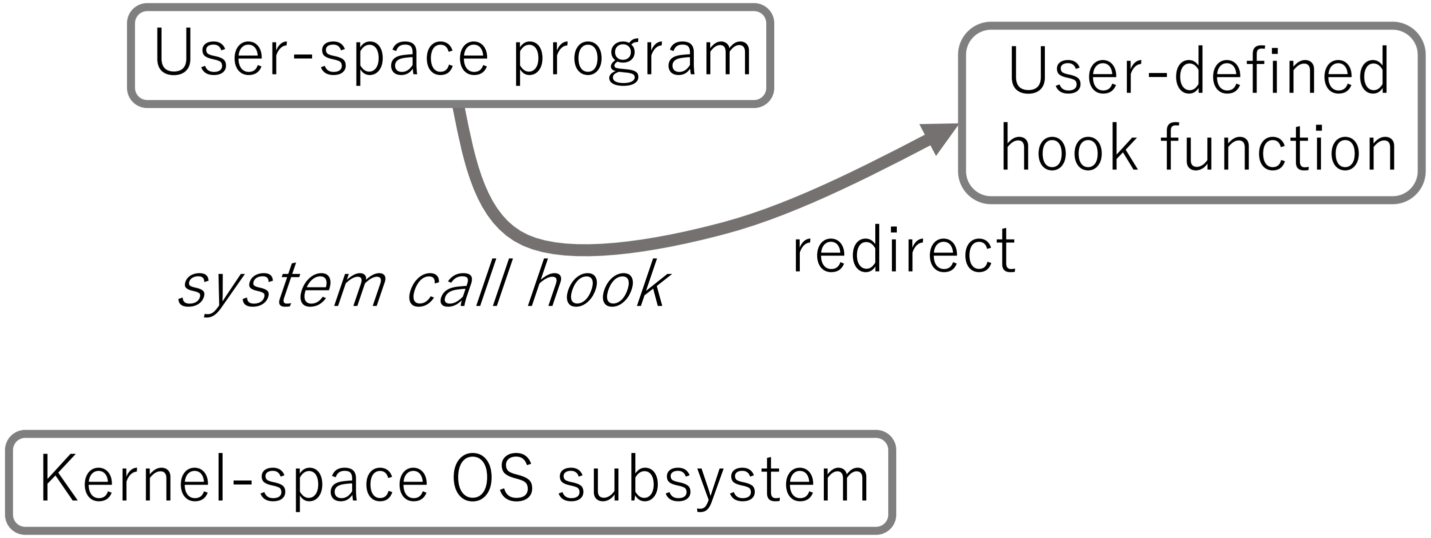
intercept

system call hook

Kernel-space OS subsystem

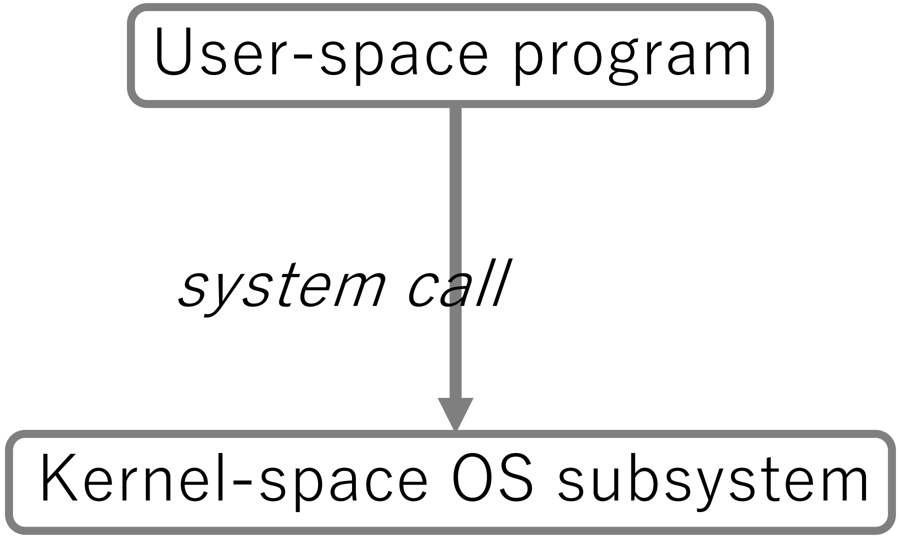
System Call Hook

- System calls are the primary interface for user-space programs to communicate with OS kernels
- A system call hook mechanism intercepts a system call, and redirects the execution to a user-defined hook function



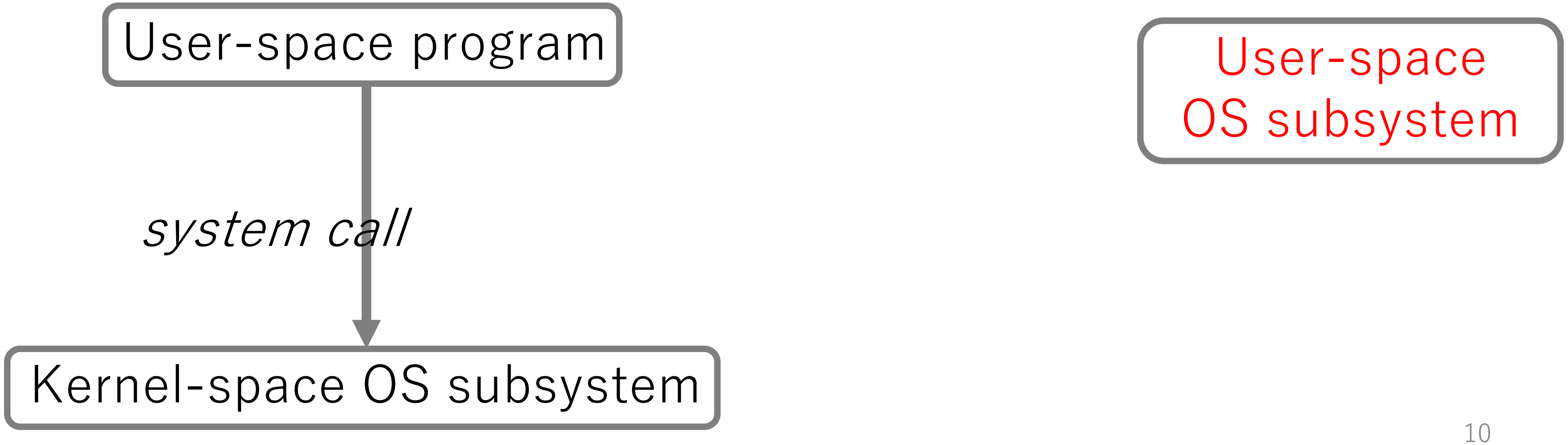
Motivating Use Case

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications



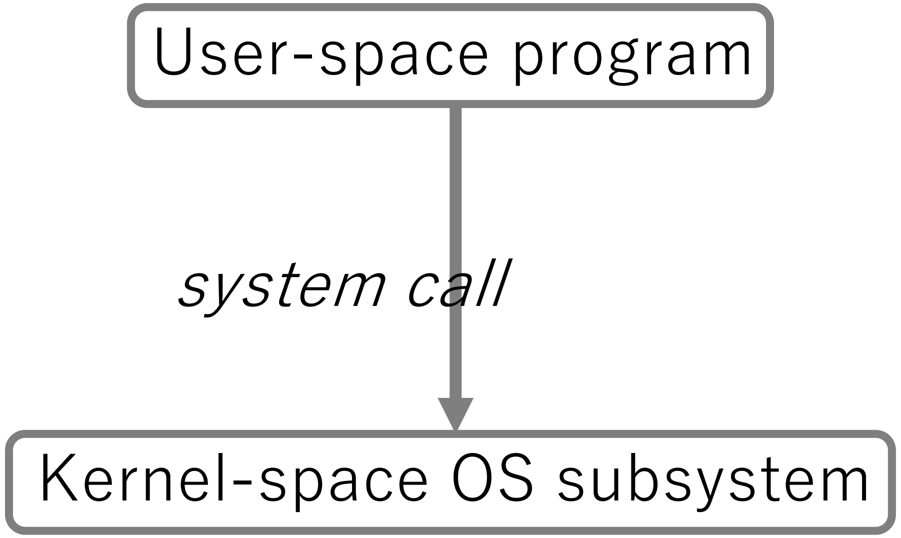
Motivating Use Case

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications



Motivating Use Case

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications



User-space program

system call

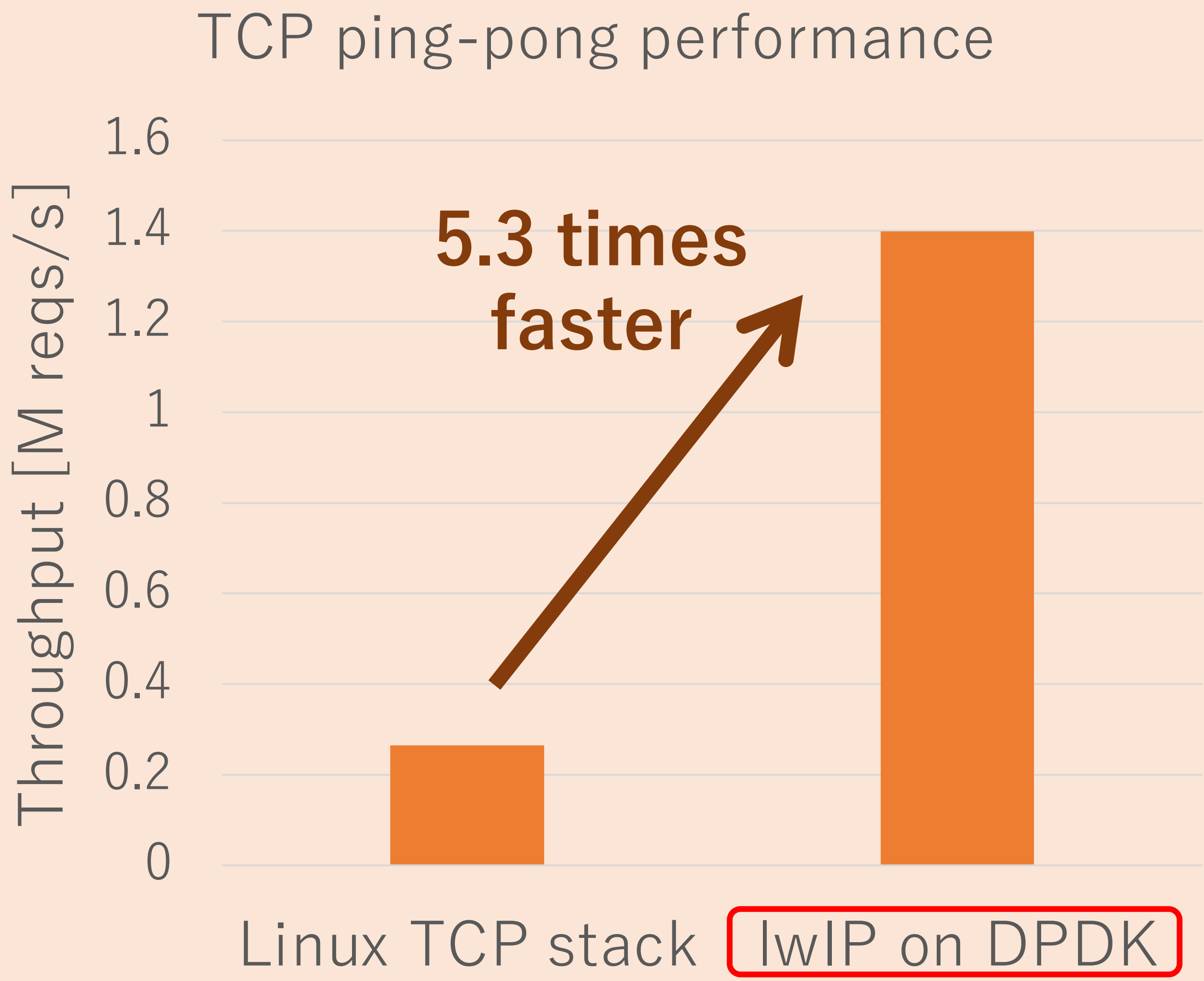
Kernel-space OS subsystem

User-space OS subsystem

Highly performant

Motivating Use Case

o transparently apply
plications



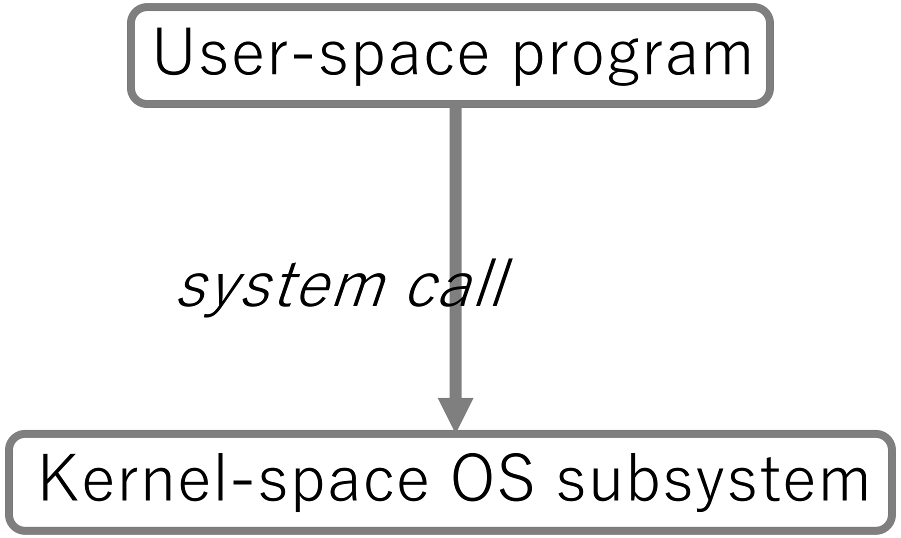
= user-space network stack

User-space
OS subsystem

Highly performant

Motivating Use Case

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications



User-space program

system call

Kernel-space OS subsystem

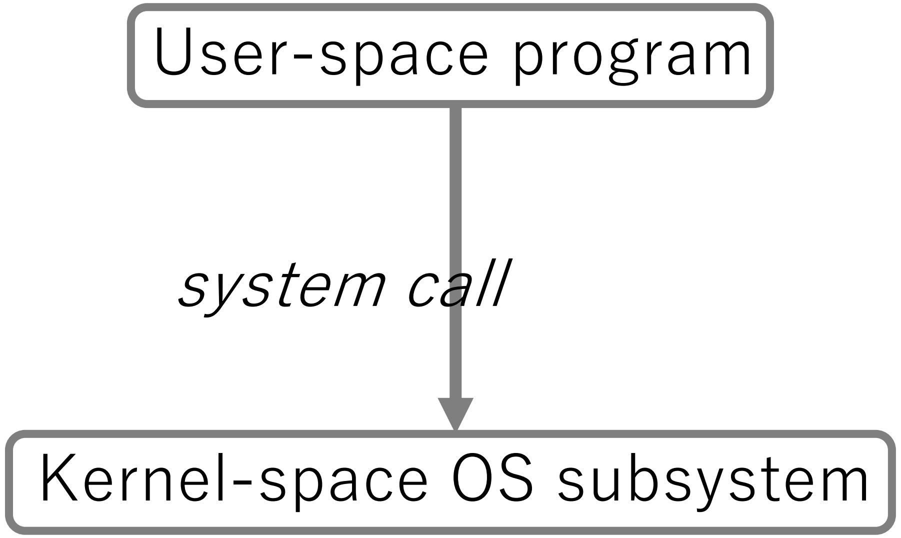
User-space OS subsystem

Highly performant

Motivating Use Case

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications

Normally, adaptation requires changes of a user-space program to apply a specific API of a user-space OS subsystem



User-space program

system call

Kernel-space OS subsystem

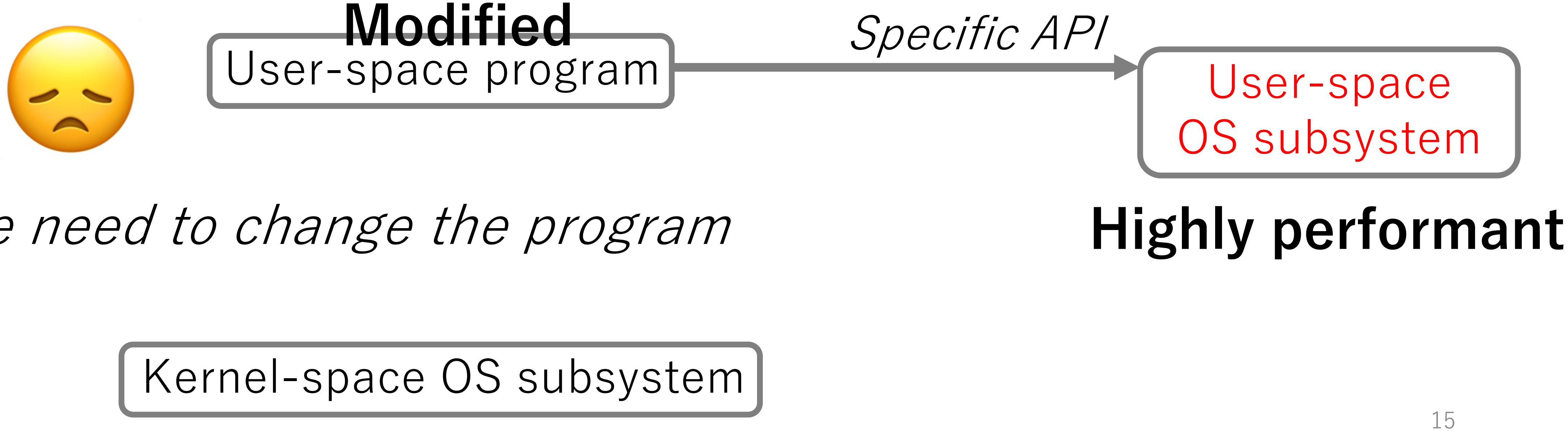
User-space OS subsystem

Highly performant

Motivating Use Case

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications

Normally, adaptation requires changes of a user-space program to apply a specific API of a user-space OS subsystem



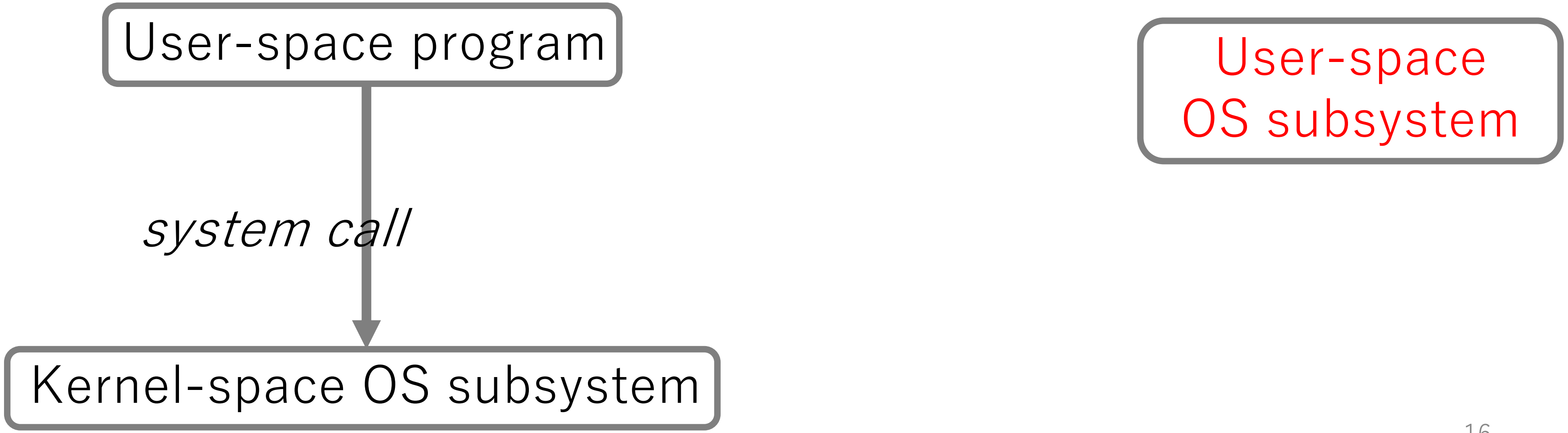
We need to change the program

Highly performant

Motivating Use Case

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications

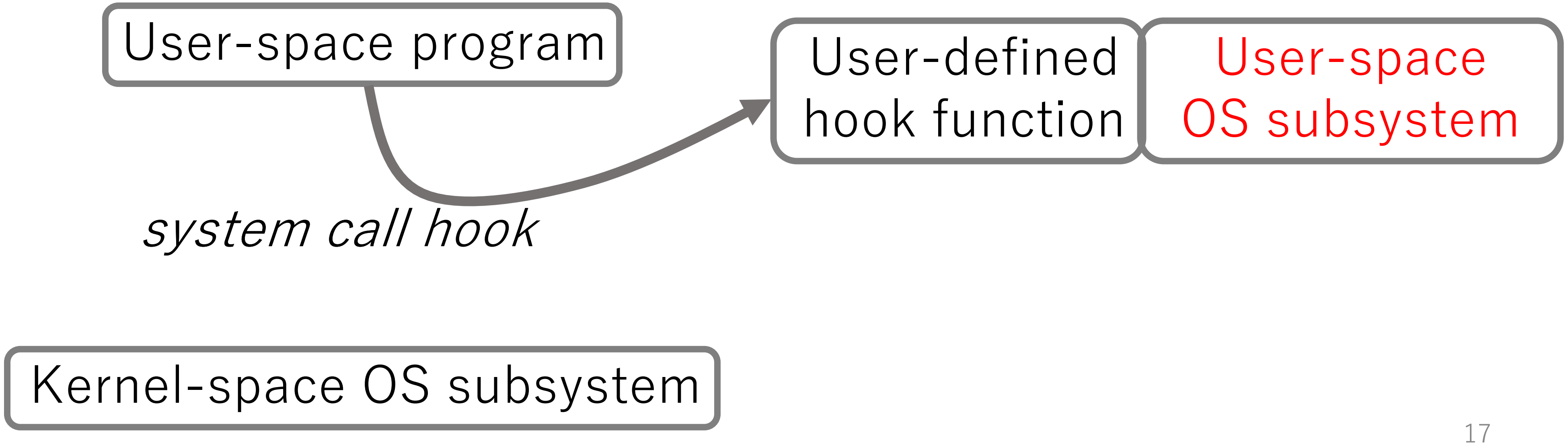
If we use a system call hook mechanism, ...



Motivating Use Case

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications

If we use a system call hook mechanism, ...



User-space program

system call hook

User-defined hook function

User-space OS subsystem

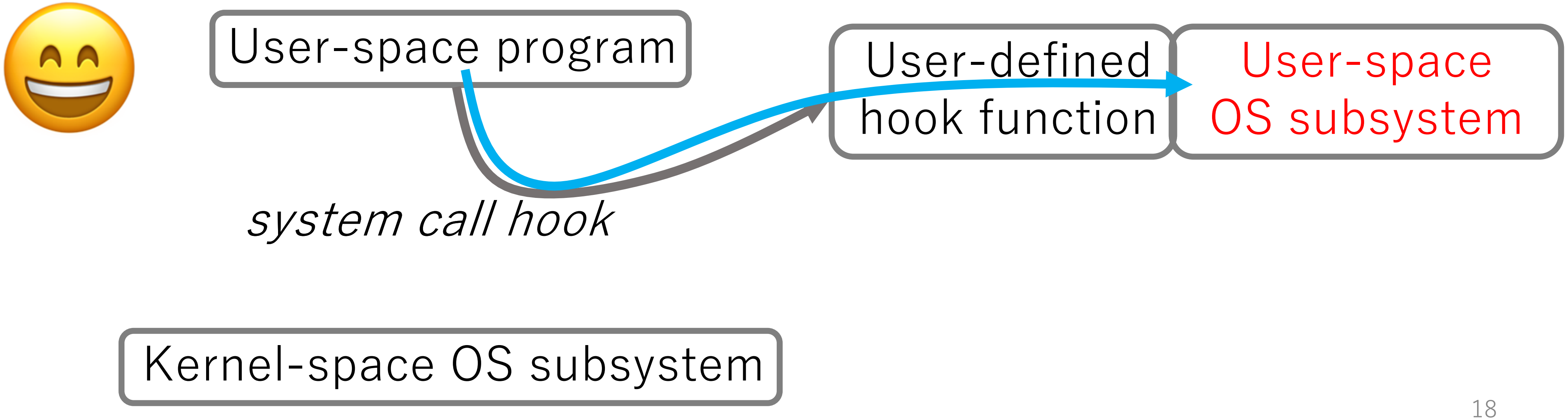
Kernel-space OS subsystem

Motivating Use Case

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications

If we use a system call hook mechanism, ...

no modification of the user-space program is necessary



User-space program

User-defined hook function

User-space OS subsystem

system call hook

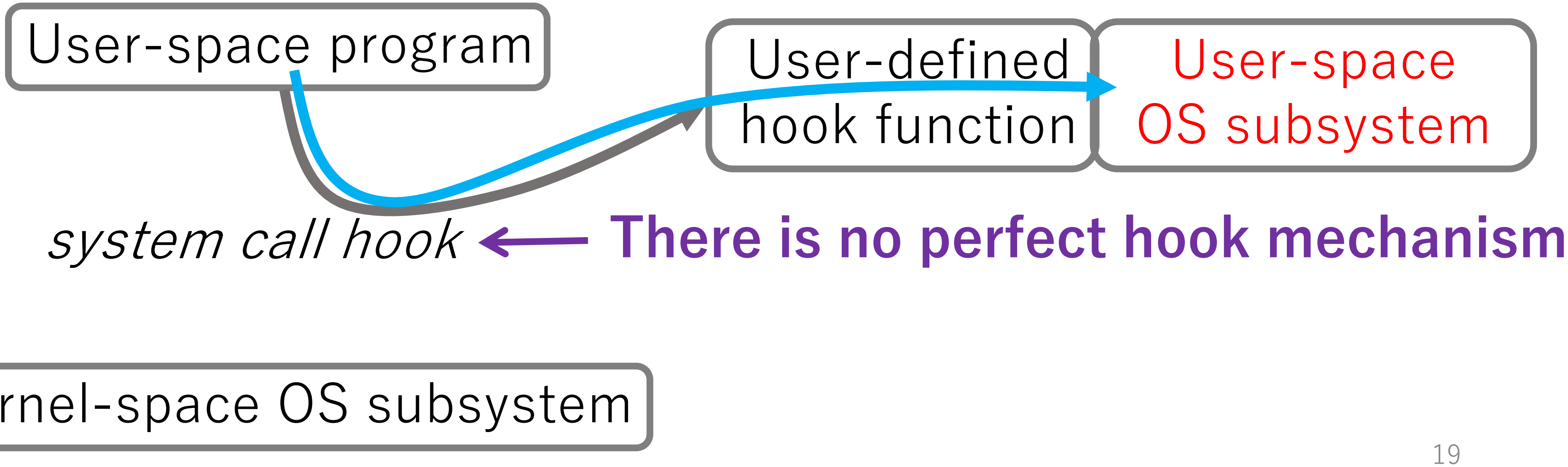
Kernel-space OS subsystem

Problem

- System call hook mechanisms allow us to transparently apply user-space OS subsystems to existing applications

If we use a system call hook mechanism, ...

no modification of the user-space program is necessary



There is no perfect hook mechanism

Problem

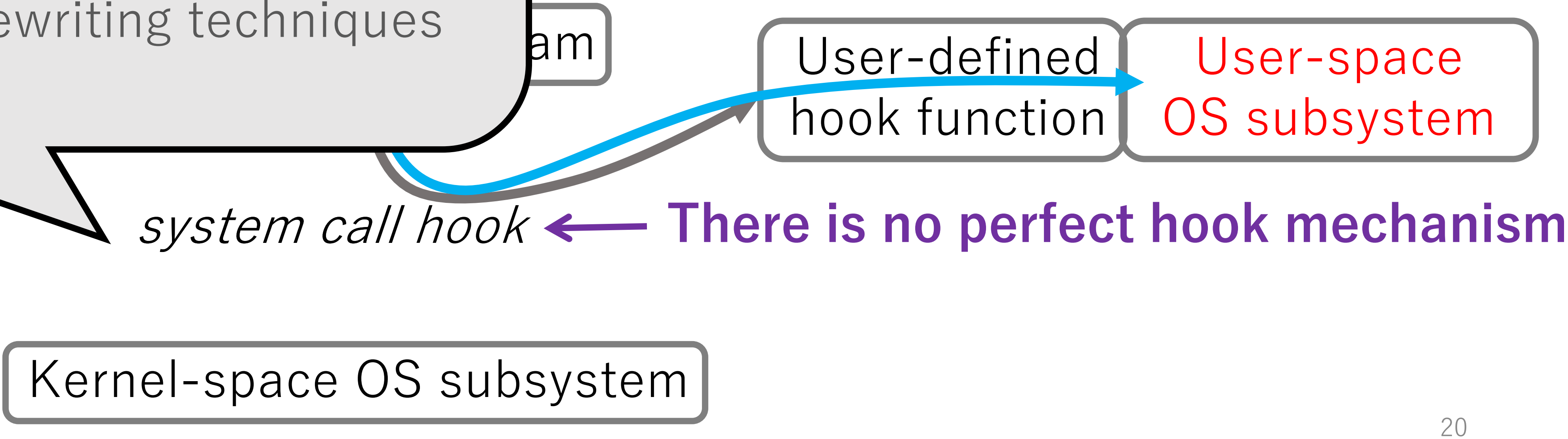
ms allow us to transparently apply
to existing applications

ok mechanism, ...

user-space program is necessary

Existing Mechanisms

- ptrace
- int3 signaling technique
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques
- …

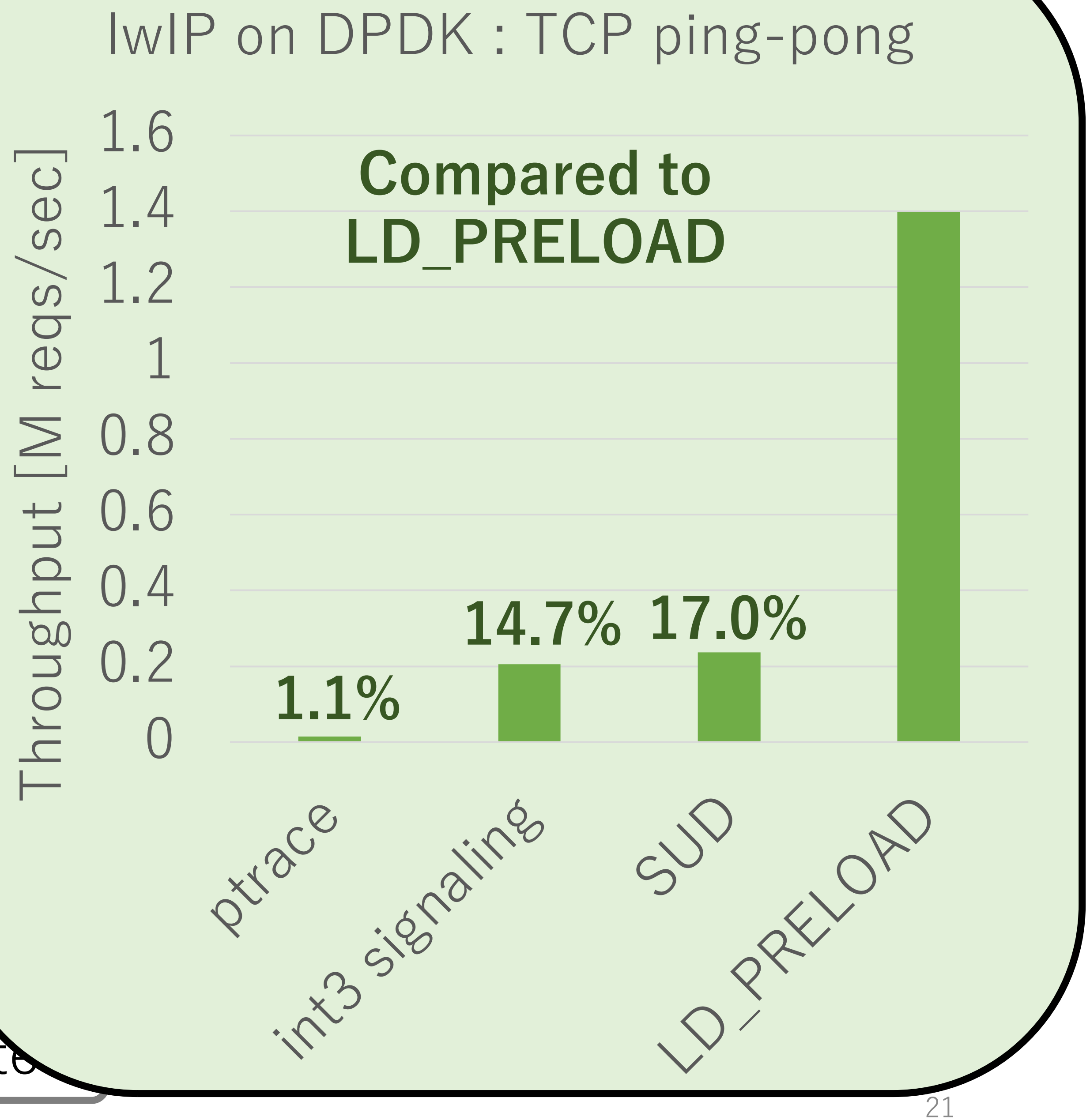


There is no perfect hook mechanism

Problem

Existing Mechanisms

- ptrace
- int3 signaling technique
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques
- …



system call hook

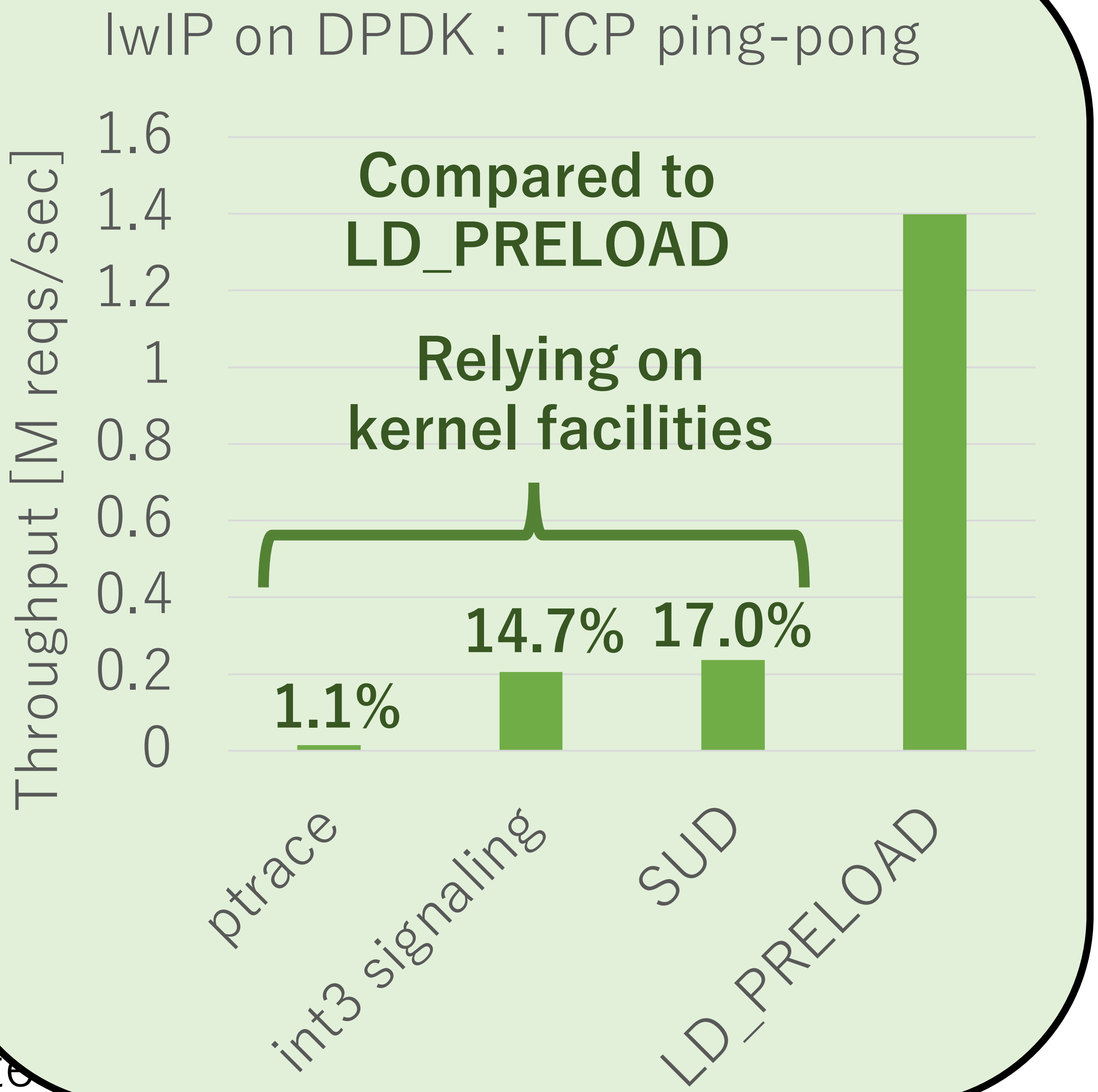
Kernel-space OS subsyste

Problem

Existing Mechanisms

- ptrace
- int3 signaling technique
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques
- …

lwIP on DPDK : TCP ping-pong



Compared to LD_PRELOAD

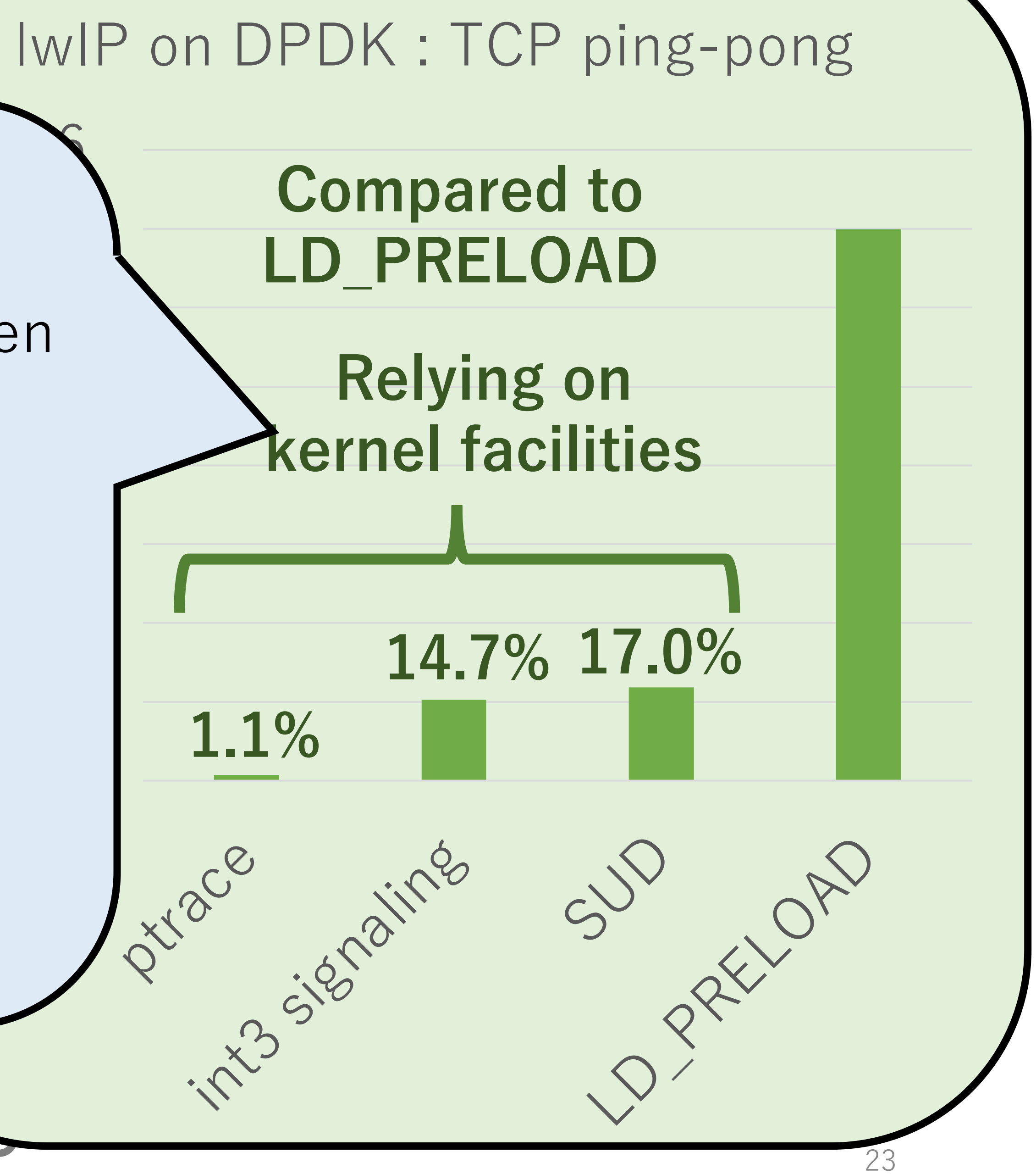
Relying on kernel facilities

system call hook

Kernel-space OS subsyste

Problem

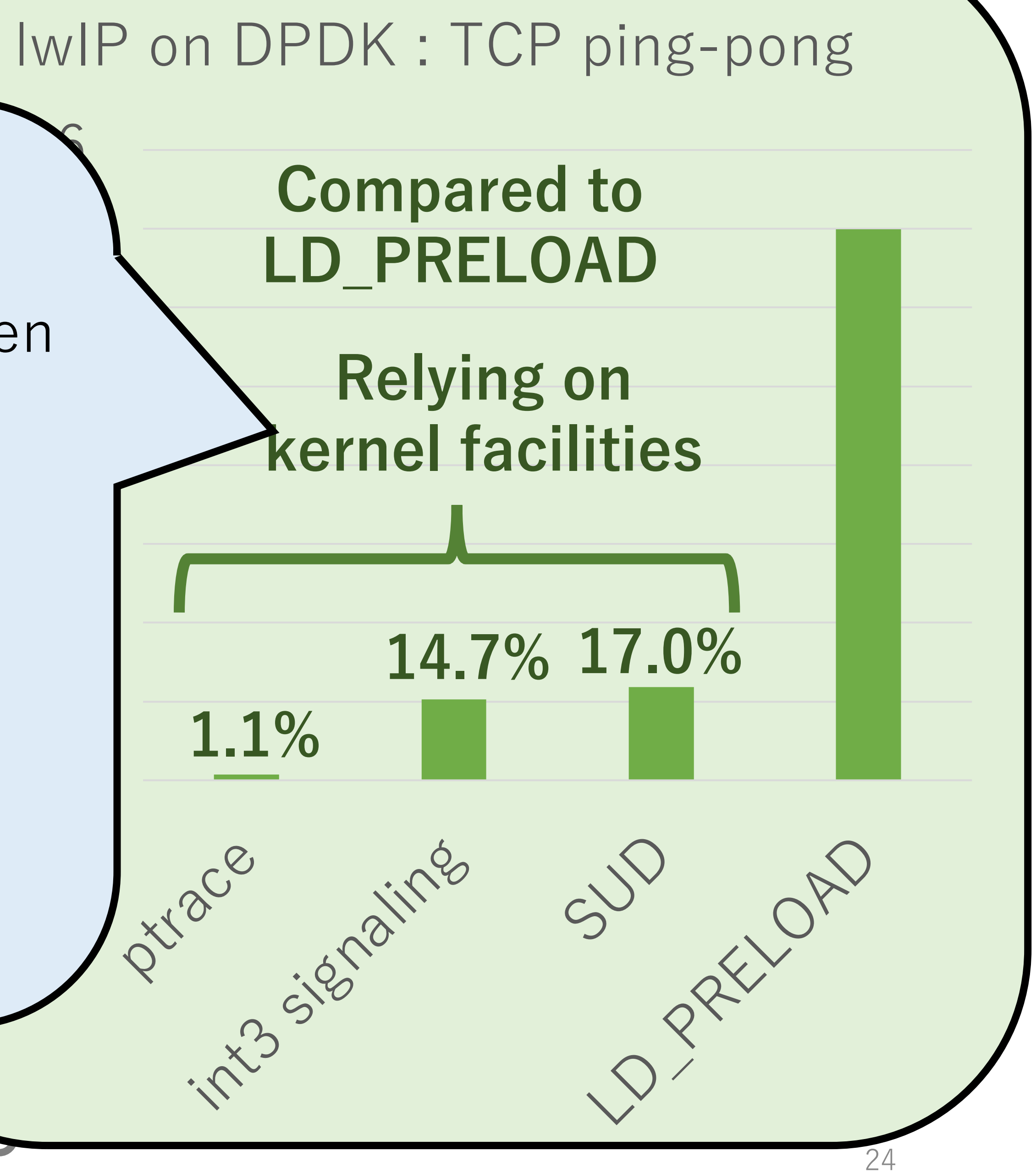
- ptrace
  - overhead: process scheduling between the tracer and tracee processes



Kernel-space OS subsyste

Problem

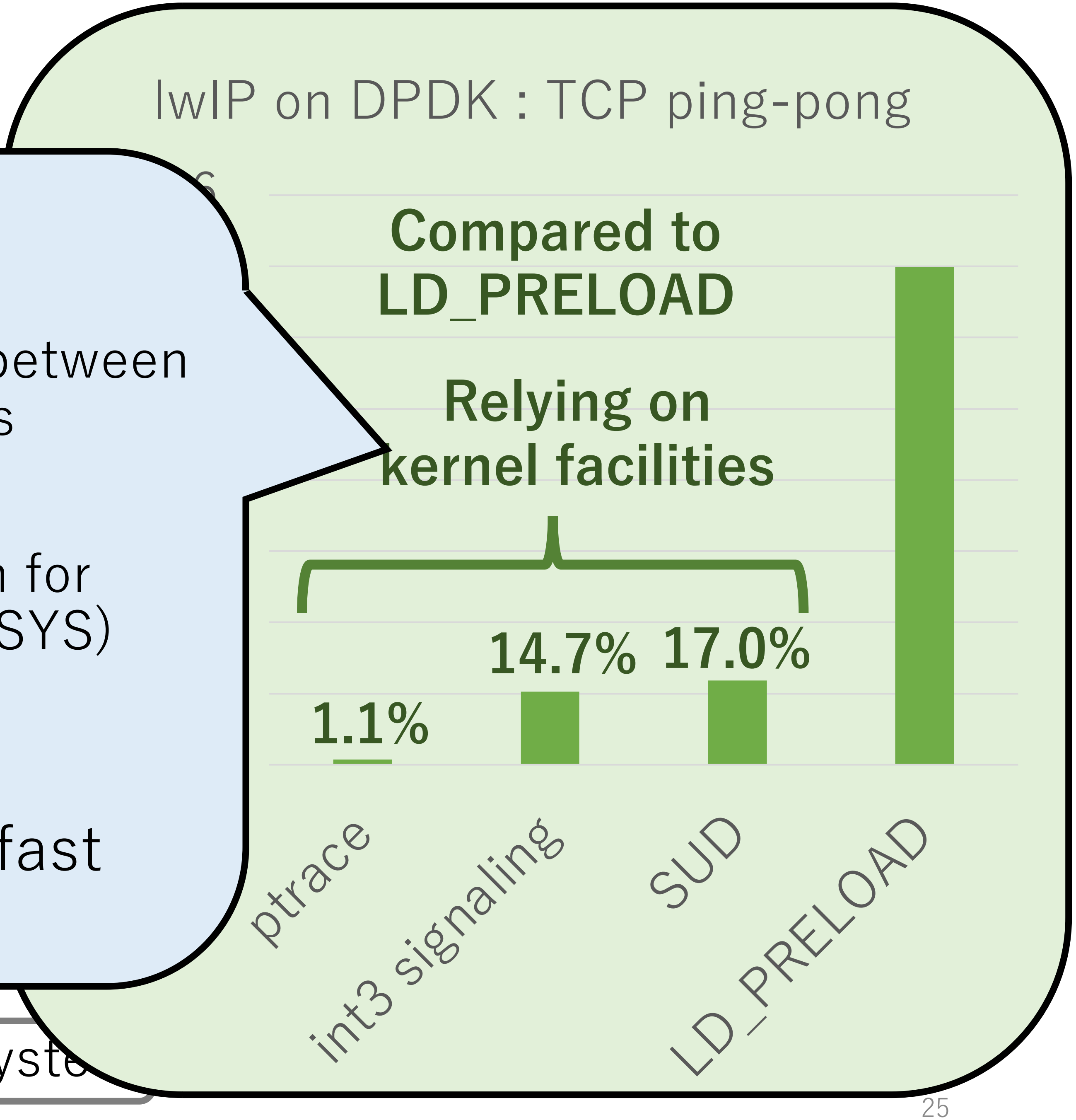
- ptrace
  - overhead: process scheduling between the tracer and tracee processes
- int3 signaling / SUD
  - overhead: context manipulation for a signal() handler (SIGINT/SIGSYS)



Kernel-space OS subsyste

Problem

- ptrace
  - overhead: process scheduling between the tracer and tracee processes
- int3 signaling / SUD
  - overhead: context manipulation for a signal() handler (SIGINT/SIGSYS)
- LD_PRELOAD just replaces function calls, therefore, it is fast



lwIP on DPDK : TCP ping-pong

Compared to LD_PRELOAD

Relying on kernel facilities

Kernel-space OS subsyste

Prob

- ptrac
  - ove
    the
- int3 s
  - ove
    a s
- LD_P
  funct

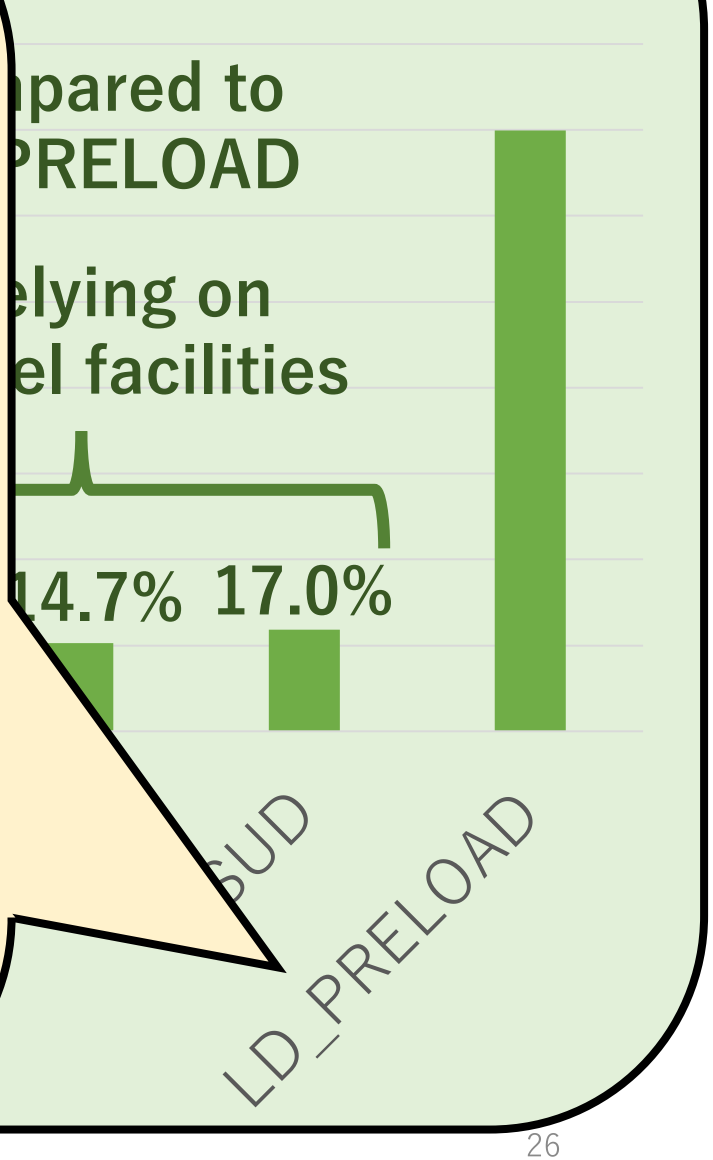
```
app_function(...)
{
  …
  write(...)
  …
}
```

libc
write()
library call

K : TCP ping-pong

pared to
PRELOAD

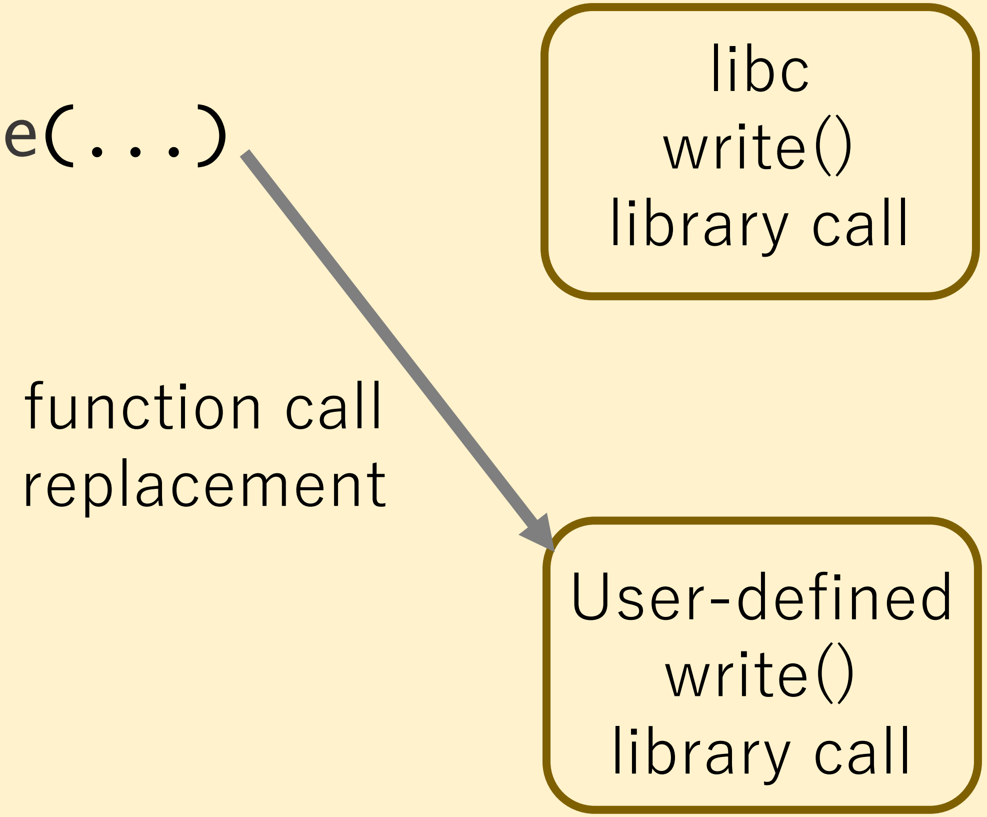
elying on
el facilities



Prob

- ptrac
  - ove
    the
- int3 s
  - ove
    a s
- LD_P
  funct

```
app_function(...)
{
  …
  write(...)
  …
}
```



function call
replacement

libc
write()
library call

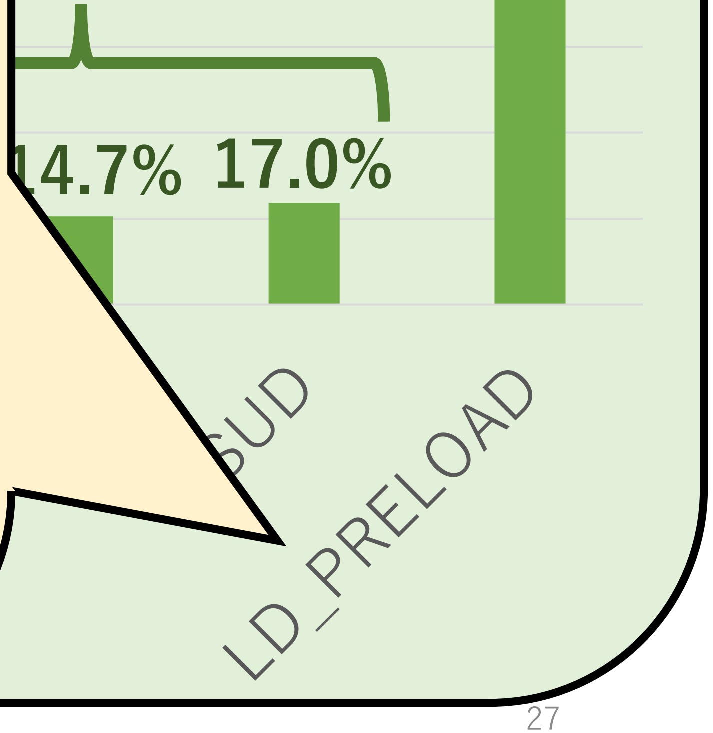
User-defined
write()
library call

LD_PRELOAD

DK : TCP ping-pong

mpared to
PRELOAD

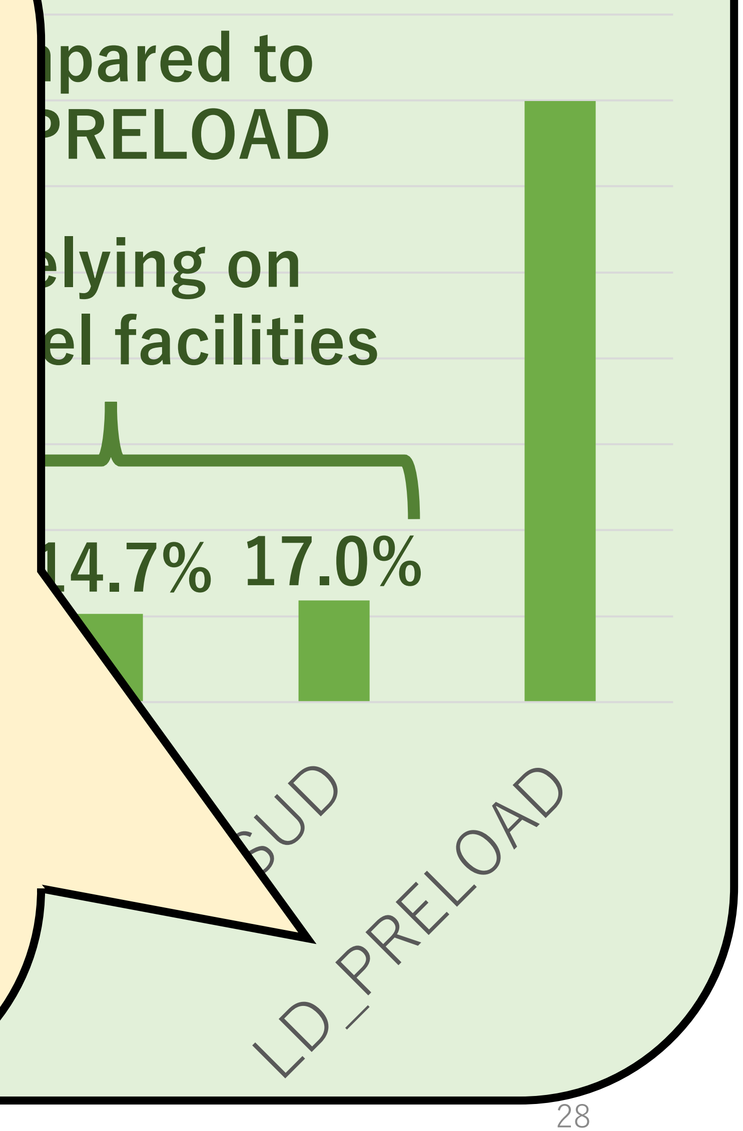
elying on
el facilities



Prob

- ptrac
  - ove
    the
- int3 s
  - ove
    a s
- LD_P
  funct

...K : TCP ping-pong



```
app_function(...)
{
   …
   special_write(...)
   …
}
```

libc
write()
library call

User-defined
write()
library call

LD_PRELOAD

Prob

- ptrac
  - ove
    the
- int3 s
  - ove
    a s
- LD_P
  funct

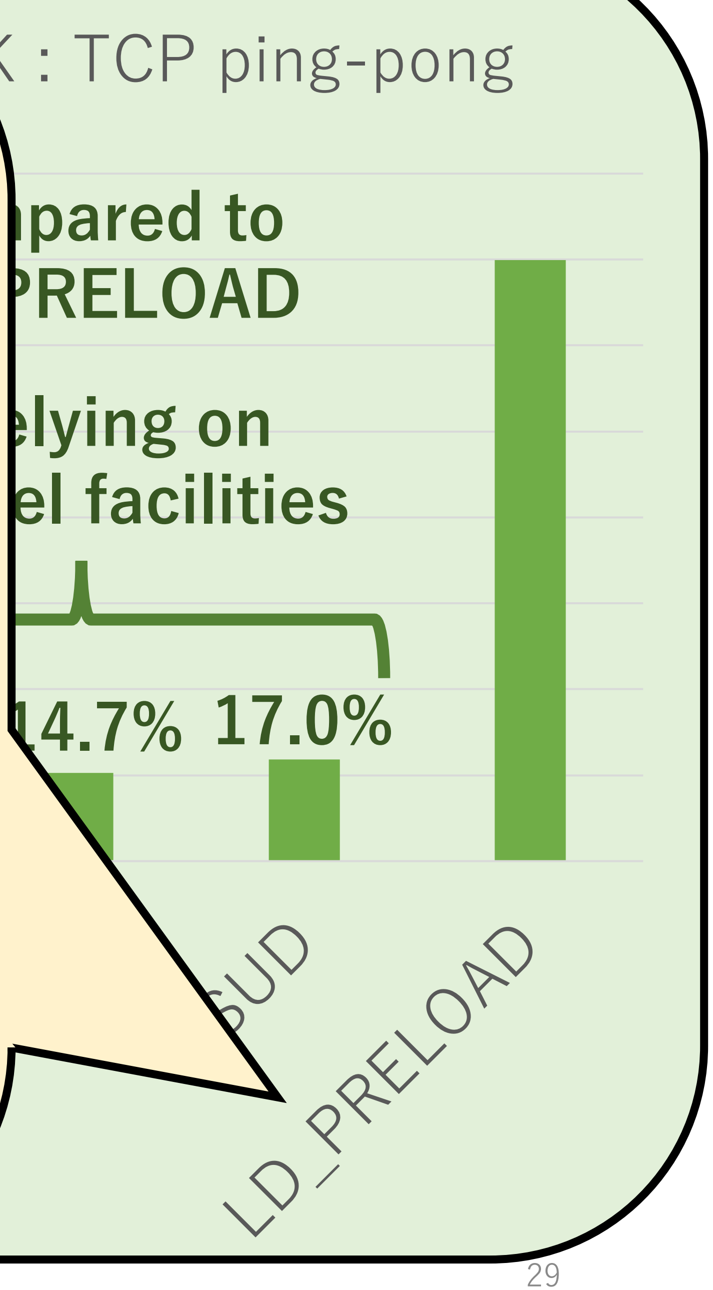
```
app_function(...)
{
  …
  special_write(...)
  …
}

special_write(...)
{
  asm volatile (
   trigger
   write syscall
  )
}
```

libc
write()
library call

User-defined
write()
library call

LD_PRELOAD



Prob

- ptrac
  - ove the
- int3 s
  - ove a s
- LD_P funct

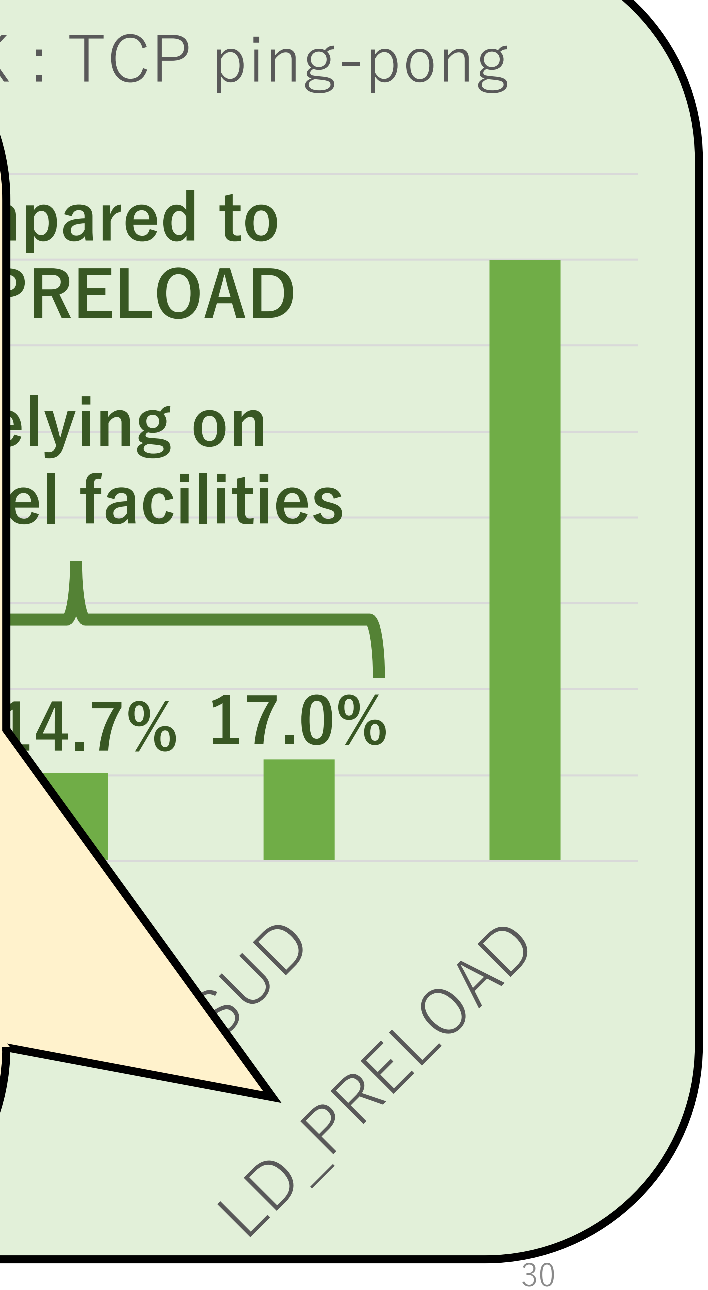
```
app_function(...)
{
   …
   special_write(...)
   …
}

special_write(...)
{
   asm volatile (
    trigger
    write syscall
   )
}
```

libc write() library call

User-defined write() library call

LD_PRELOAD



Prob

- ptrac
  - ove the
- int3 s
  - ove a s
- LD_P funct

```
app_function(...)
{
  …
  special_write(...)
  …
}
```

libc write() library call

```
special_write(...)
{
  asm volatile (
   trigger
   write syscall
  )
}
```

Hook is not applied

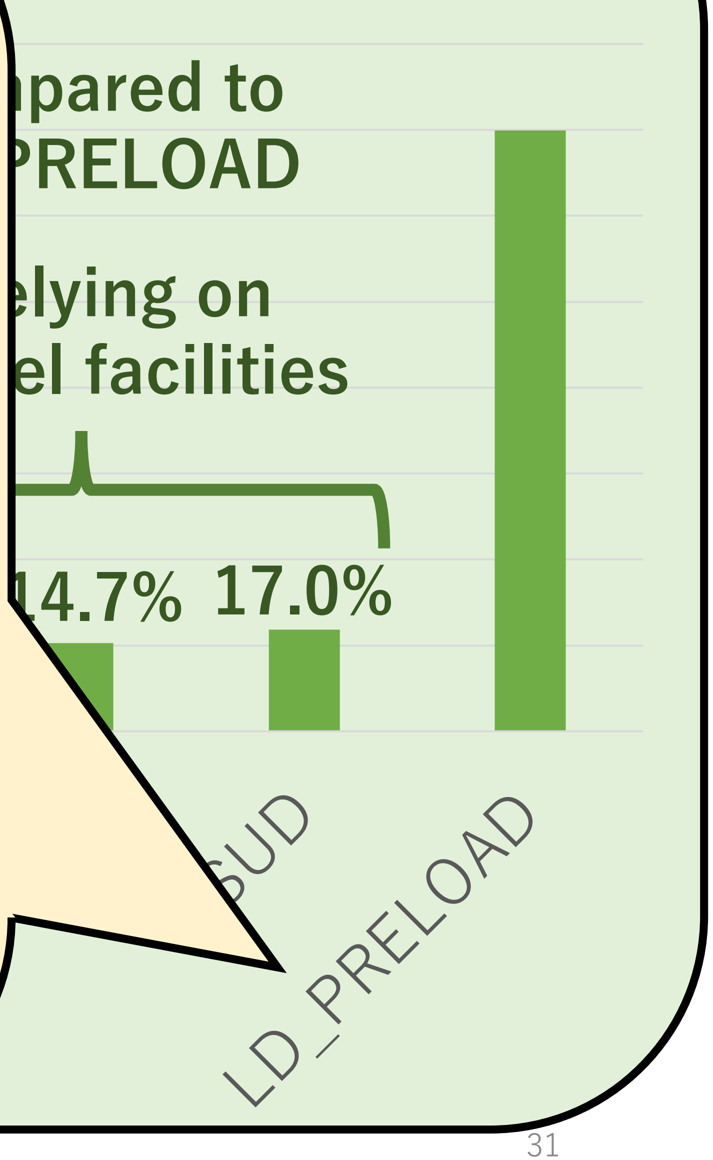
User-defined write() library call

LD_PRELOAD

DK : TCP ping-pong

pared to PRELOAD

elying on el facilities



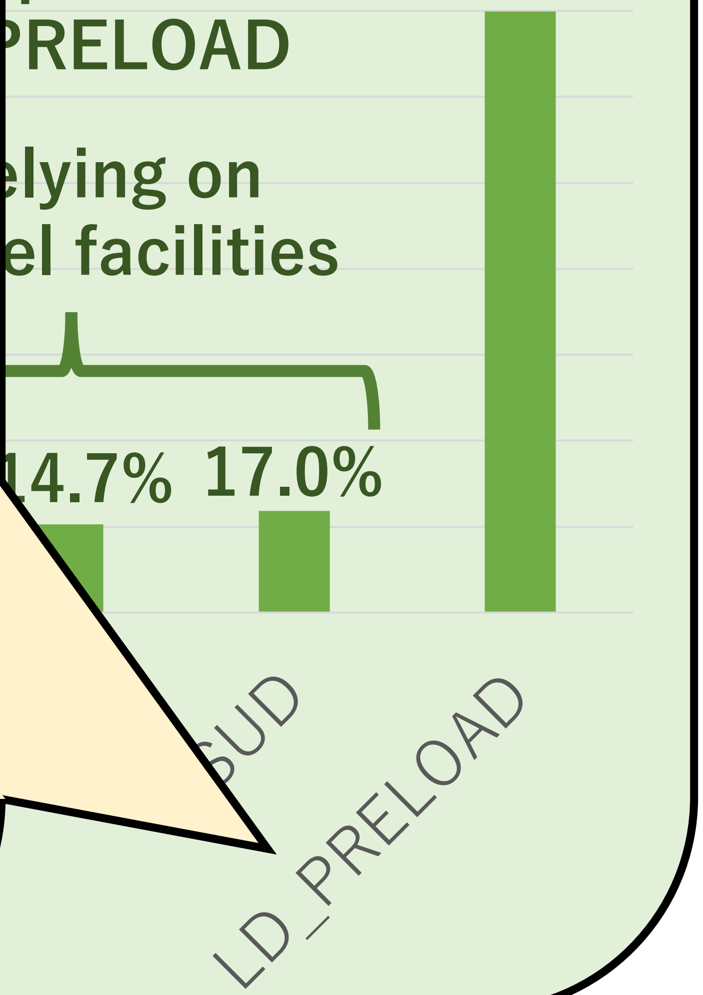
Prob

- ptrac
  - ove the
- int3 s
  - ove a s
- LD_P funct

DK : TCP ping-pong

pared to PRELOAD

elying on el facilities



```
app_function(...)
{
   …
   special_write(...)
   …
}

special_write(...)
{
   asm volatile (
    trigger
    write syscall
   )
}
```

libc
write()
library call

because names are different

User-defined
write()
library call

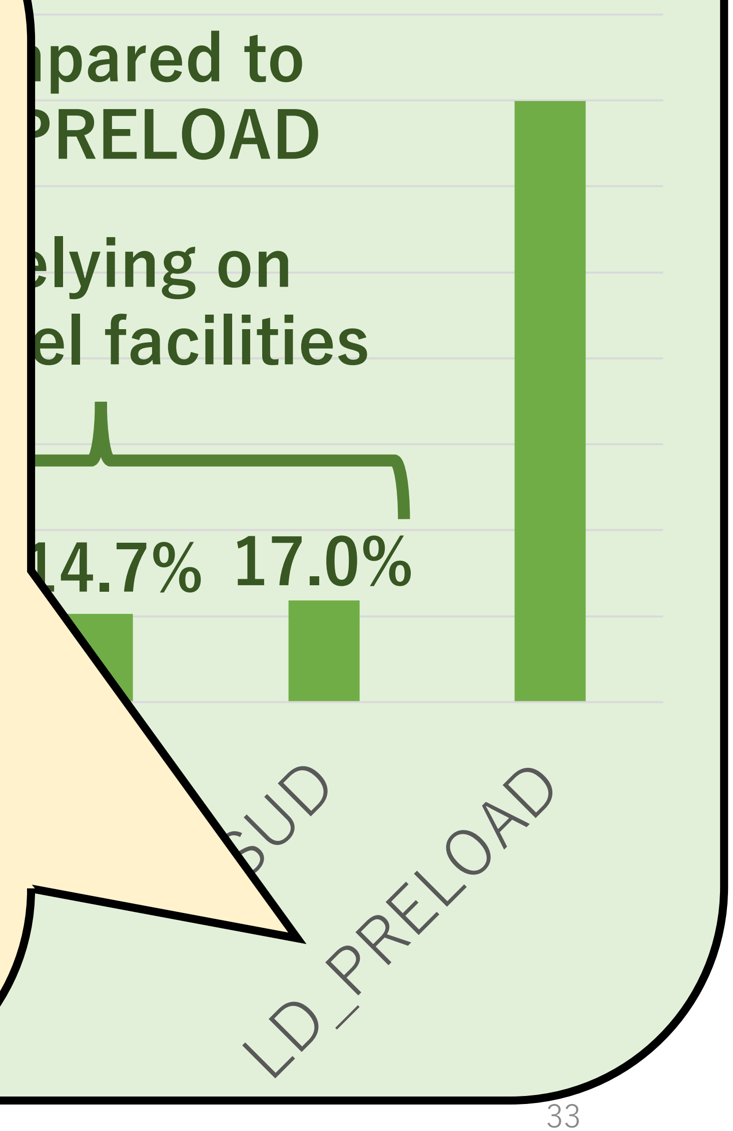
Hook is not applied

LD_PRELOAD

Prob

- ptrac
  - ove
    the
- int3 s
  - ove
    a s
- LD_P
  funct

K : TCP ping-pong



We see this case in glibc often

```
app_function(...)
{
  …
  special_write(...)
  …
}

special_write(...)
{
  asm volatile (
   trigger
   write syscall
  )
}
```

libc
write()
library call

because names
are different

User-defined
write()
library call

LD_PRELOAD

Hook is not applied

Problem

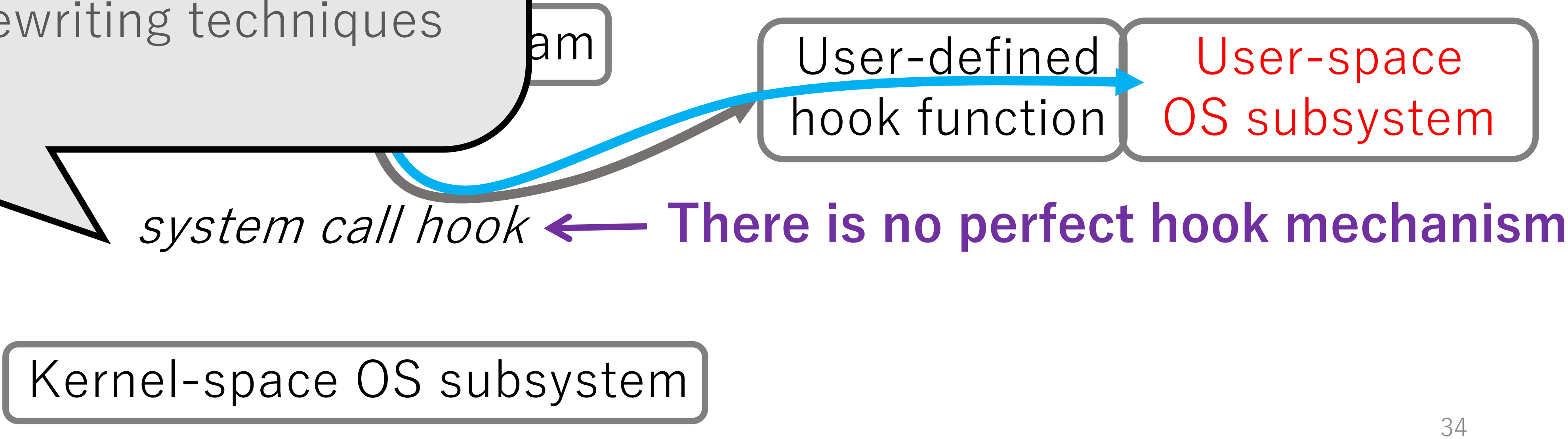
<u>Existing Mechanisms</u>

- ptrace
- int3 signaling technique
- Syscall User Dispatch (SUD)
- LD_PRELOAD trick
- Binary rewriting techniques
- …

ms allow us to transparently apply
to existing applications

ok mechanism, …

user-space program is necessary



There is no perfect hook mechanism

Problem

ms allow us to transparently apply
to existing applications

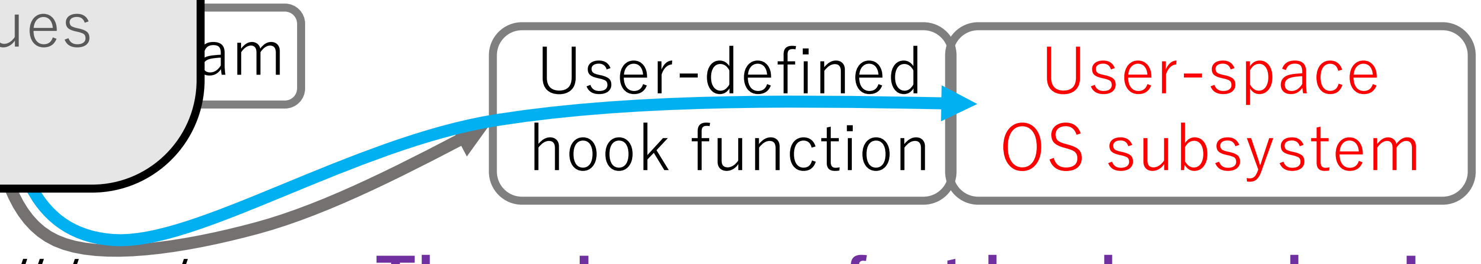
ok mechanism, ...

user-space program is necessary

Existing Mechanisms

High performance penalty

- int3 sig... ...hnique
- Syscall User Disp... ...D)
- LD_PRELOAD trick
- Binary rewriting techniques
- …



system call hook ← **There is no perfect hook mechanism**

Kernel-space OS subsystem

Problem

Existing Mechanisms

- int3 sig... ...chnique
- Syscall User Disp... ...D)
- ...ELOAD trick
- Binary re... ...chniques
- …

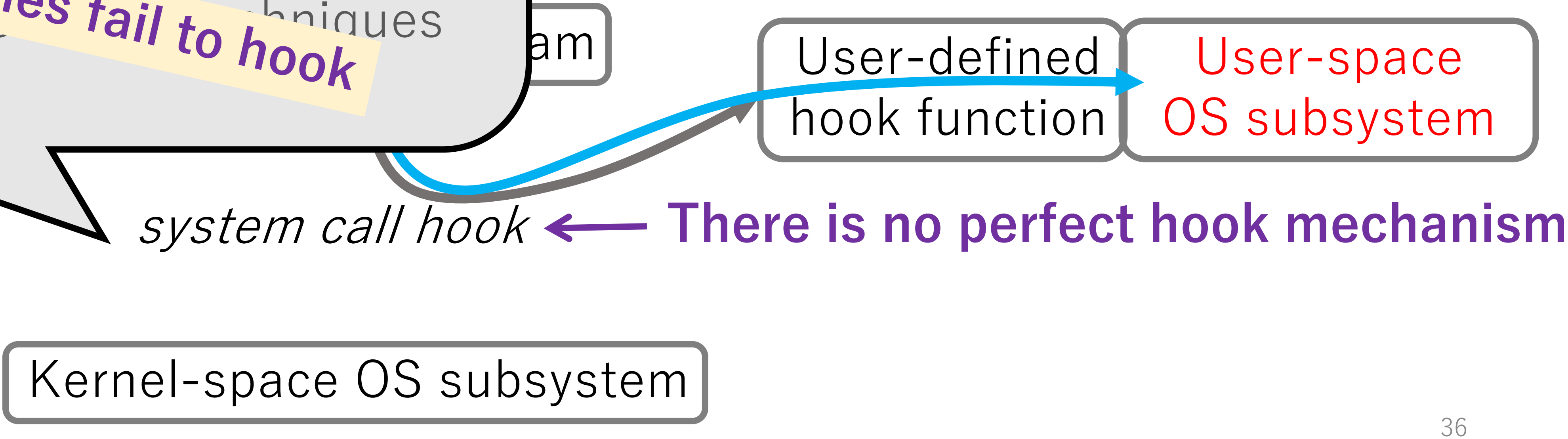
High performance penalty

Sometimes fail to hook

...ms allow us to transparently apply
... to existing applications

...ok mechanism, …

... user-space program is necessary



There is no perfect hook mechanism

Problem ➡ Applicability of user-space OS subsystems has been limited regardless of their benefits

ms allow us to transparently apply
to existing applications

ok mechanism, ...

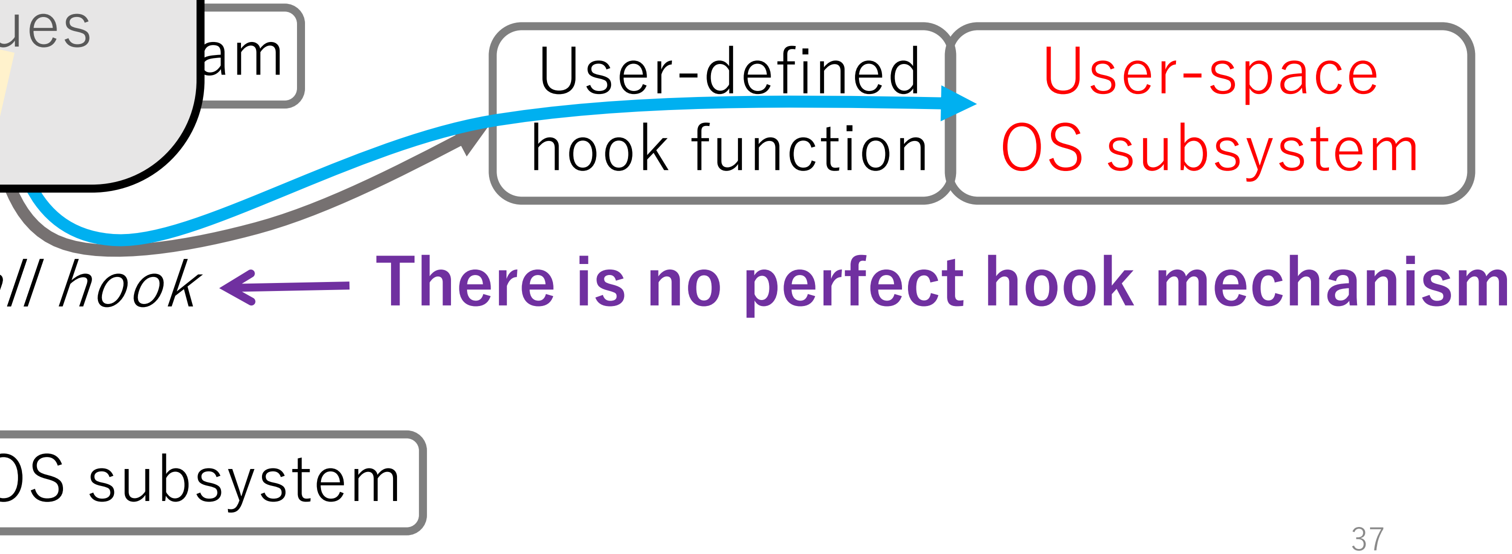
user-space program is necessary

Existing Mechanisms

- int3 sig chnique
- Syscall User Disp D)
- RELOAD trick
- Binary re chniques
- …

High performance penalty

Sometimes fail to hook



There is no perfect hook mechanism

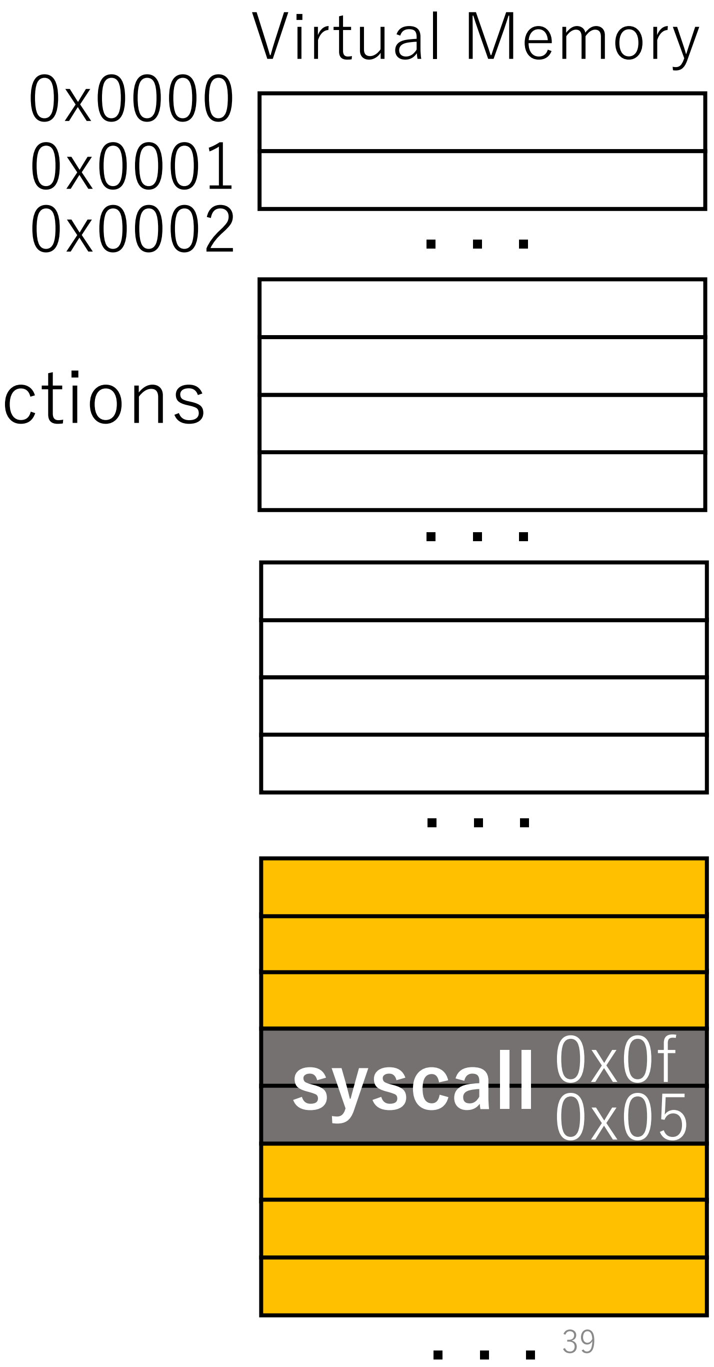
Contribution

- zpoline: a system call hook mechanism for x86-64 CPUs
    - based on binary rewriting
    - free from the drawbacks of the pervious mechanisms

- This work addresses a challenge that is specific to binary rewriting approaches

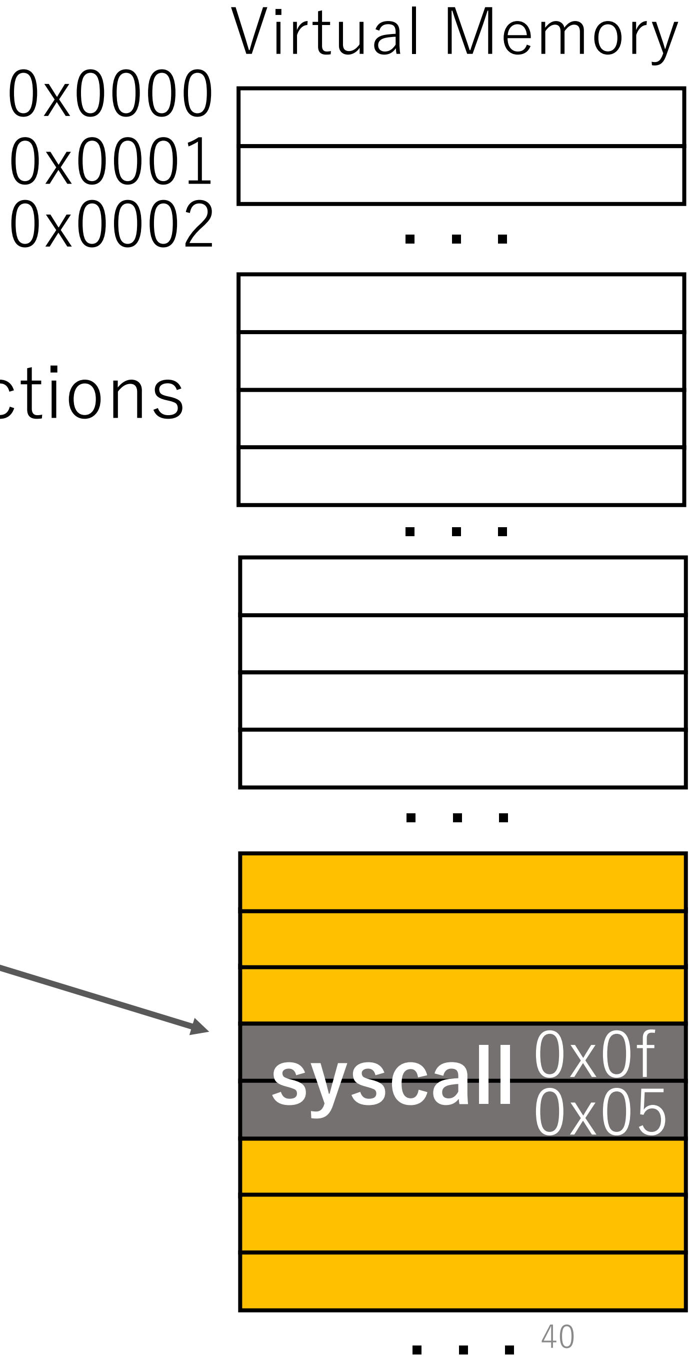
Binary Rewriting Approach

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34



Binary Rewriting Approach

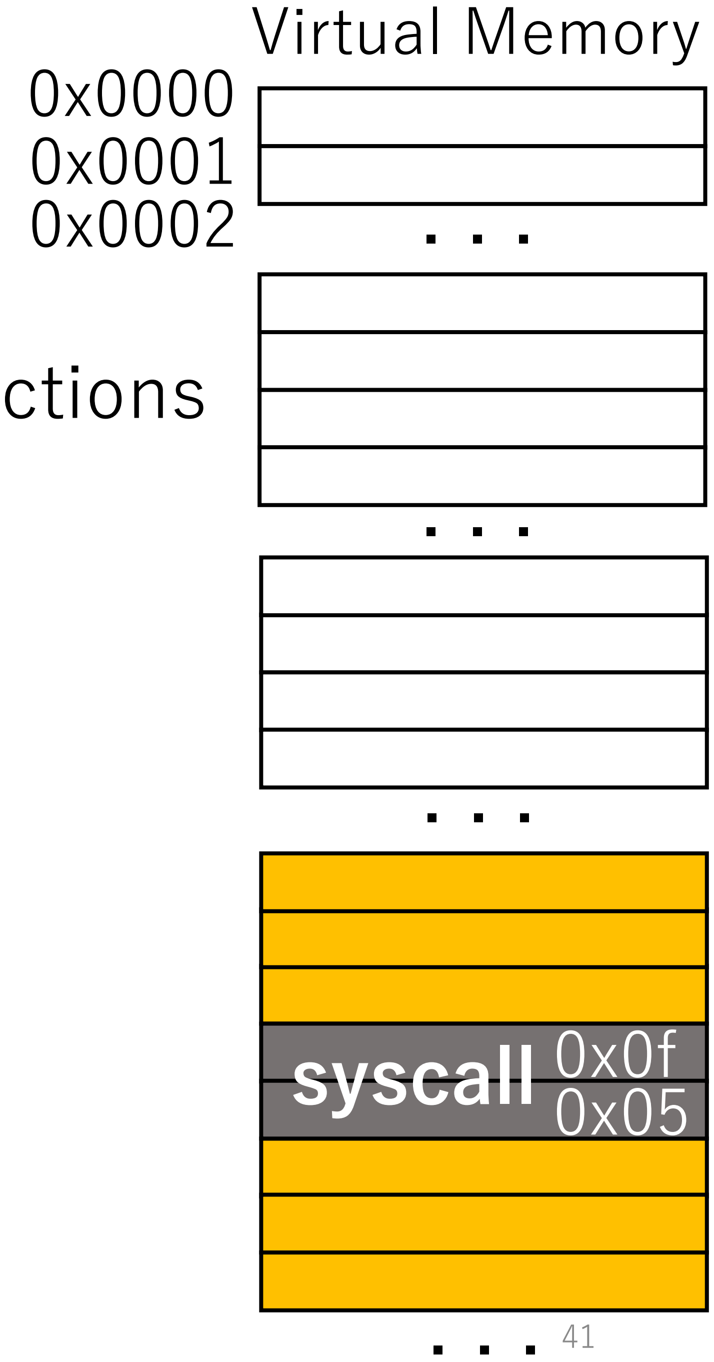
- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34



Binary Rewriting Approach

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34

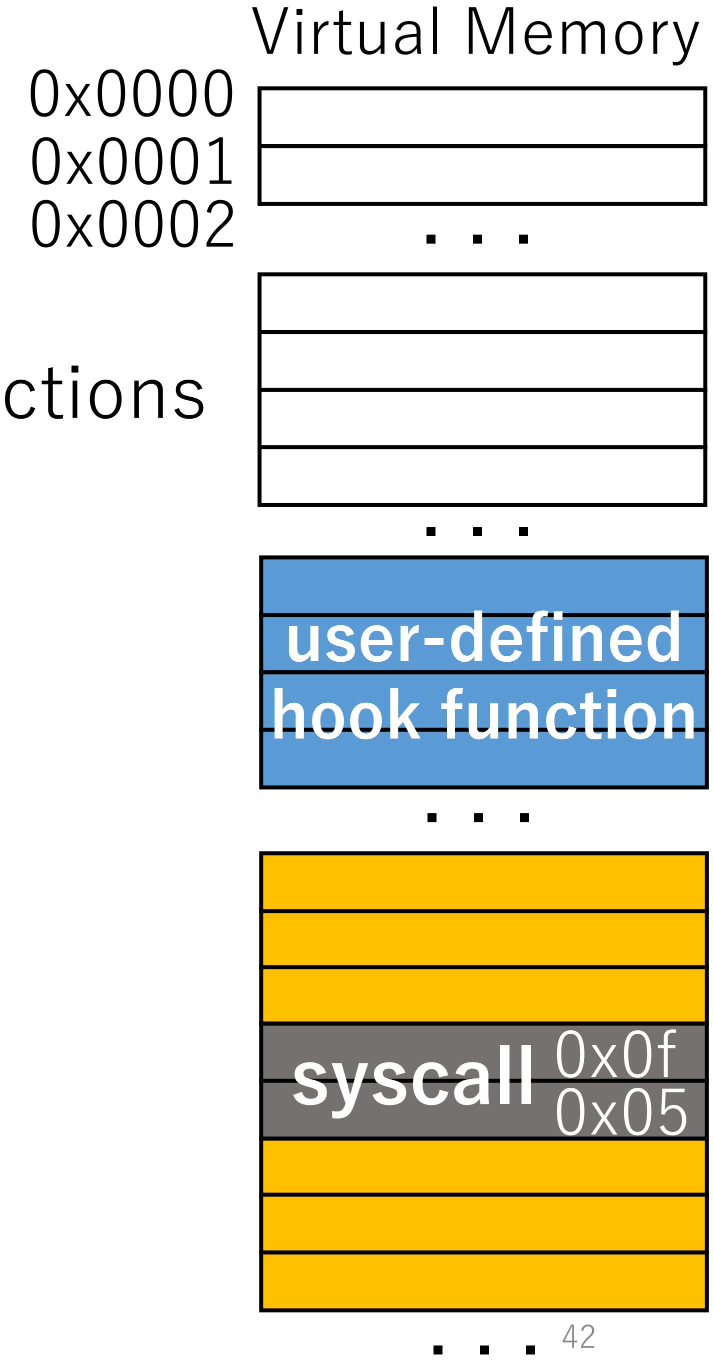
- What we wish to achieve



Binary Rewriting Approach

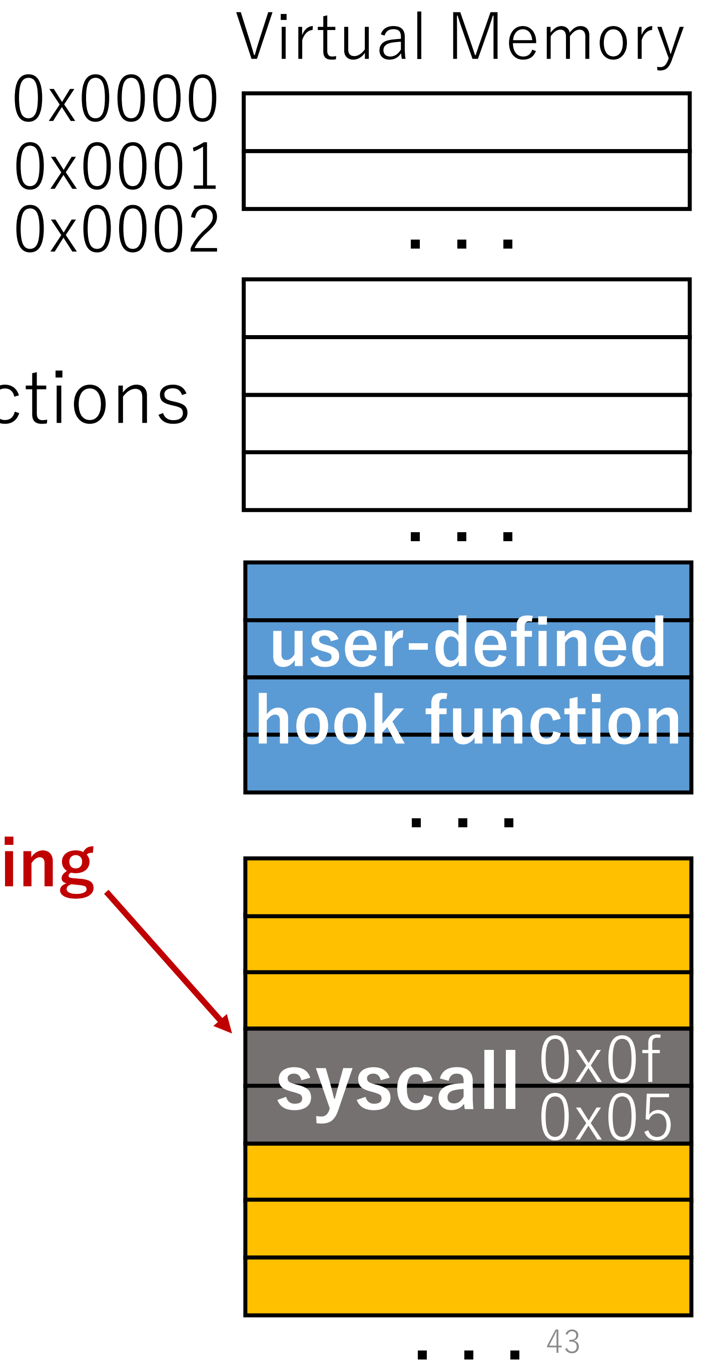
- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34

- What we wish to achieve



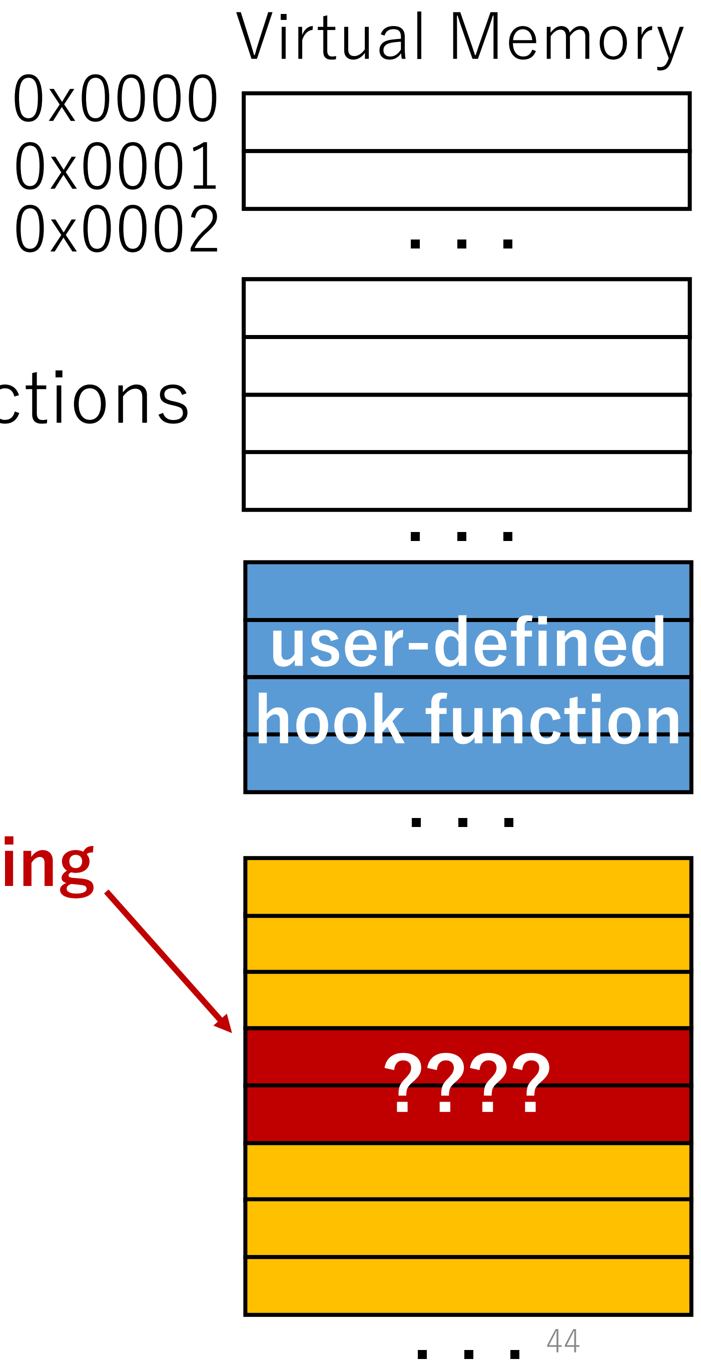
Binary Rewriting Approach

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34
- What we wish to achieve
  - replace syscall/sysenter instruction with **something**



Binary Rewriting Approach

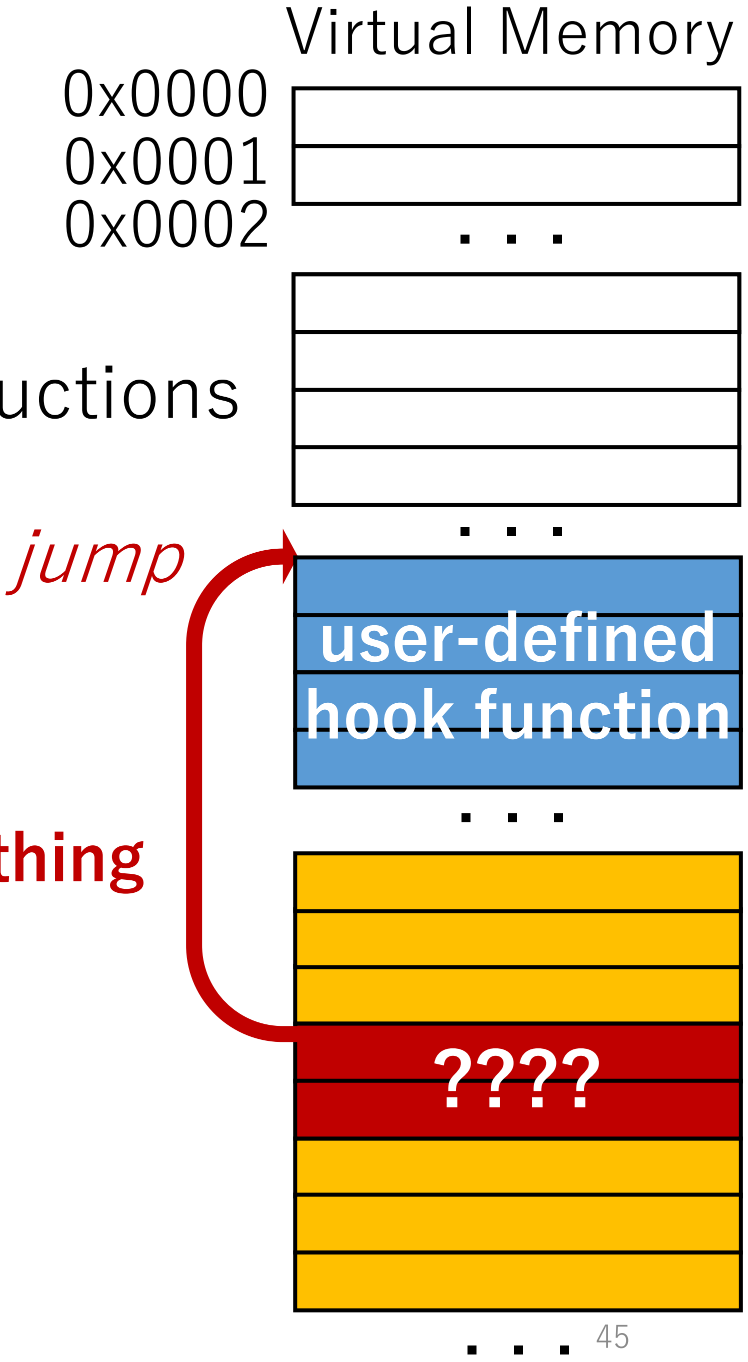
- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34
- What we wish to achieve
  - replace syscall/sysenter instruction with **something**



Binary Rewriting Approach

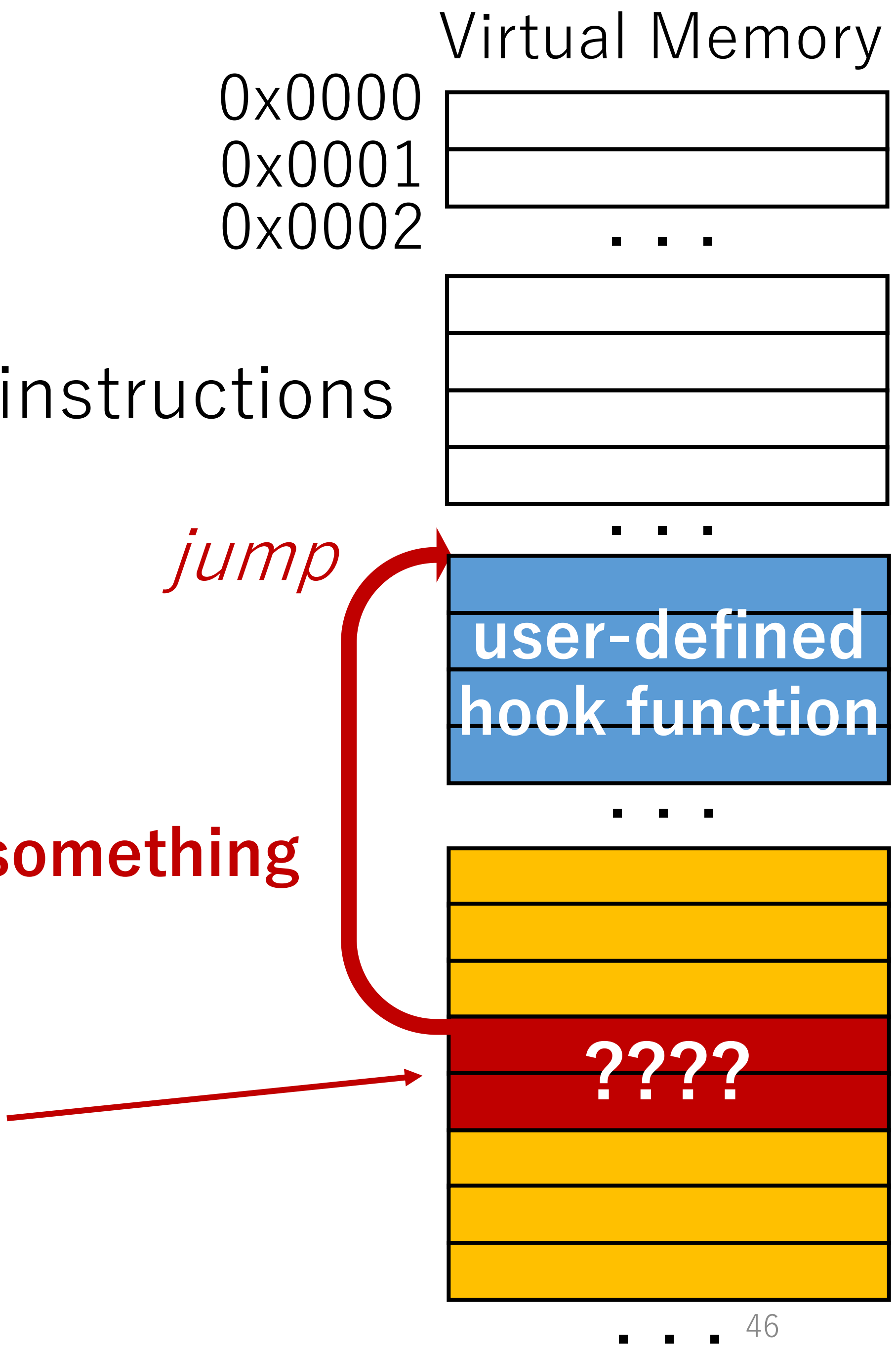
- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34

- What we wish to achieve
  - replace syscall/sysenter instruction with **something**
  - to jump to a user-defined hook function



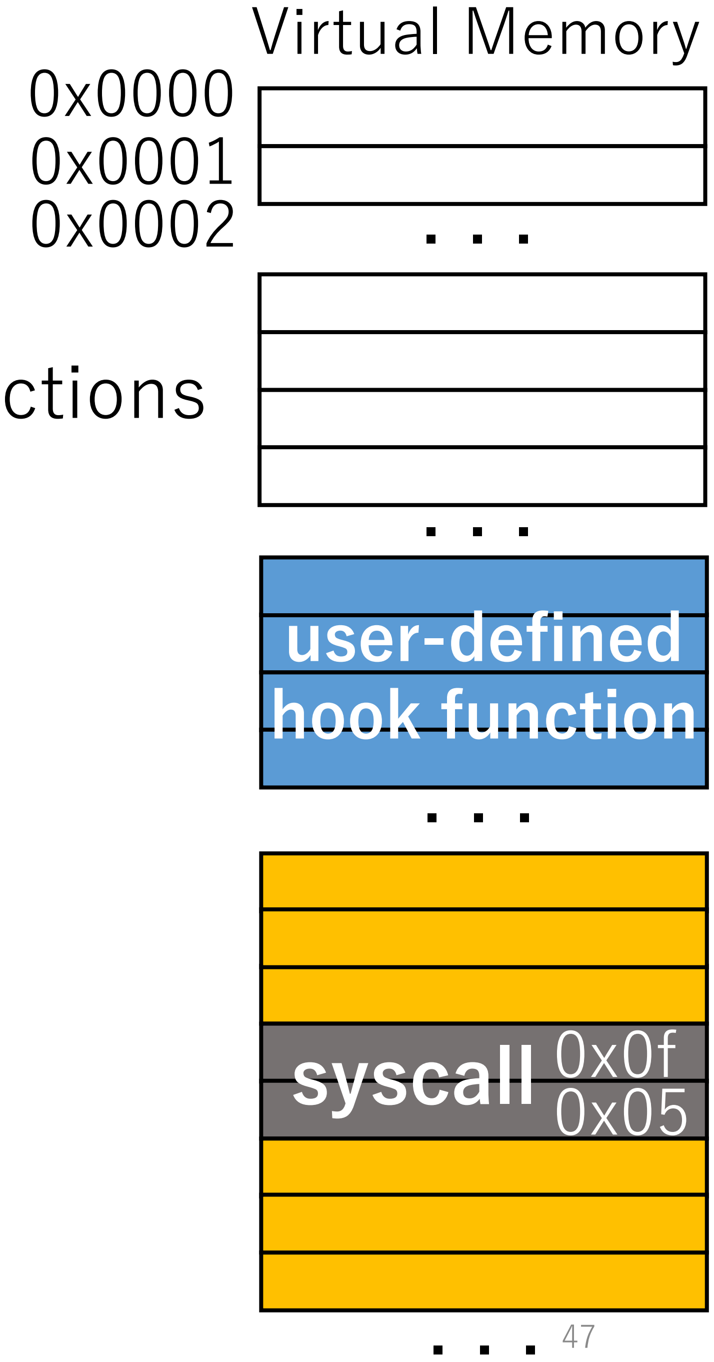
Binary Rewriting Approach

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34
- What we wish to achieve
  - replace syscall/sysenter instruction with **something**
  - to jump to a user-defined hook function
- **Question: what should we put here?**



Challenge

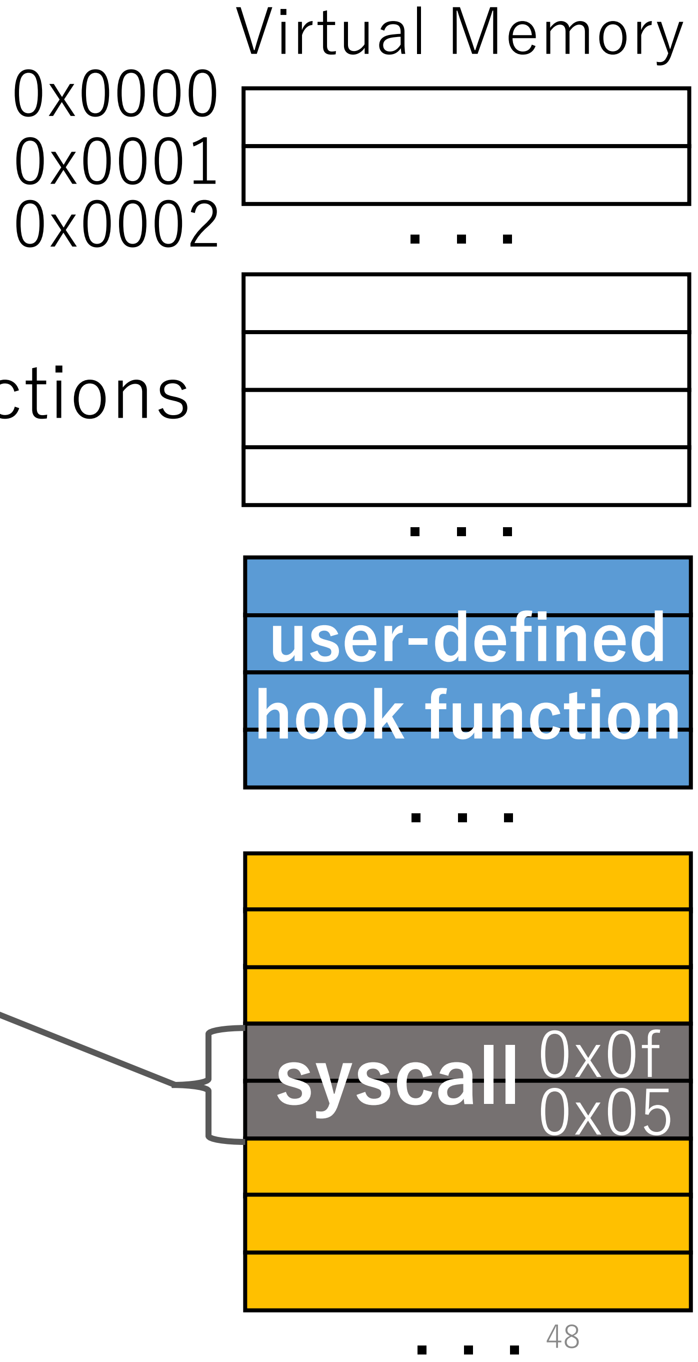
- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34



Challenge

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34

- syscall and sysenter are **2-byte** instructions

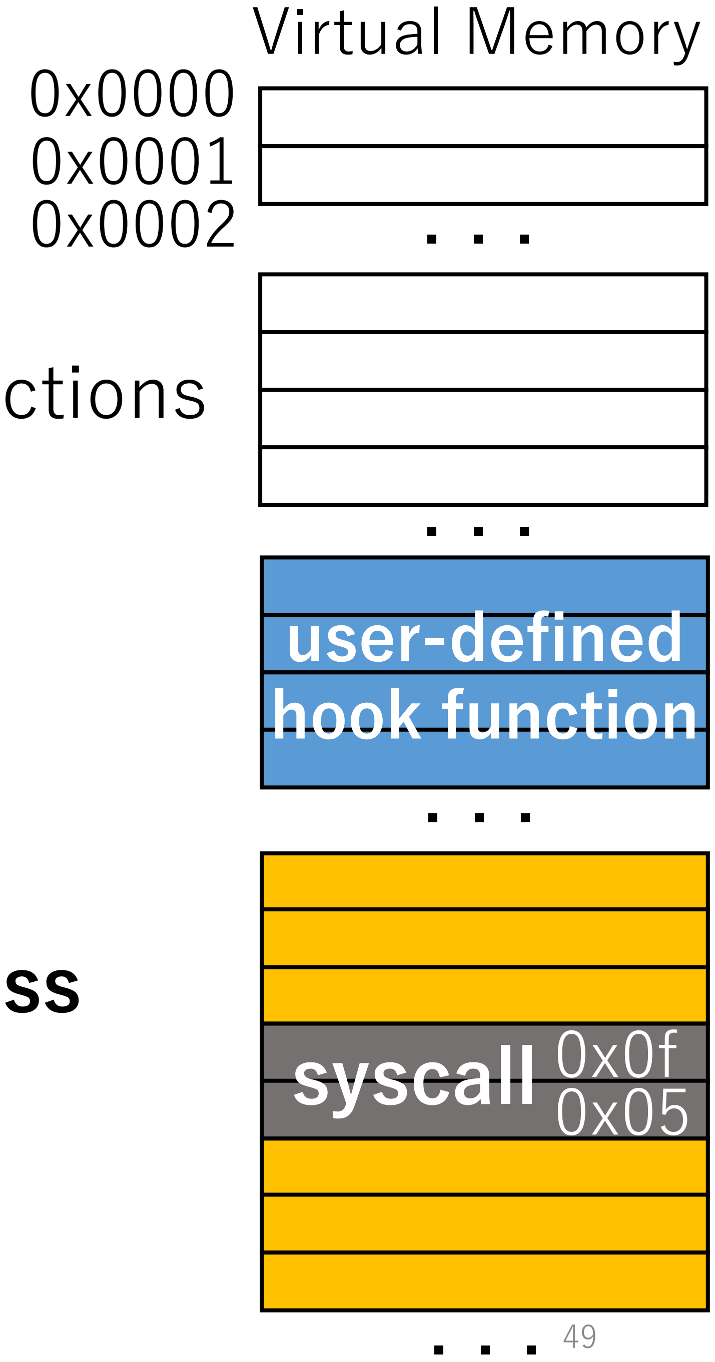


Challenge

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34

- syscall and sysenter are **2-byte** instructions

- **Specification for a jump destination address needs more than 2 bytes**

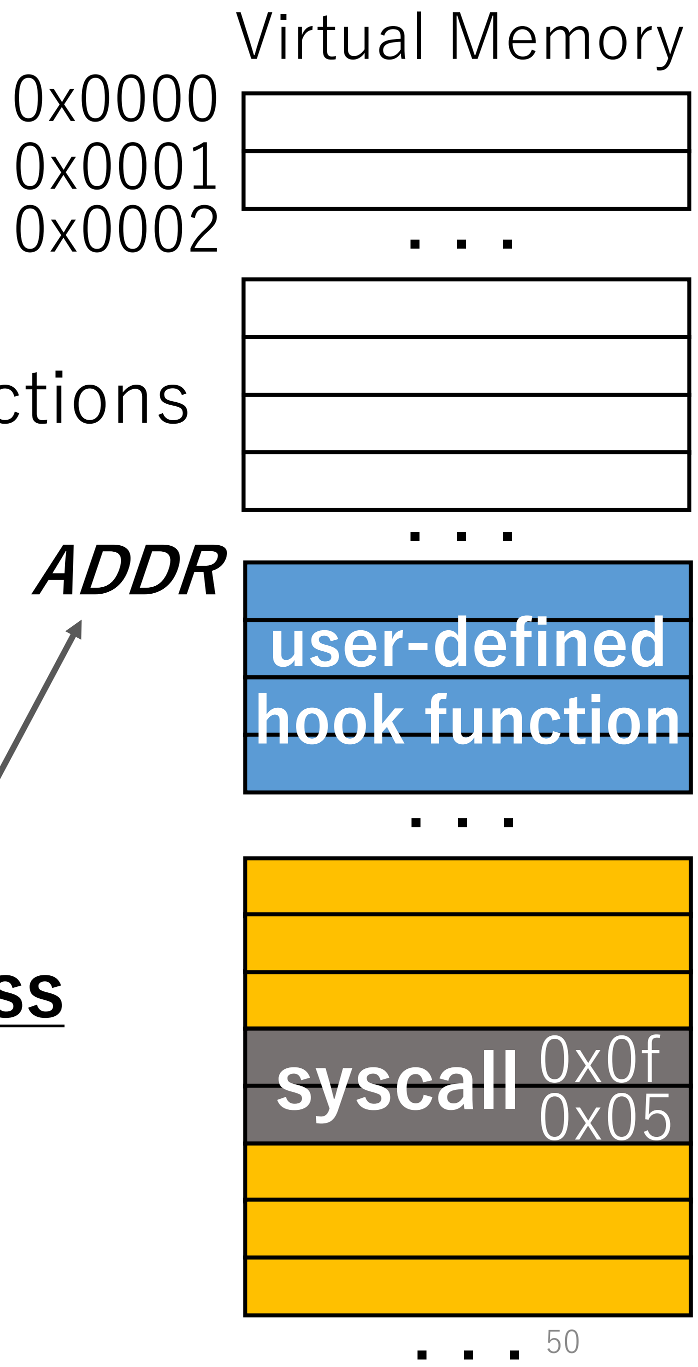


Challenge

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34

- syscall and sysenter are **2-byte** instructions

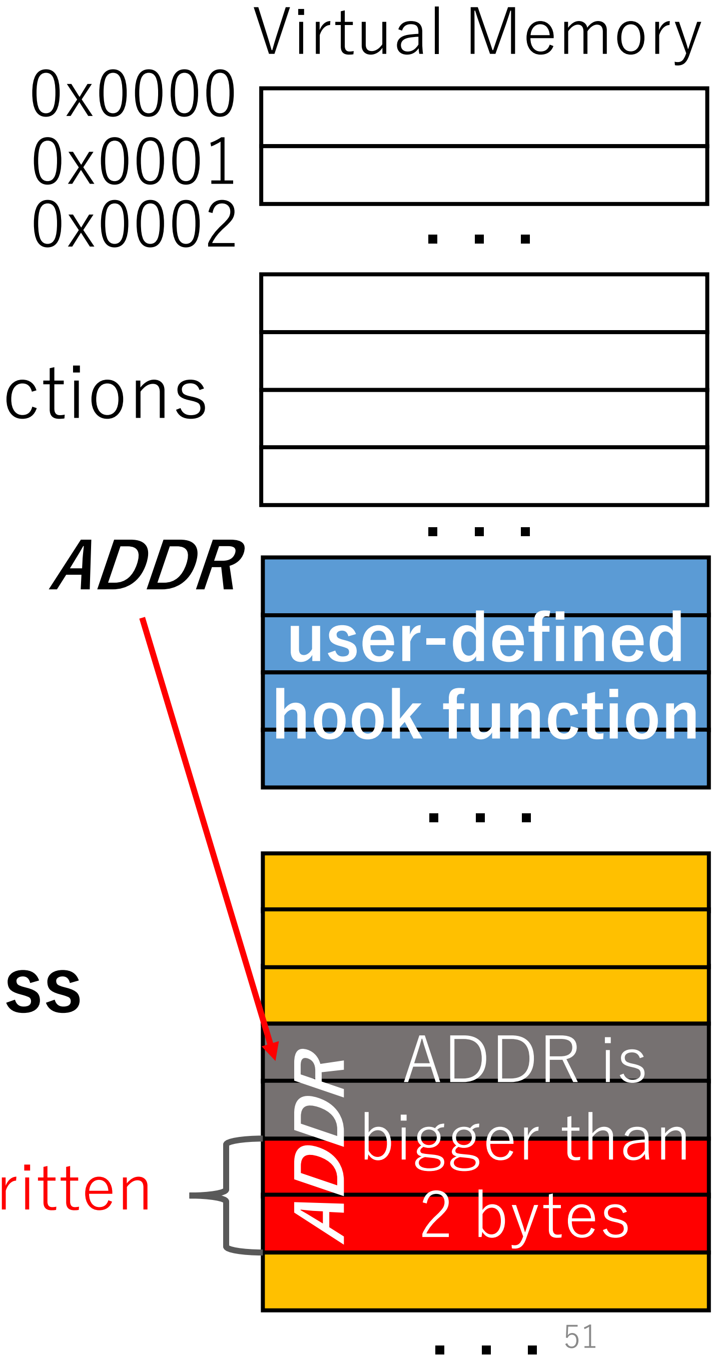
- **Specification for a jump destination address needs more than 2 bytes**



Challenge

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
    - syscall: 0x0f 0x05, sysenter: 0x0f 0x34
- syscall and sysenter are **2-byte** instructions
- **Specification for a jump destination address needs more than 2 bytes**

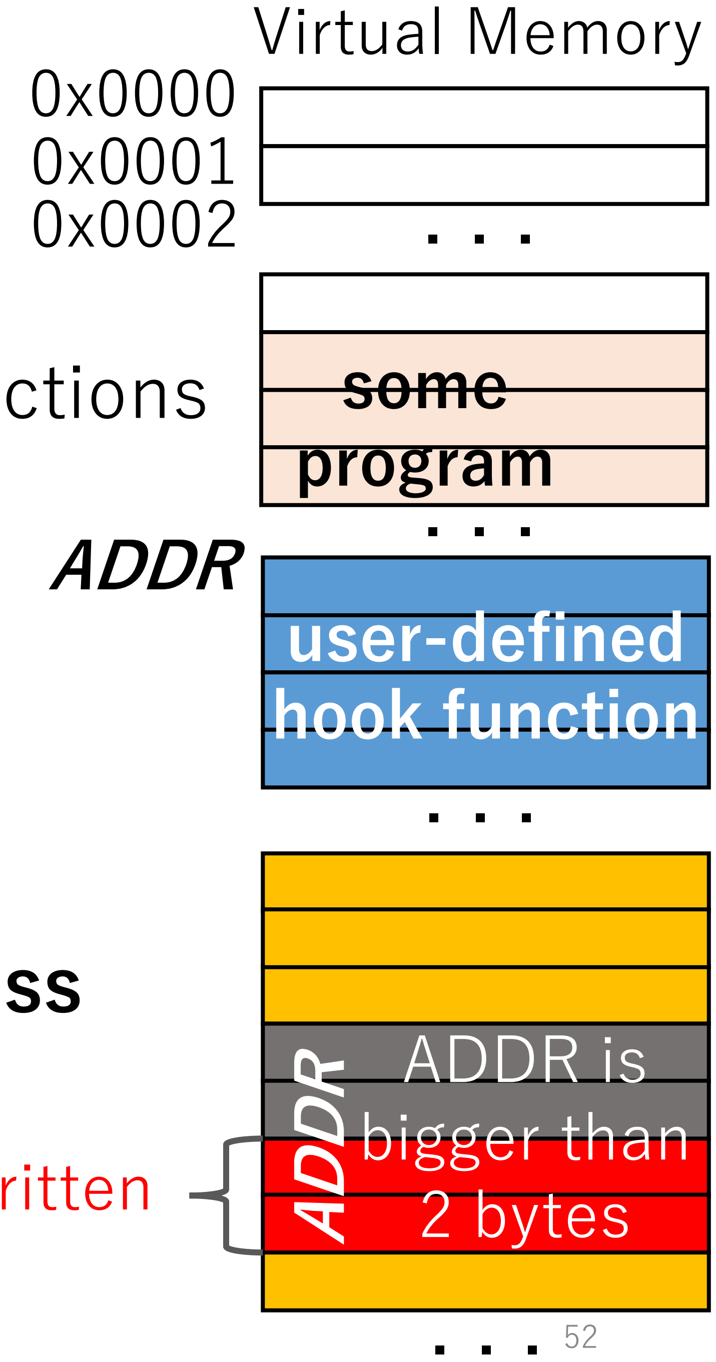
If we put ADDR, subsequent instructions are overwritten



Challenge

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34
- syscall and sysenter are **2-byte** instructions
- **Specification for a jump destination address needs more than 2 bytes**

If we put ADDR, subsequent instructions are overwritten



Challenge

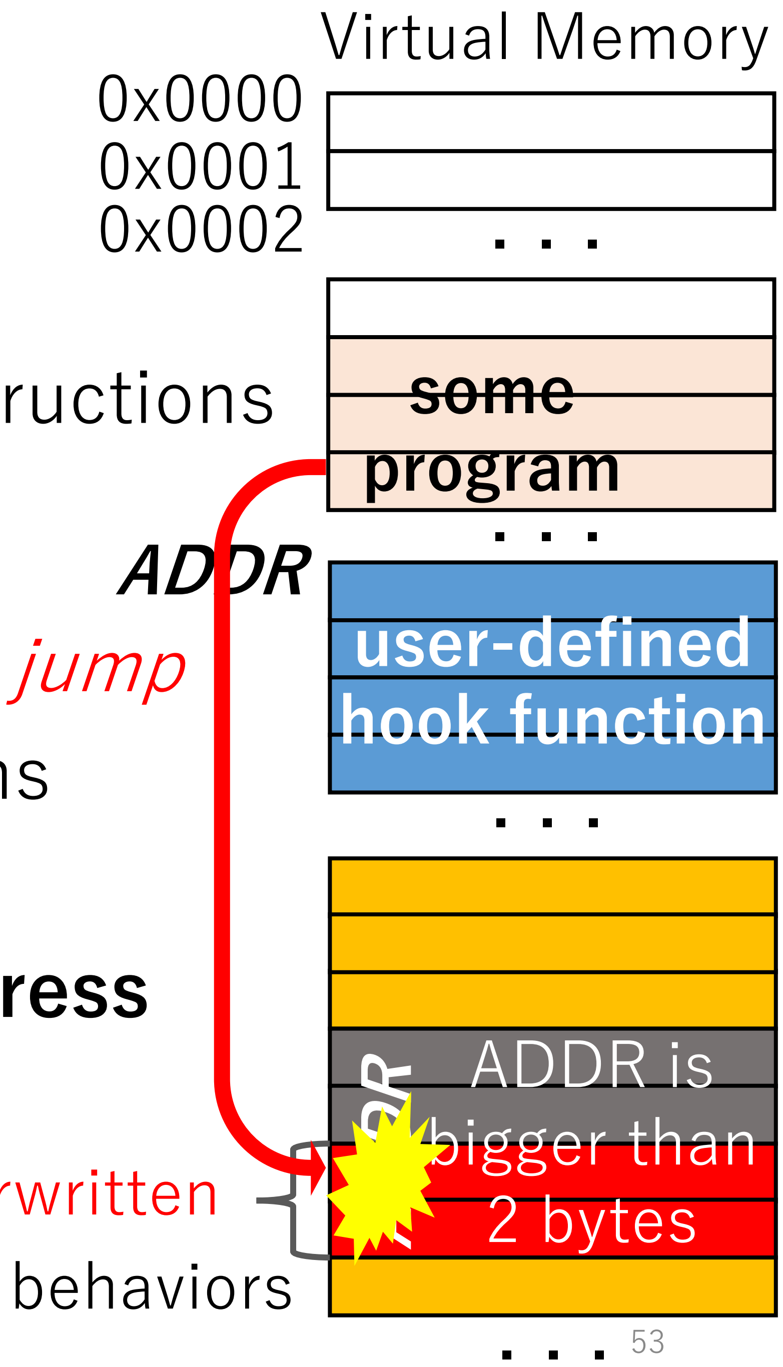
- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34

- syscall and sysenter are **2-byte** instructions

- **Specification for a jump destination address needs more than 2 bytes**

If we put ADDR, subsequent instructions are overwritten

jump to the overwritten part leads to unexpected behaviors



Challenge

- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34

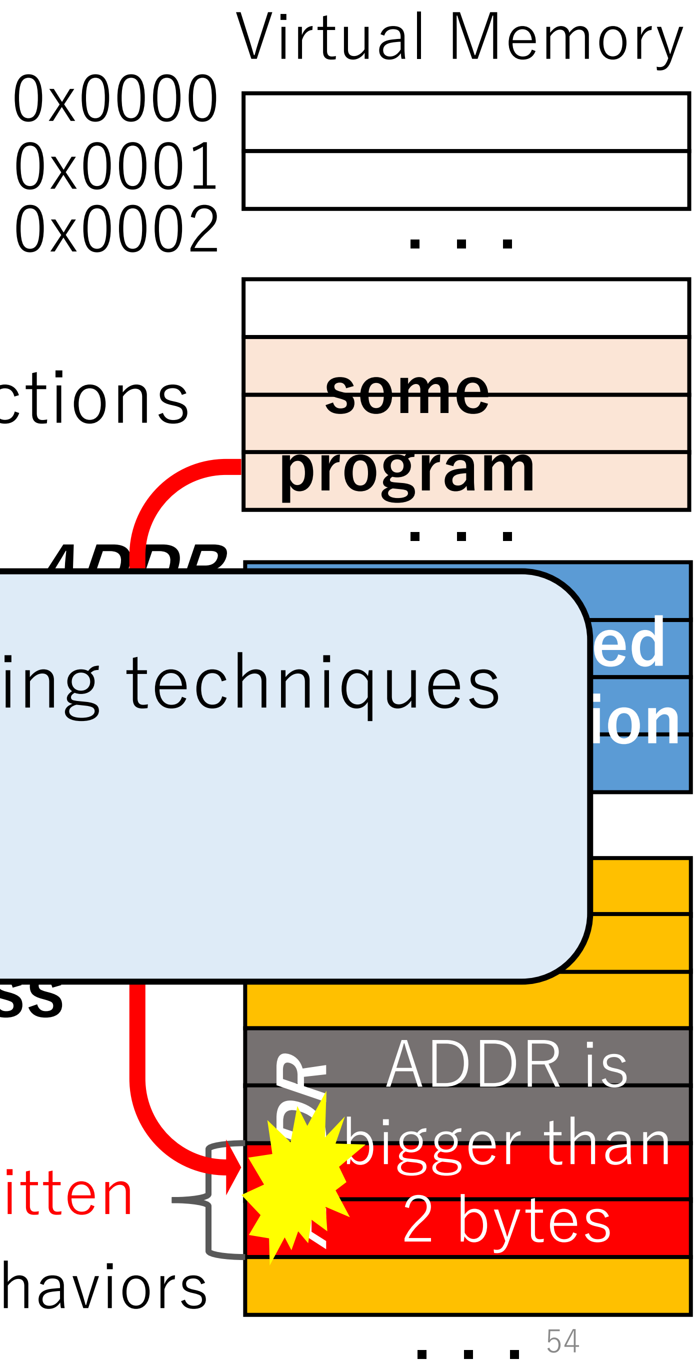
Because of this issue, previous binary rewriting techniques

- could not ensure exhaustive hooking
- or, overwrite neighbour instructions

- **Specification for a jump destination address needs more than 2 bytes**

If we put ADDR, subsequent instructions are overwritten

jump to the overwritten part leads to unexpected behaviors



Goal ➡ **jump to a function using only 2 bytes originally occupied by syscall/sysenter**

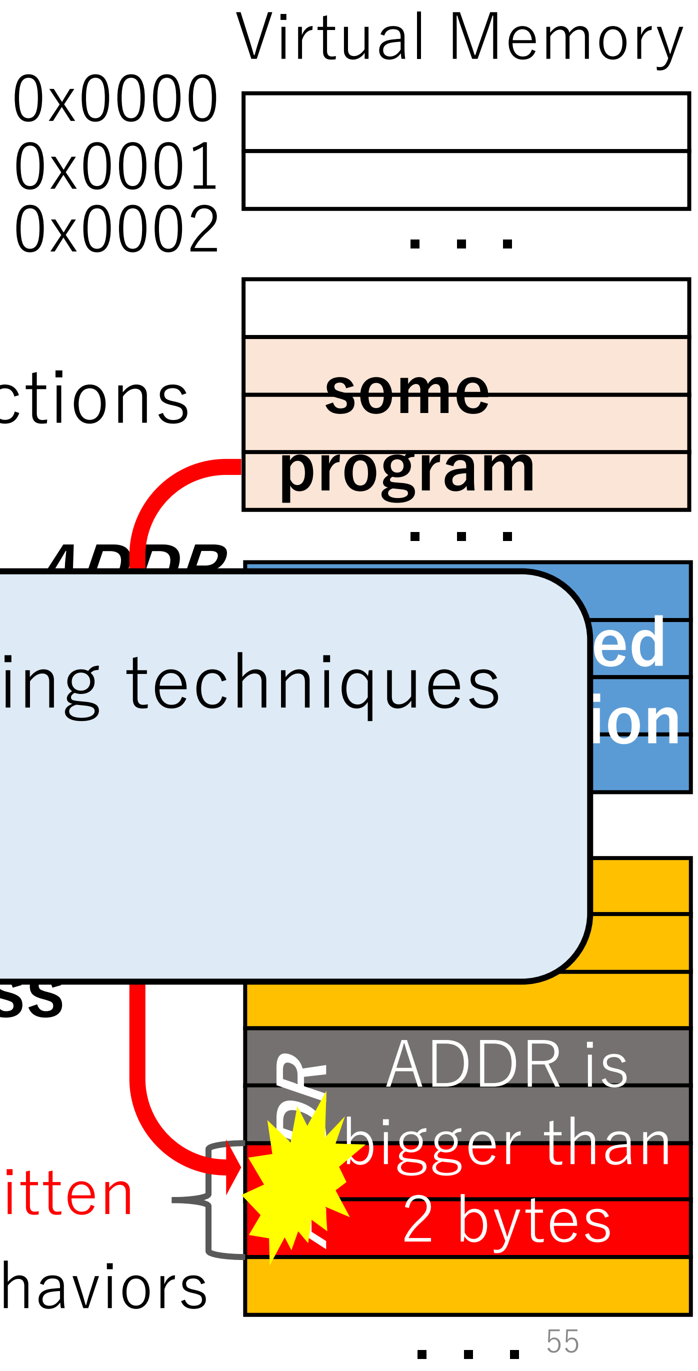
- On x86-64 CPUs, syscall and sysenter instructions trigger a system call
  - syscall: 0x0f 0x05, sysenter: 0x0f 0x34

Because of this issue, previous binary rewriting techniques
- could not ensure exhaustive hooking
- or, overwrite neighbour instructions

- **Specification for a jump destination address needs more than 2 bytes**

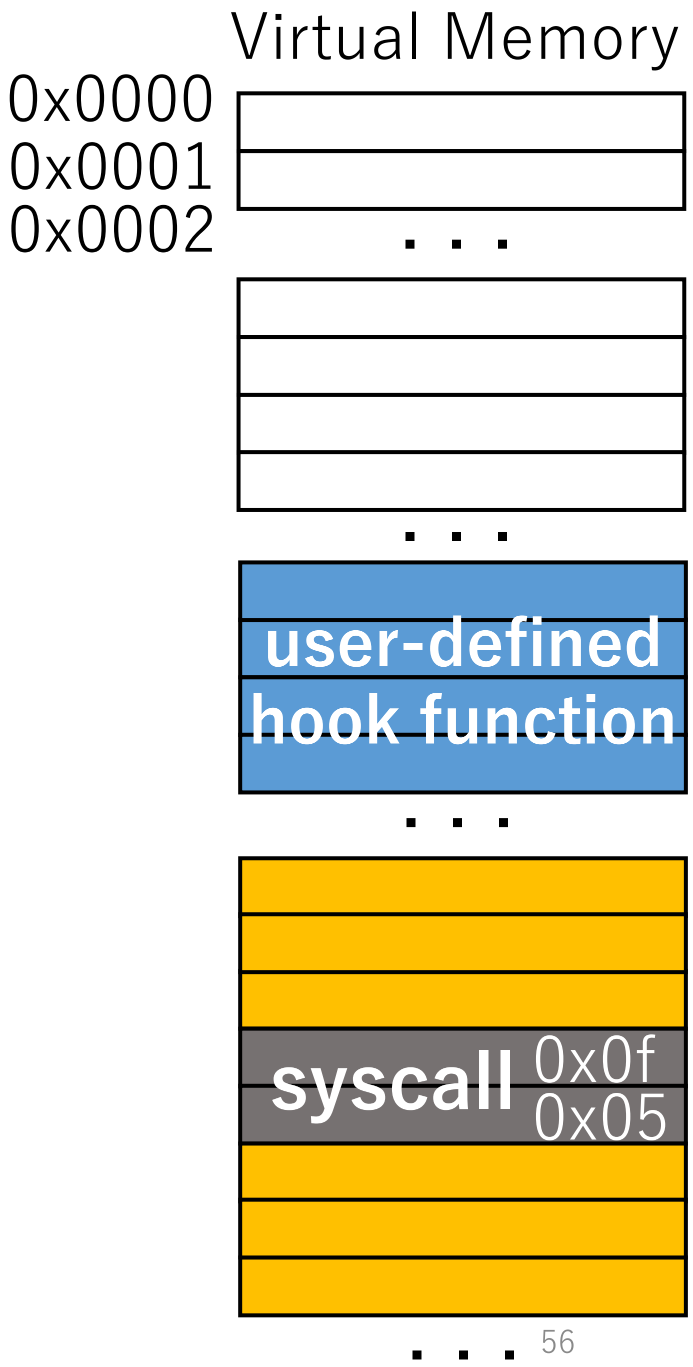
If we put ADDR, subsequent instructions are overwritten

jump to the overwritten part leads to unexpected behaviors



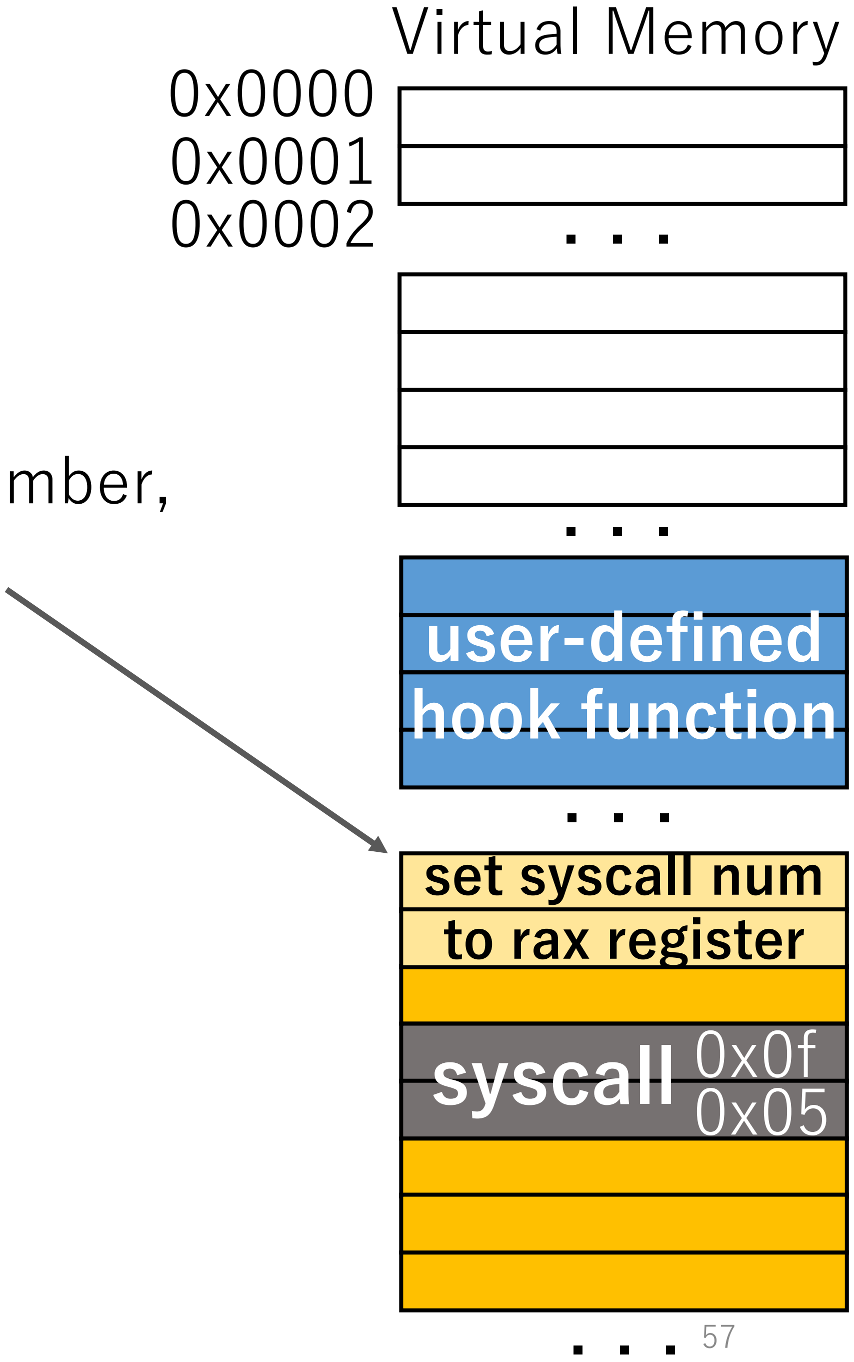
Calling Convention

- How to invoke a system call



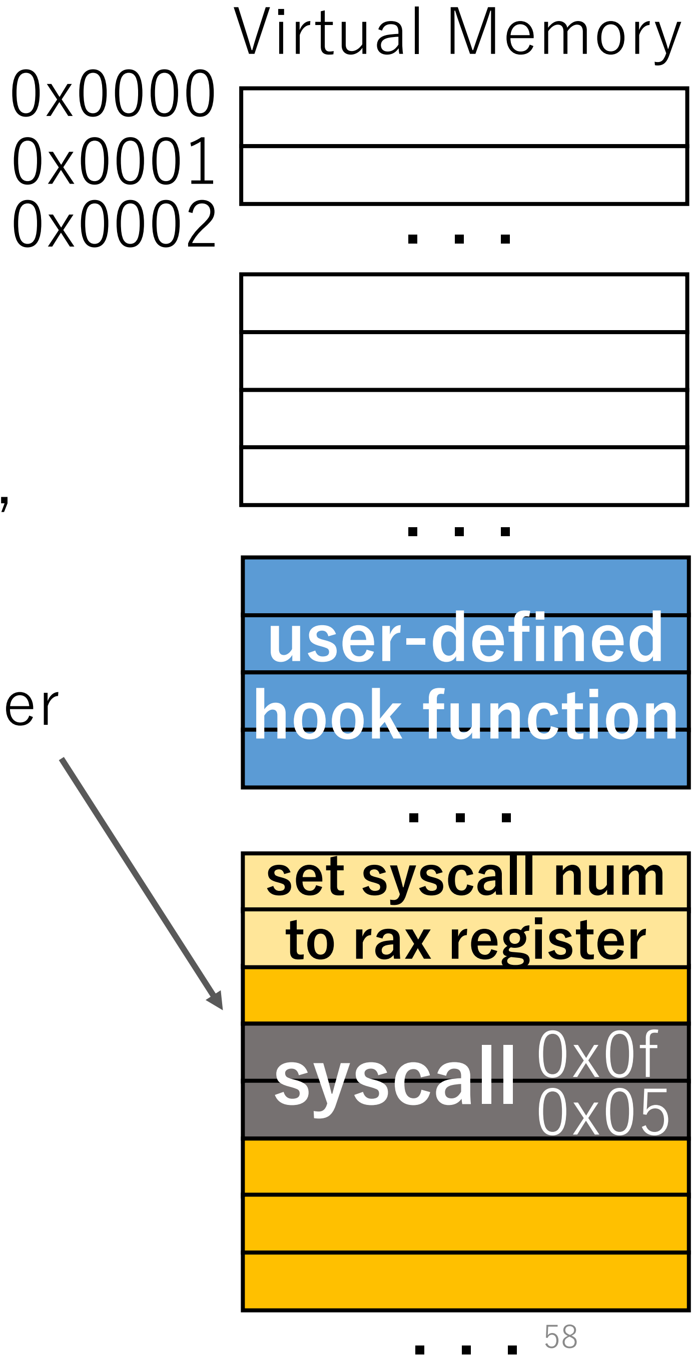
Calling Convention

- How to invoke a system call
  - A user-space program sets a system call number, predefined by the kenel, to the **rax register**
    - e.g., 0: read(), 1: write(), 2: open(), …



Calling Convention

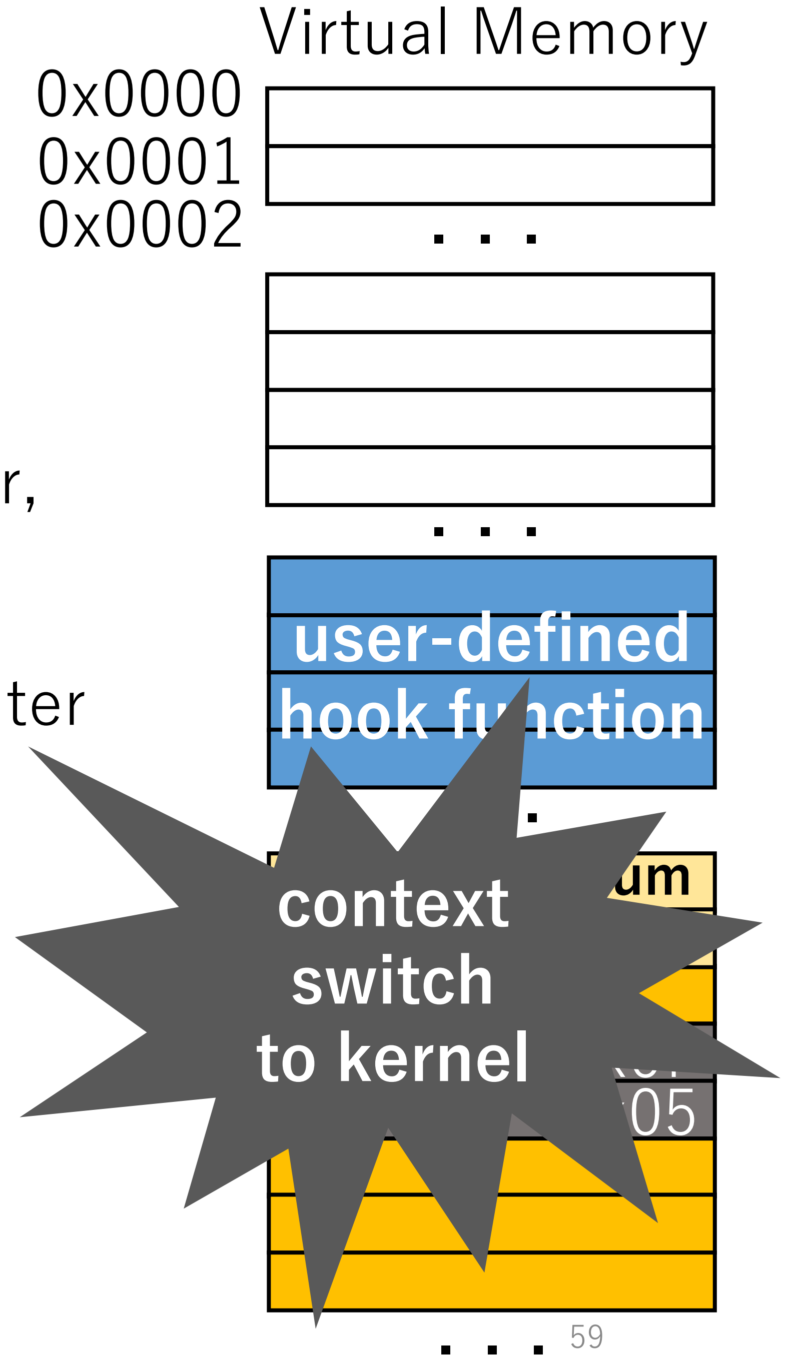
- How to invoke a system call
  - A user-space program sets a system call number, predefined by the kenel, to the **rax register**
    - e.g., 0: read(), 1: write(), 2: open(), …
  - The user-space program executes syscall/sysenter



Calling Convention

- How to invoke a system call
  - A user-space program sets a system call number, predefined by the kenel, to the **rax register**
    - e.g., 0: read(), 1: write(), 2: open(), …
  - The user-space program executes syscall/sysenter

---- the context is switched to the kernel ----

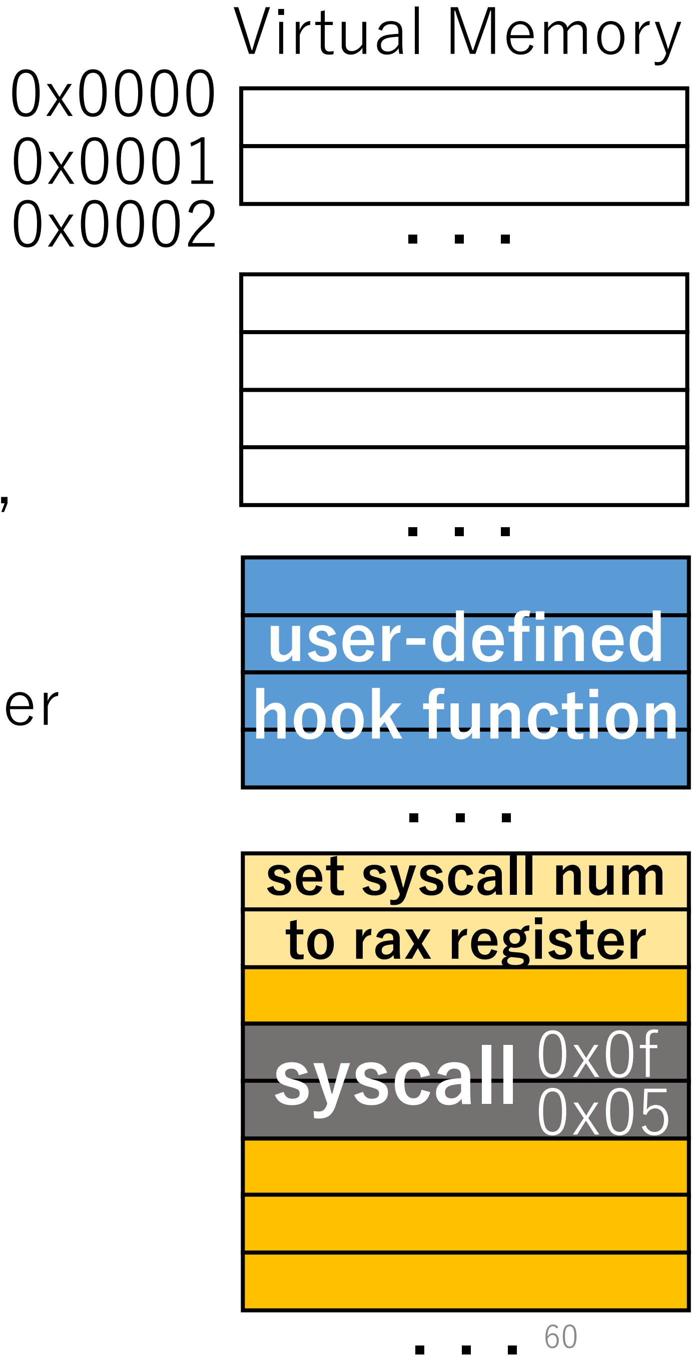


Calling Convention

- How to invoke a system call
  - A user-space program sets a system call number, predefined by the kenel, to the **rax register**
    - e.g., 0: read(), 1: write(), 2: open(), …
  - The user-space program executes syscall/sysenter

---- the context is switched to the kernel ----

  - Kernel executes a system call specified through the system call number set to the **rax register**

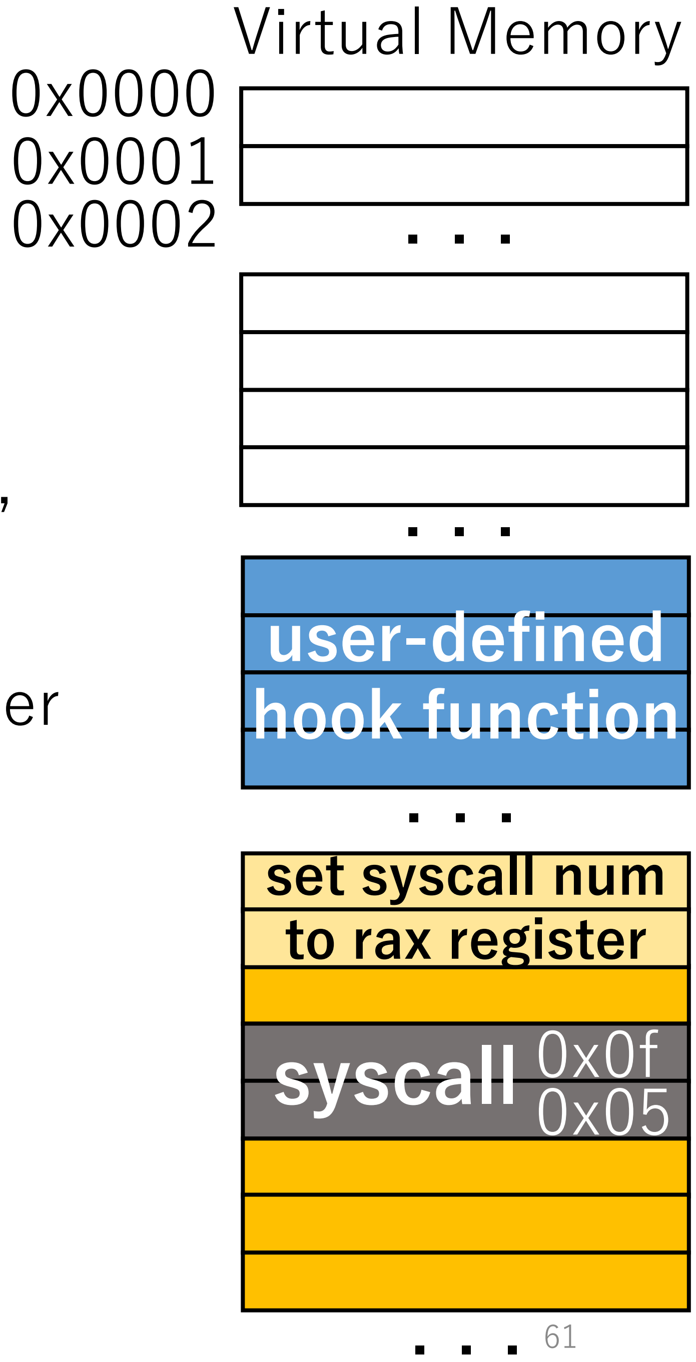


Calling Convention

- How to invoke a system call
  - A user-space program sets a system call number, predefined by the kenel, to the **rax register**
    - e.g., 0: read(), 1: write(), 2: open(), …
  - The user-space program executes syscall/sysenter

---- the context is switched to the kernel ----

  - Kernel executes a system call specified through the system call number set to the **rax register**
    - if the rax register has 0, the kernel executes read()
    - if the rax register has 1, the kernel executes write()
    - if the rax register has 2, the kernel executes open()



Calling Convention

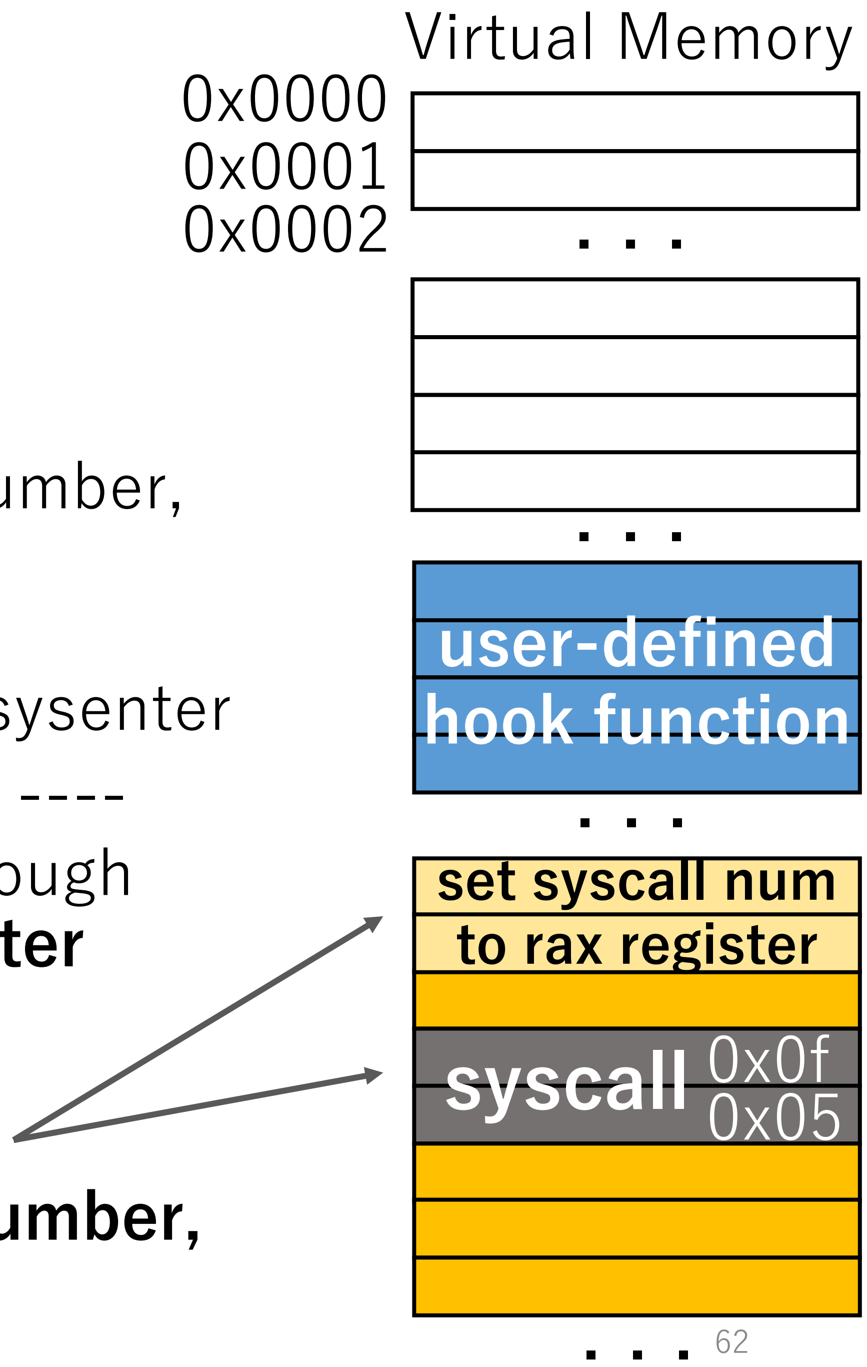
- How to invoke a system call
  - A user-space program sets a system call number, predefined by the kenel, to the **rax register**
    - e.g., 0: read(), 1: write(), 2: open(), …
  - The user-space program executes syscall/sysenter

---- the context is switched to the kernel ----

  - Kernel executes a system call specified through the system call number set to the **rax register**

<u>Point: Calling Convention</u>

When syscall/sysenter is executed,
the rax register always has a system call number,



Calling Convention

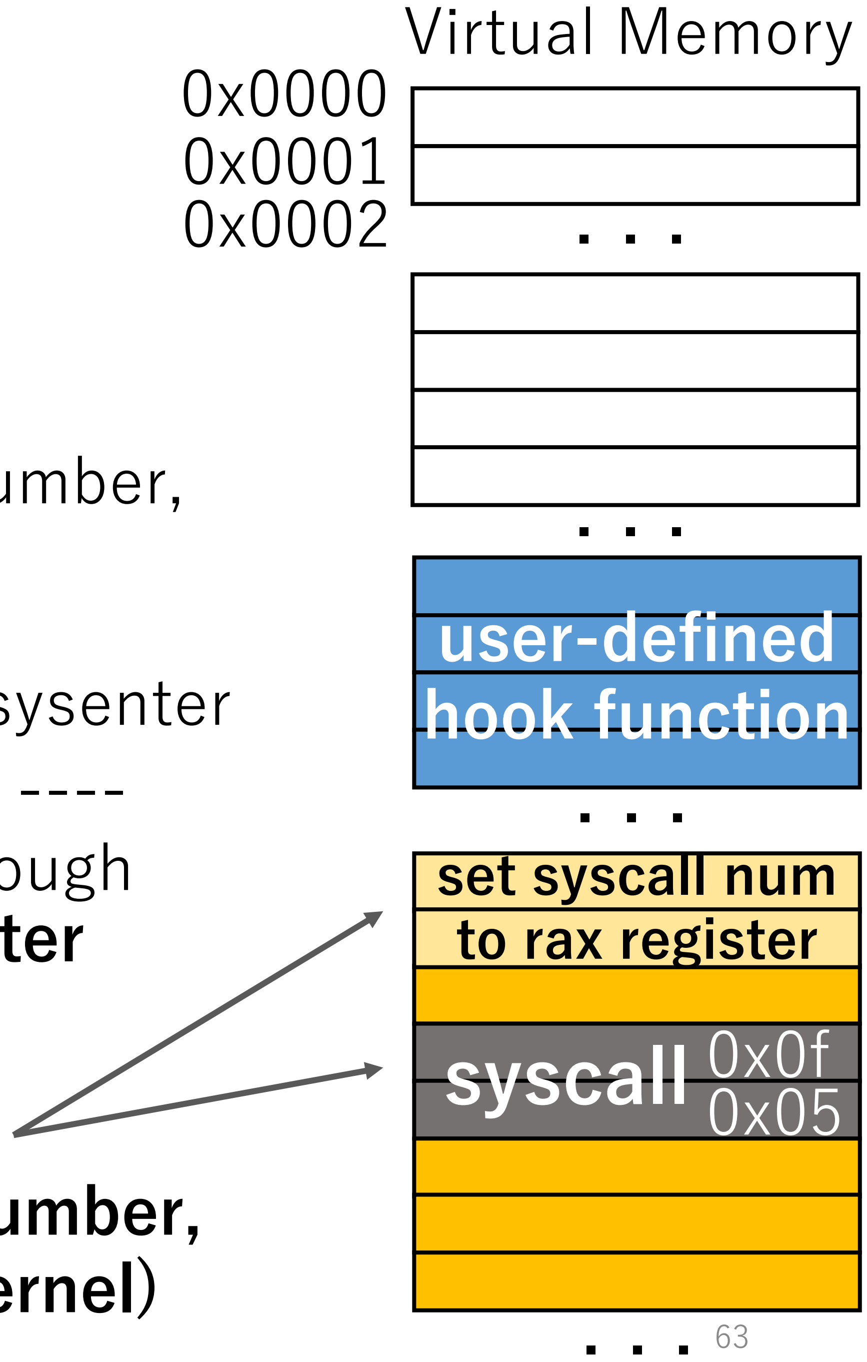
- How to invoke a system call
  - A user-space program sets a system call number, predefined by the kenel, to the **rax register**
    - e.g., 0: read(), 1: write(), 2: open(), ...
  - The user-space program executes syscall/sysenter

---- the context is switched to the kernel ----

  - Kernel executes a system call specified through the system call number set to the **rax register**

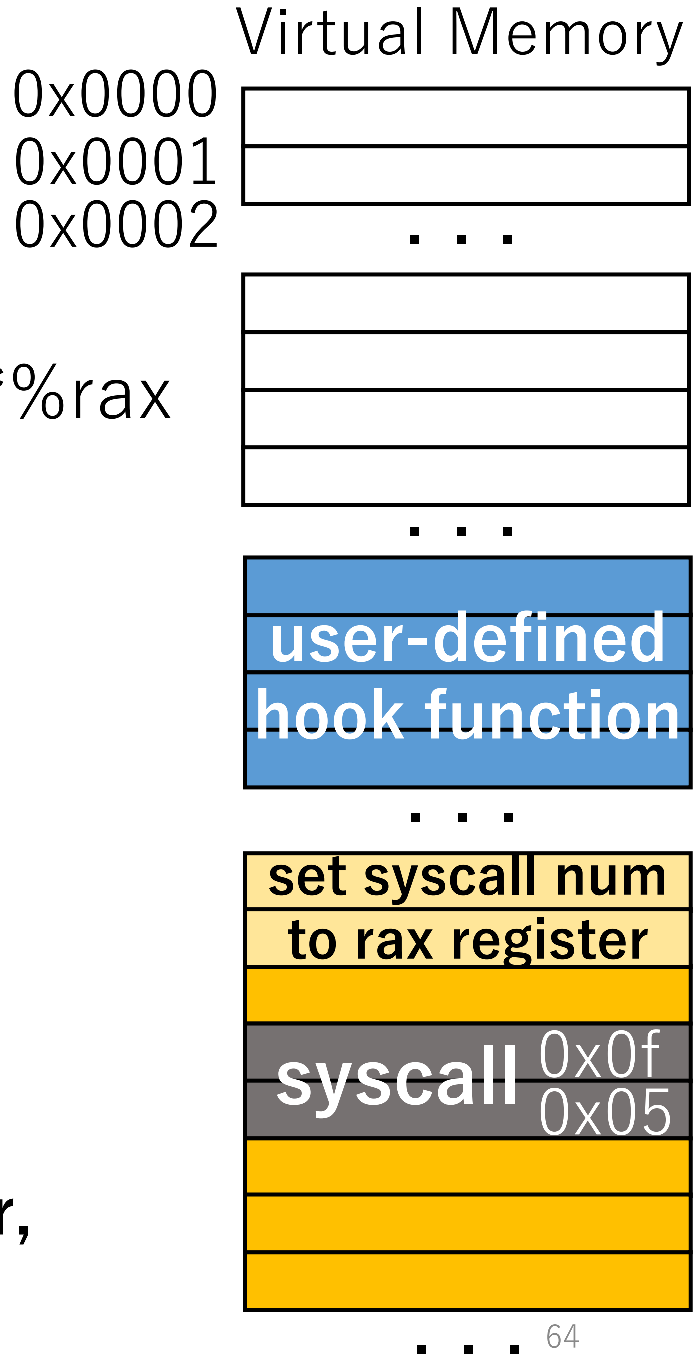
Point: Calling Convention

When syscall/sysenter is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)



zpoline

- zpoline replaces syscall/sysenter with callq *%rax

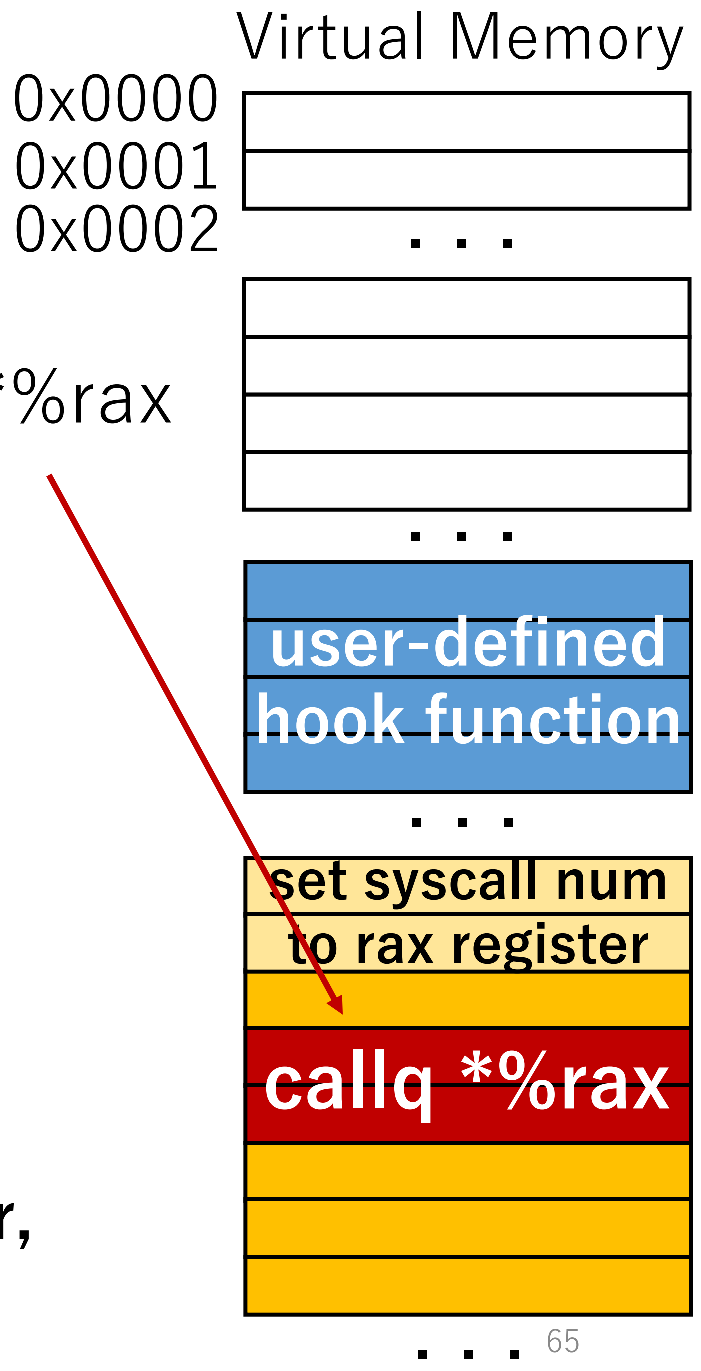


Point: Calling Convention

**When syscall/sysenter is executed,
the rax register always has a system call number,
which is 0 ~ around 500 (defined in the kernel)**

zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)

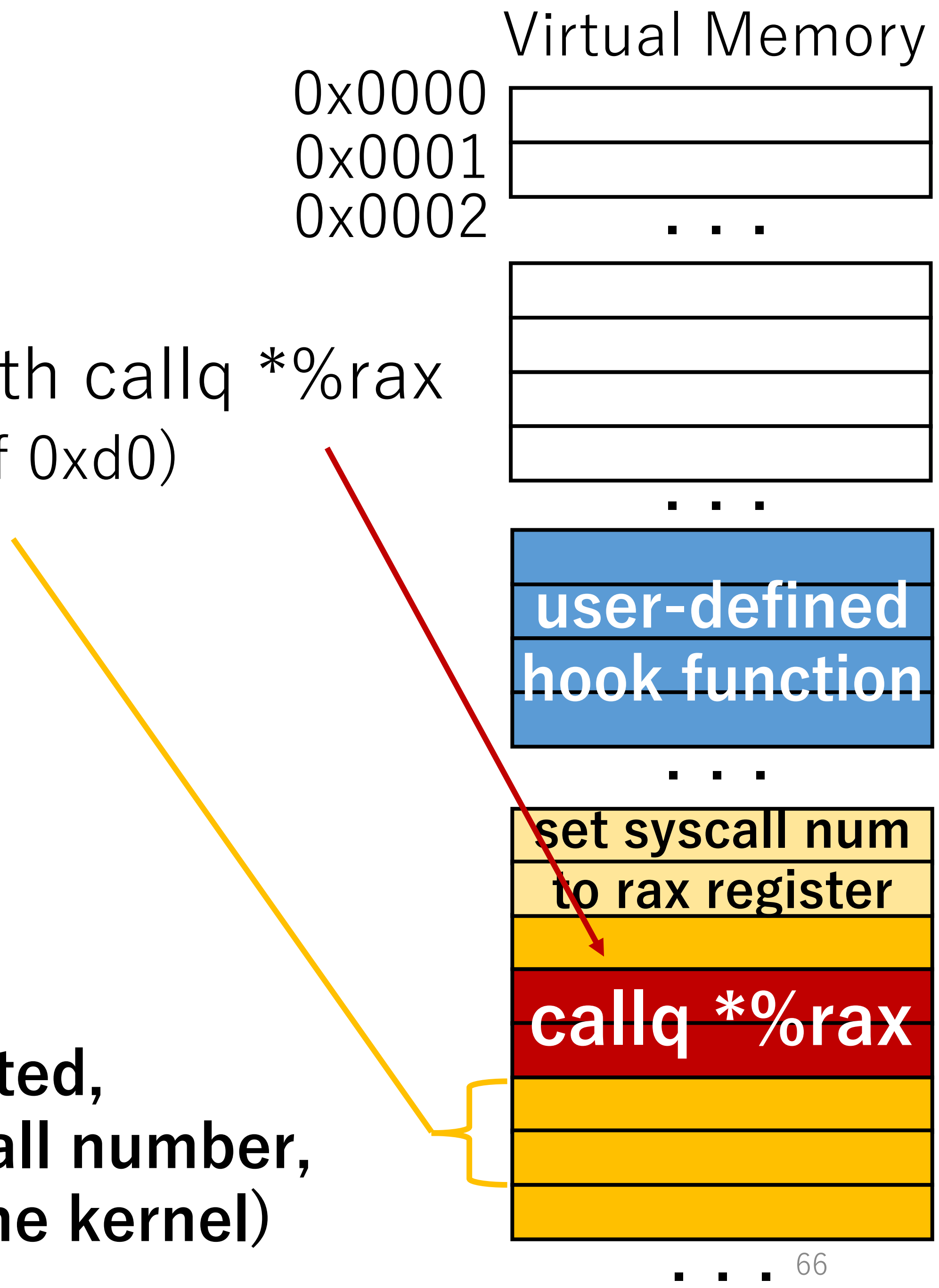


Point: Calling Convention

When syscall/sysenter is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)

zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten

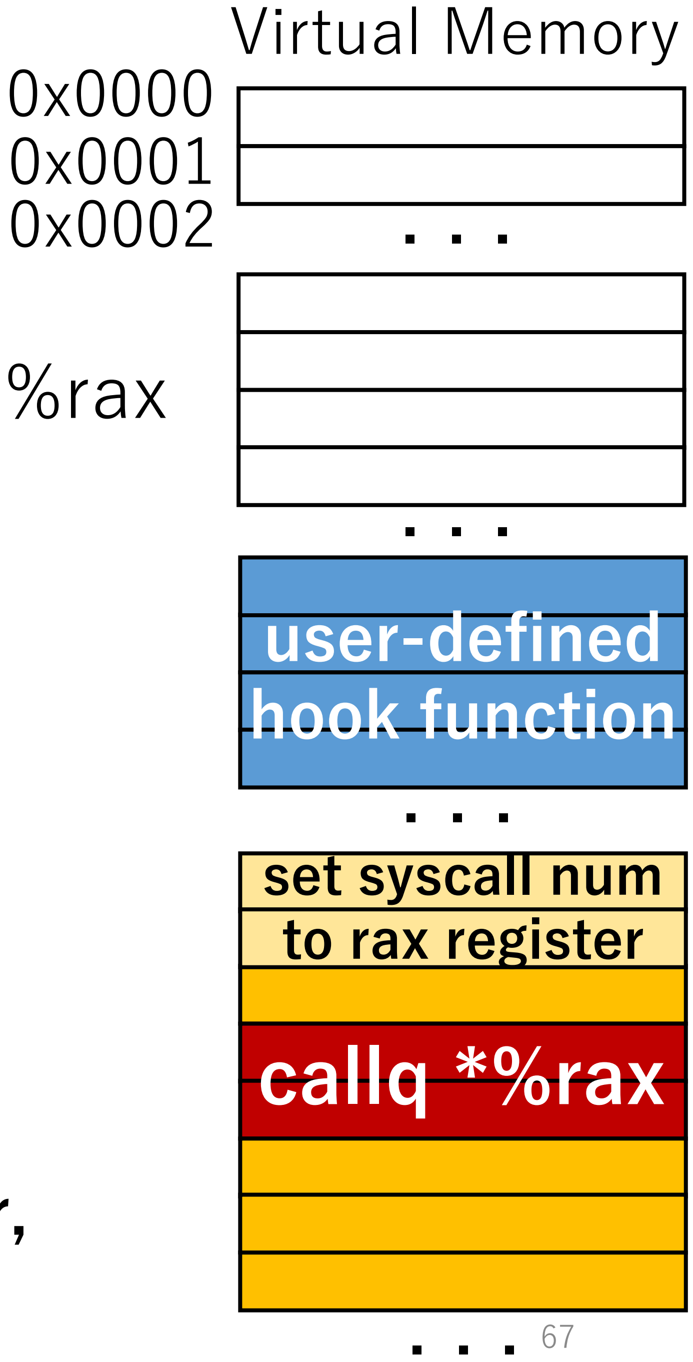


Point: Calling Convention

When syscall/sysenter is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)

zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register

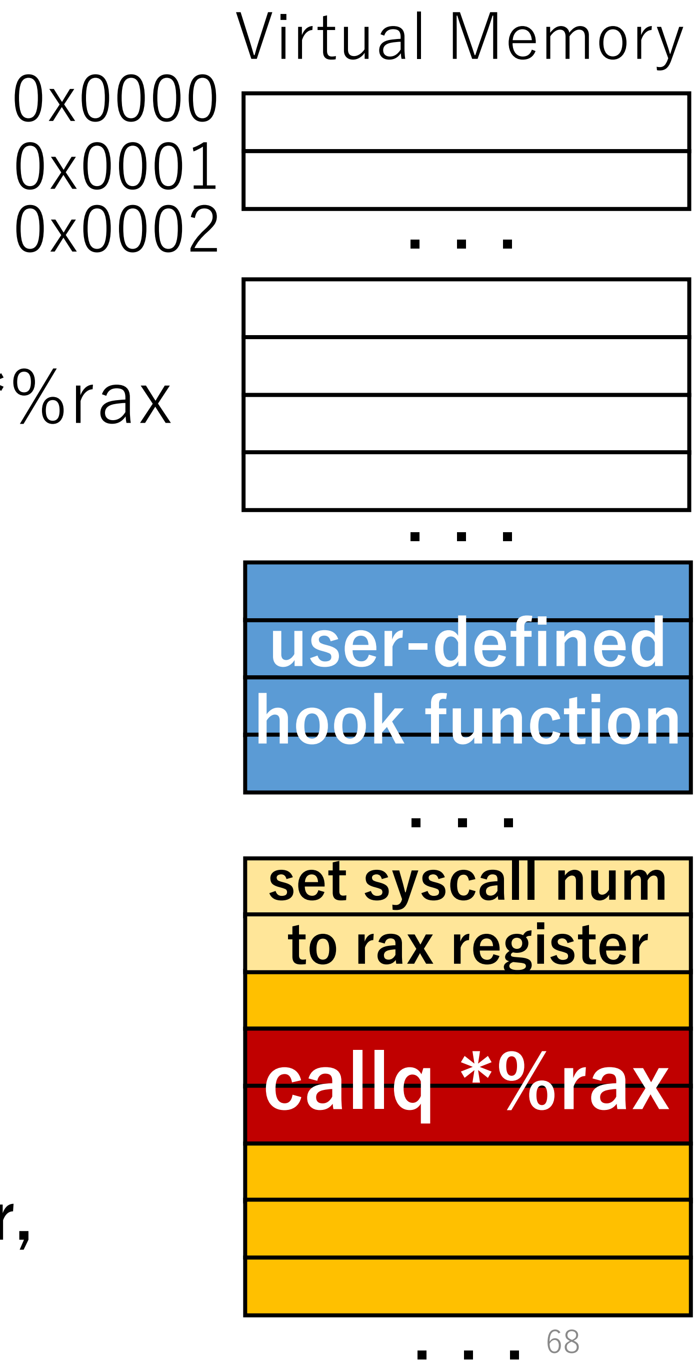


<u>Point: Calling Convention</u>

When syscall/sysenter is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)

zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register

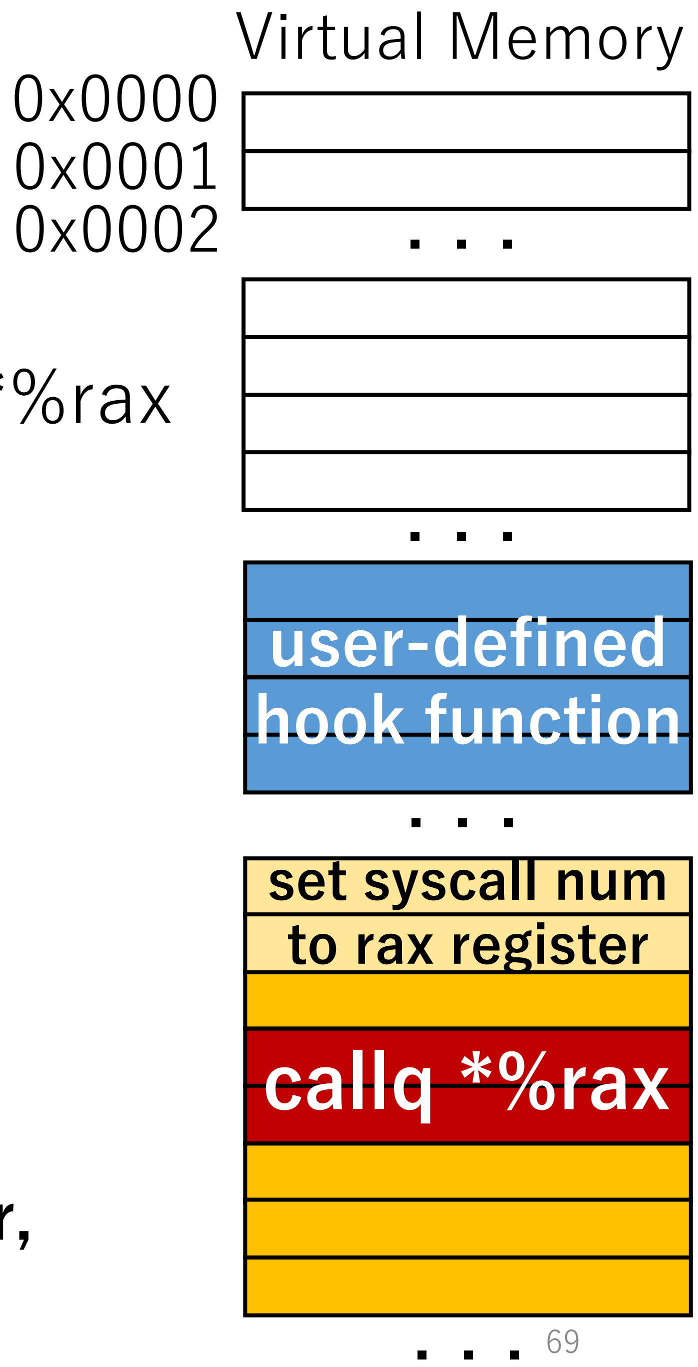


Point: Calling Convention

When syscall/sysenter is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)

zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register



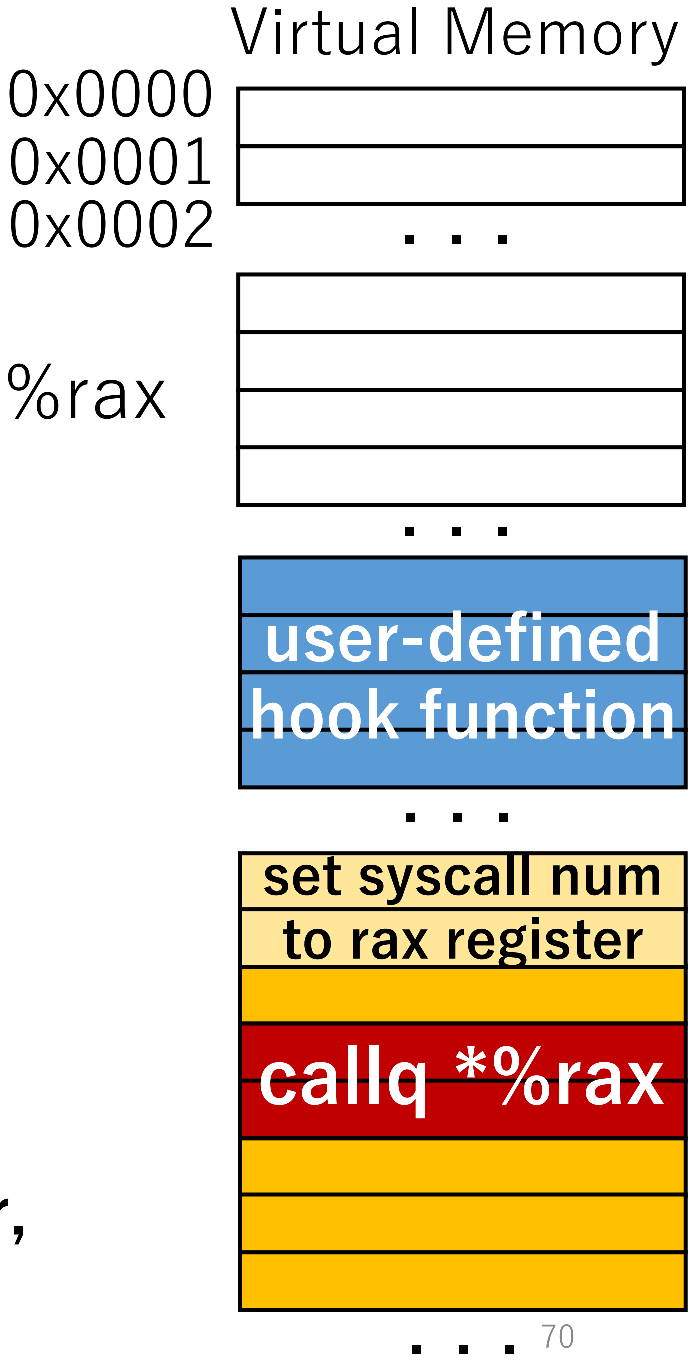
After the binary rewriting

Point: Calling Convention

When syscall/sysenter is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)

zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register



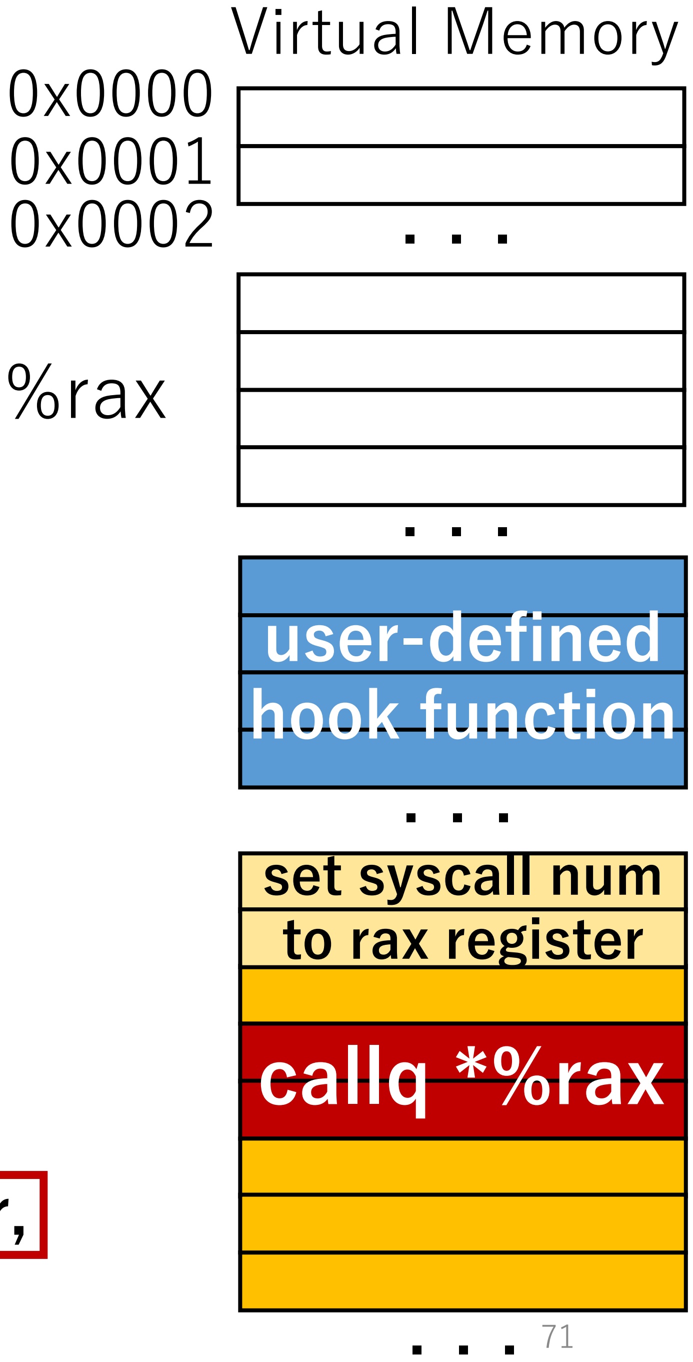
After the binary rewriting

Point: Calling Convention

When ~~syscall/sysenter~~ callq *%rax is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)

zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register



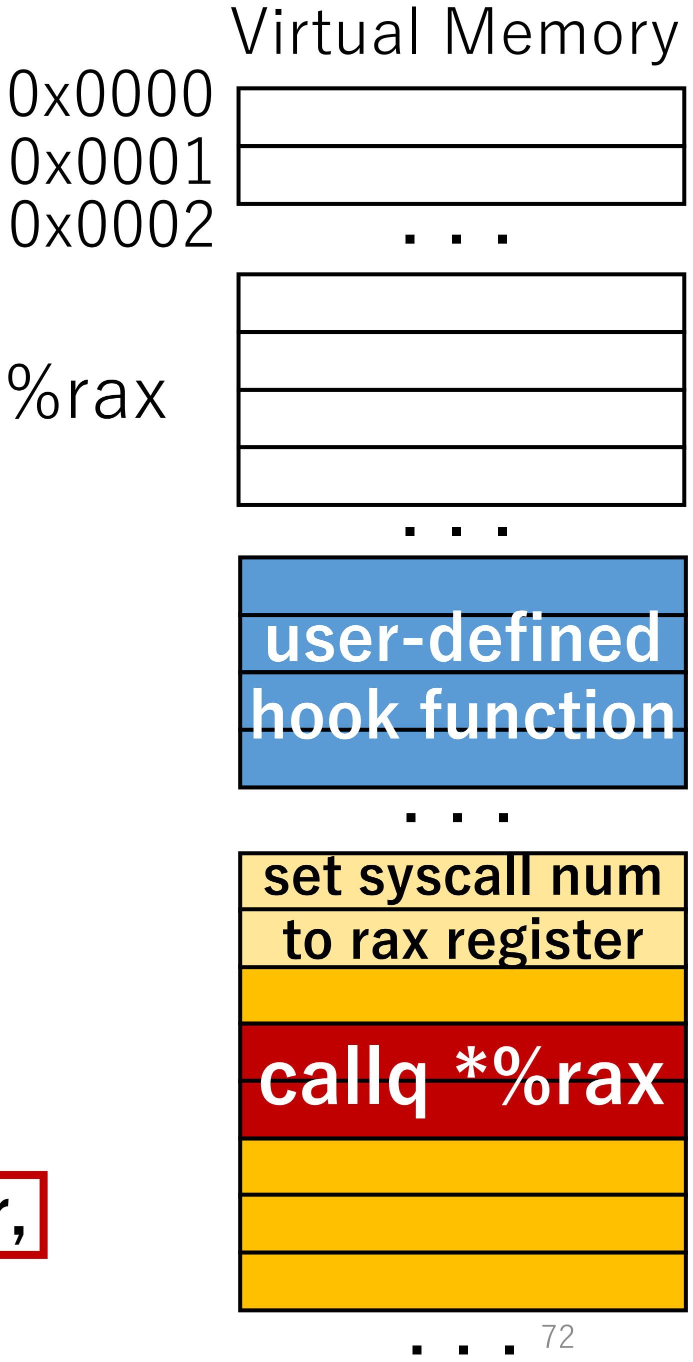
After the binary rewriting

Point: Calling Convention

When ~~syscall/sysenter~~ callq *%rax is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)

zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register



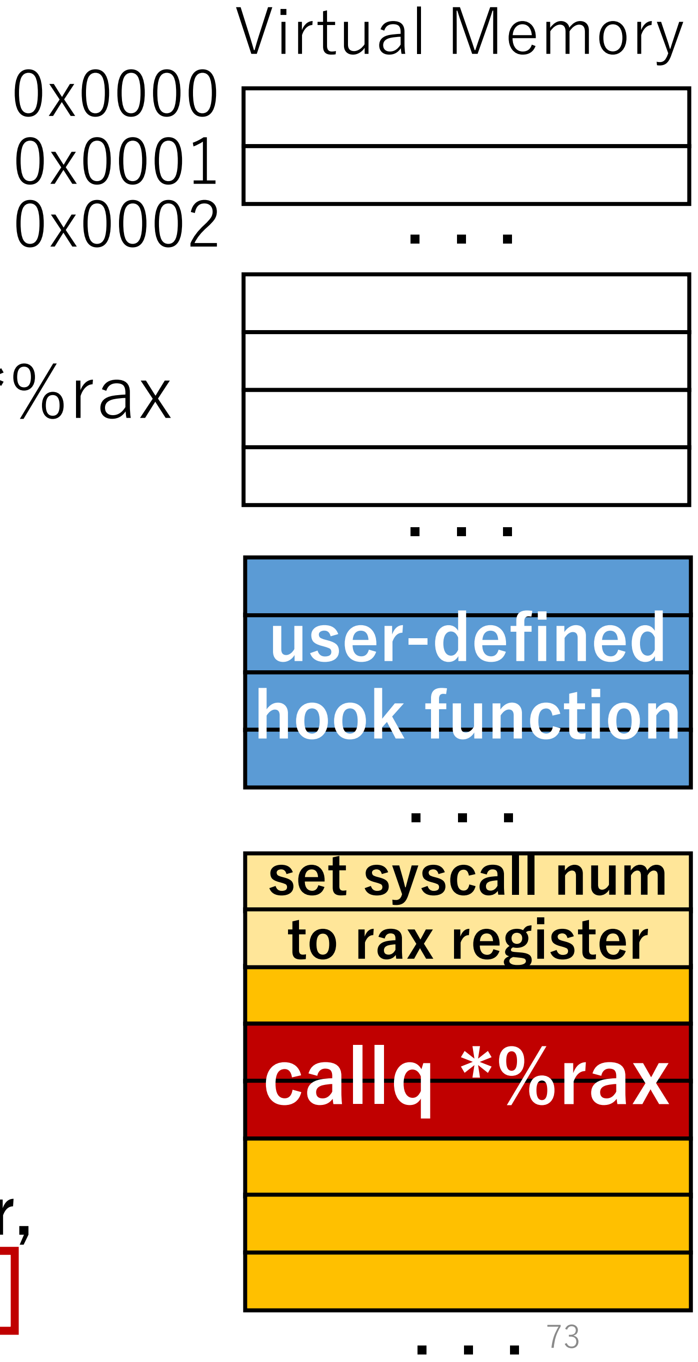
After the binary rewriting

Point: Calling Convention

When ~~syscall/sysenter~~ callq *%rax is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)

zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register



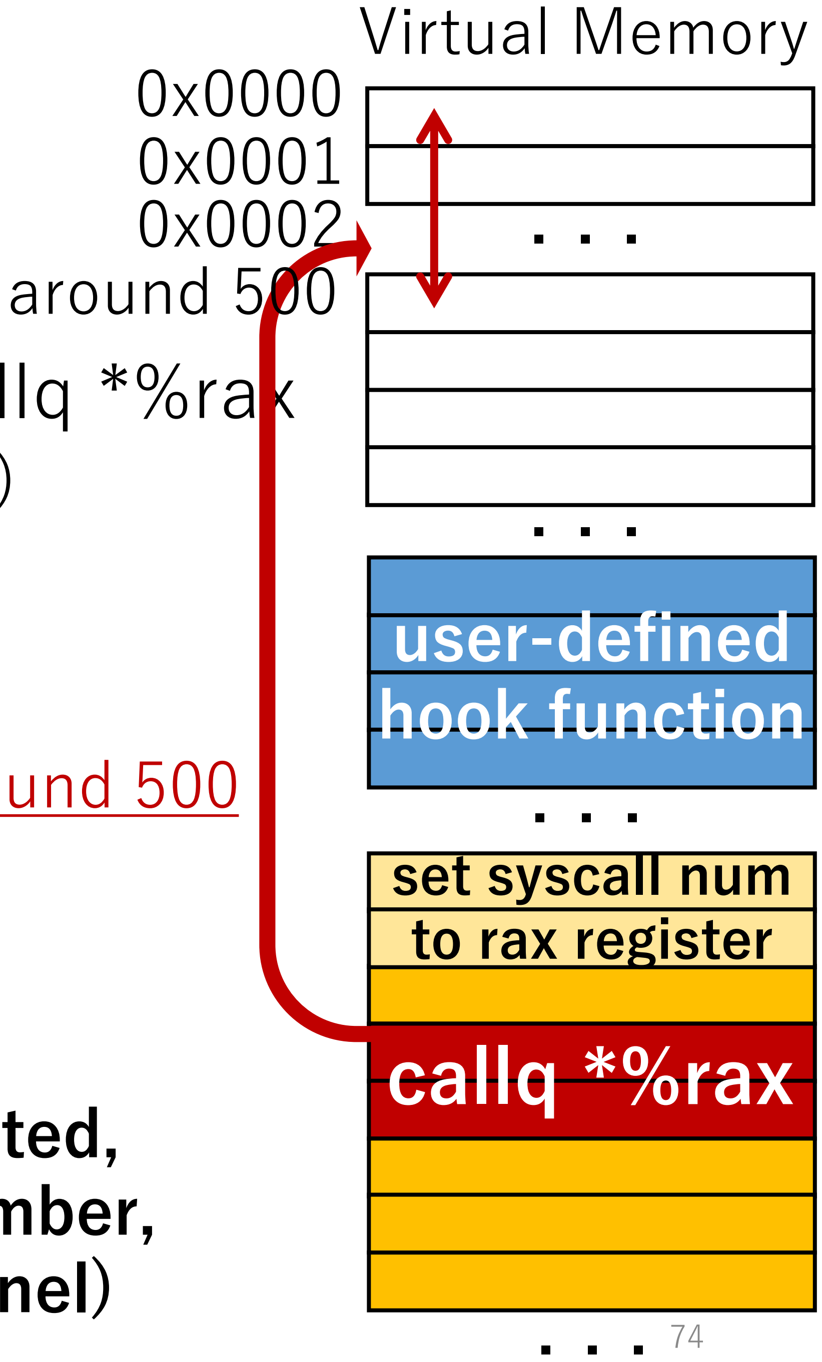
After the binary rewriting

<u>Point: Calling Convention</u>

When ~~syscall/sysenter~~ callq *%rax is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)

zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register
  - replaced callq *%rax jumps to address 0~around 500



After the binary rewriting

Point: Calling Convention

When ~~syscall/sysenter~~ callq *%rax is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)

zpoline

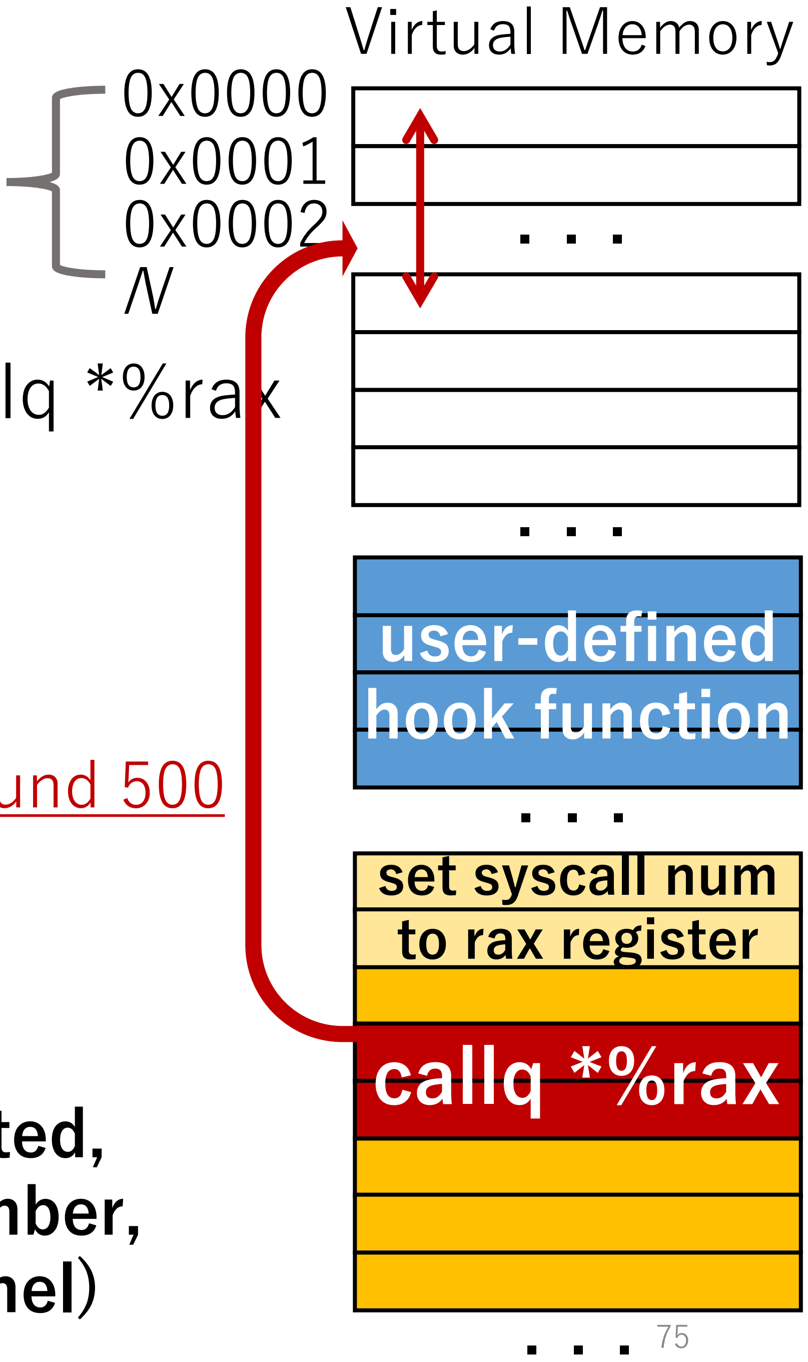
address range, potentially replaced "callq *%rax" jumps to
(*N* is the max syscall number)

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register
  - replaced callq *%rax jumps to address 0~around 500

After the binary rewriting

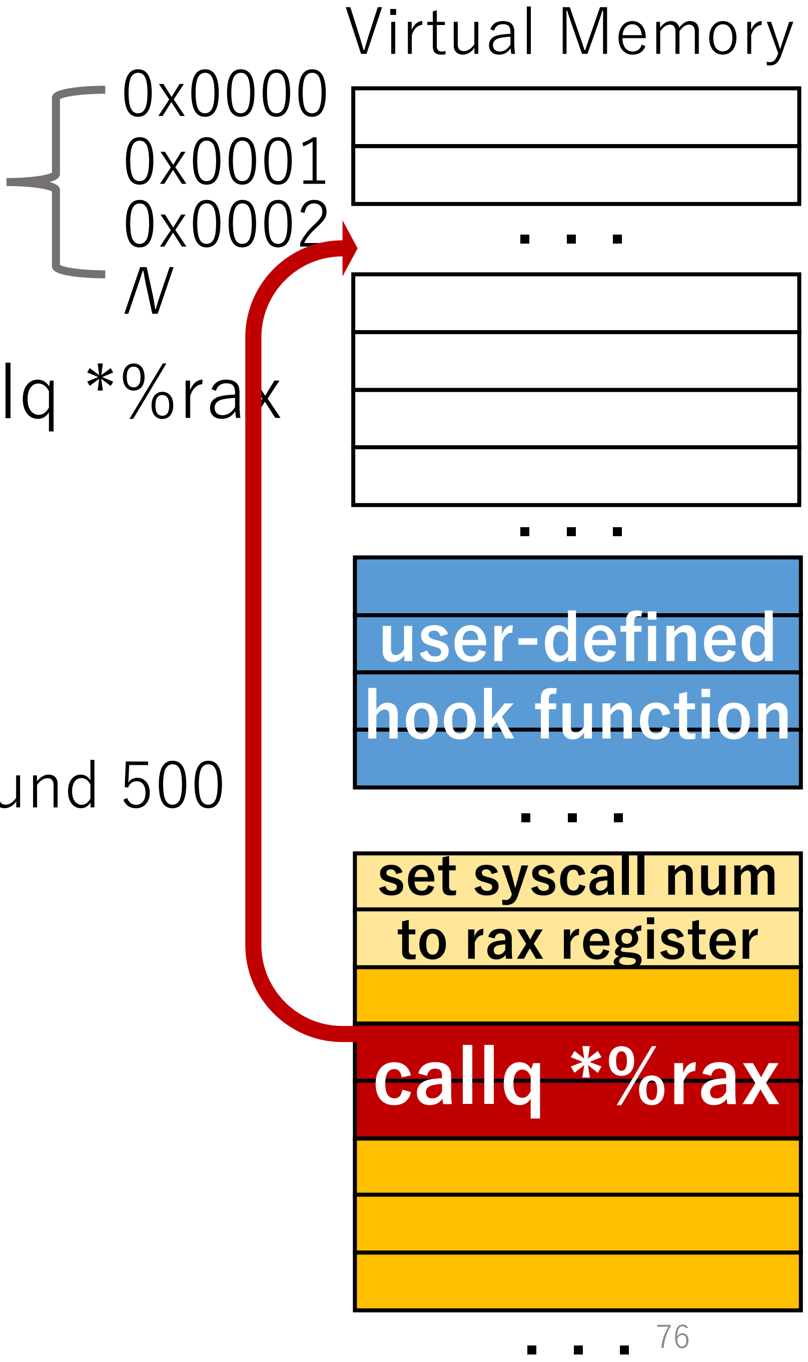
Point: Calling Convention

When ~~syscall/sysenter~~ callq *%rax is executed, the rax register always has a system call number, which is 0 ~ around 500 (defined in the kernel)



zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register
  - replaced callq *%rax jumps to address 0~around 500



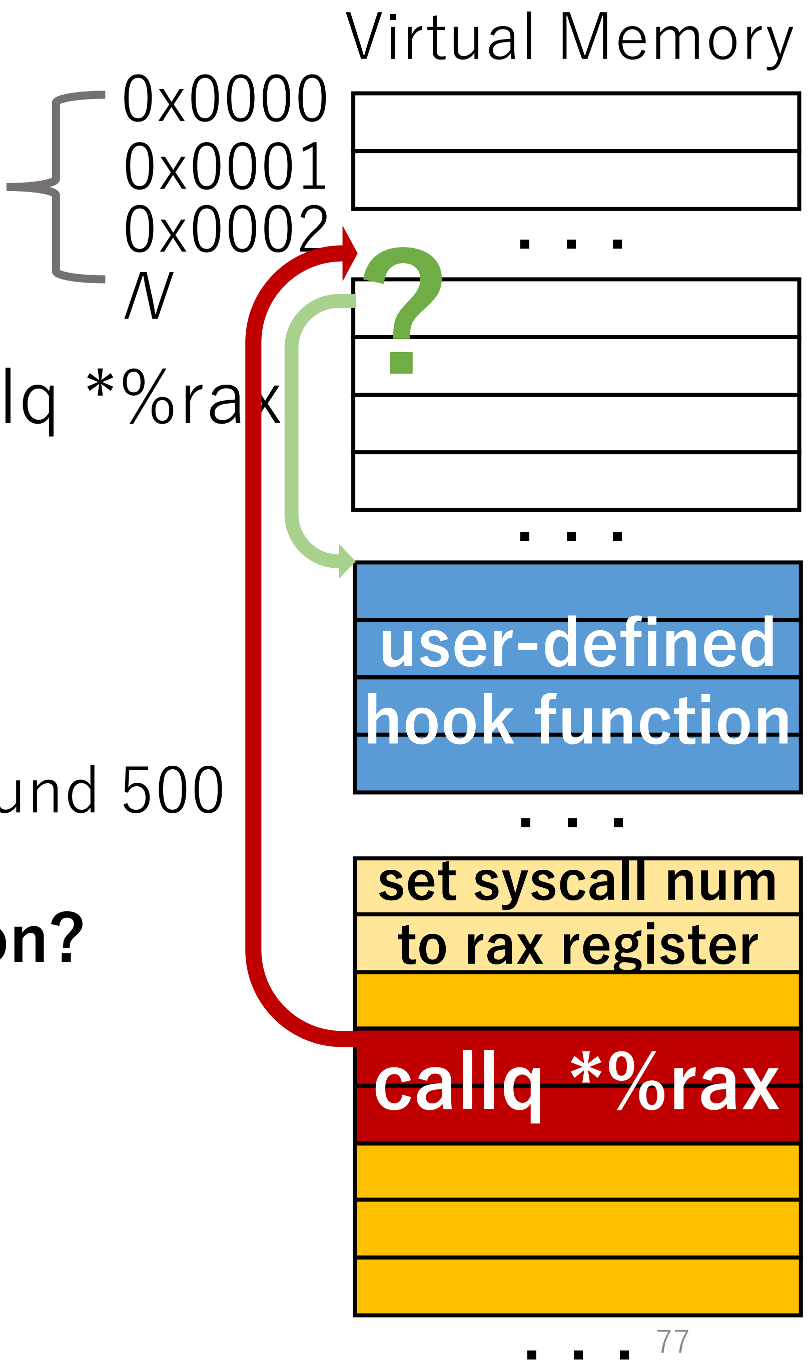
address range, potentially replaced "callq *%rax" jumps to
(*N* is the max syscall number)

zpoline

address range, potentially replaced “callq *%rax” jumps to
(N is the max syscall number)

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register
  - replaced callq *%rax jumps to address 0~around 500

How to redirect to the user-defined hook function?

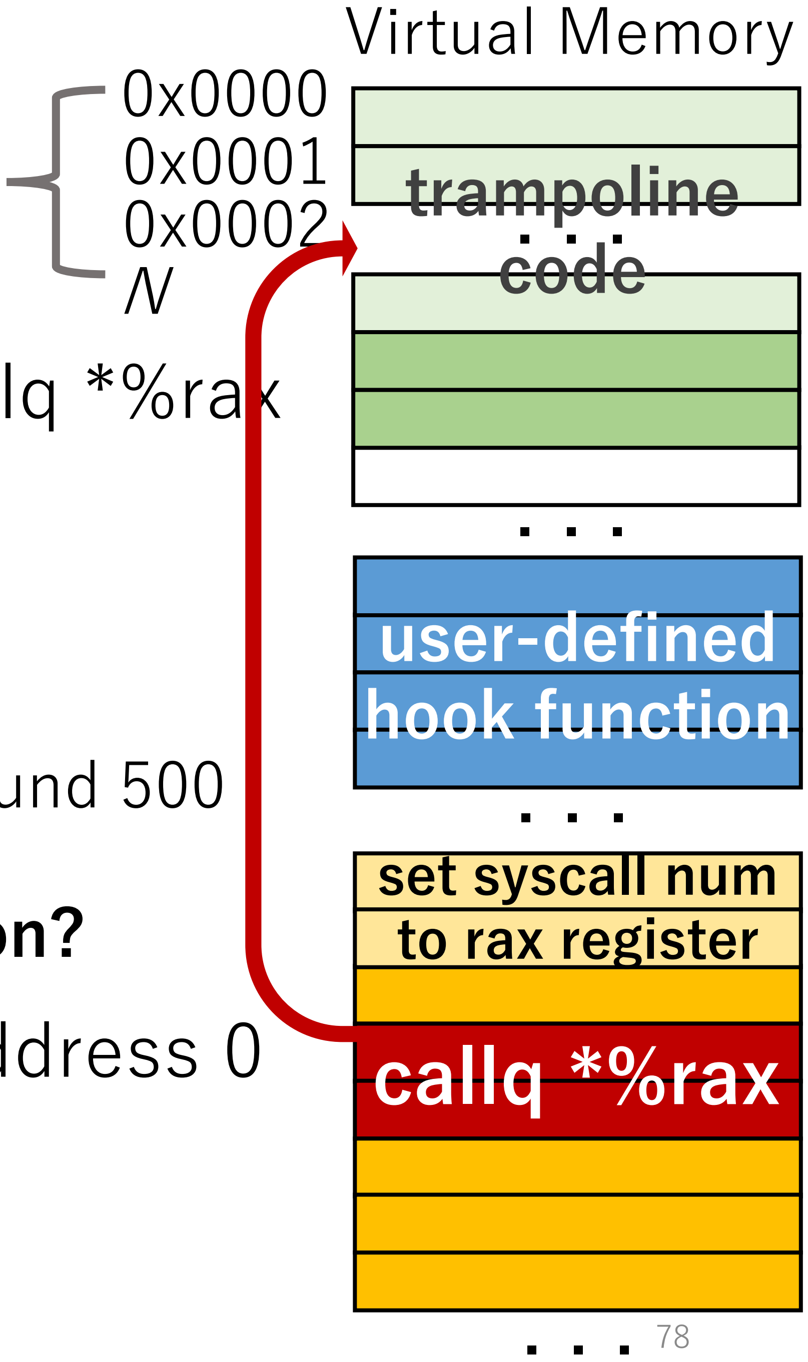


zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register
  - replaced callq *%rax jumps to address 0~around 500

How to redirect to the user-defined hook function?

- zpoline instantiates trampoline code at address 0



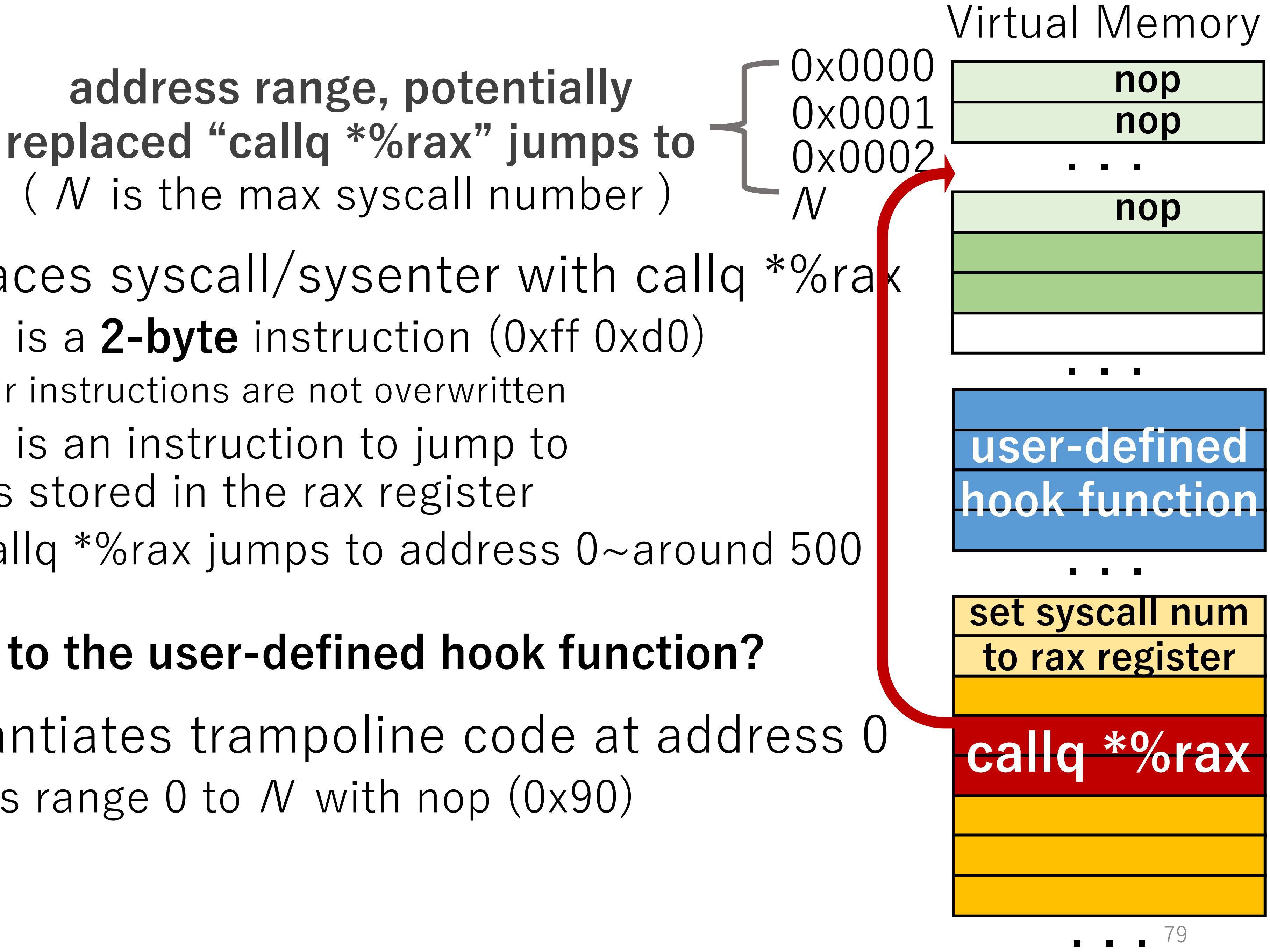
Virtual Memory
address range, potentially replaced "callq *%rax" jumps to
(N is the max syscall number)
0x0000
0x0001
0x0002
N
trampoline code
user-defined hook function
set syscall num to rax register
callq *%rax

zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register
  - replaced callq *%rax jumps to address 0~around 500

How to redirect to the user-defined hook function?

- zpoline instantiates trampoline code at address 0
  - fills address range 0 to N with nop (0x90)



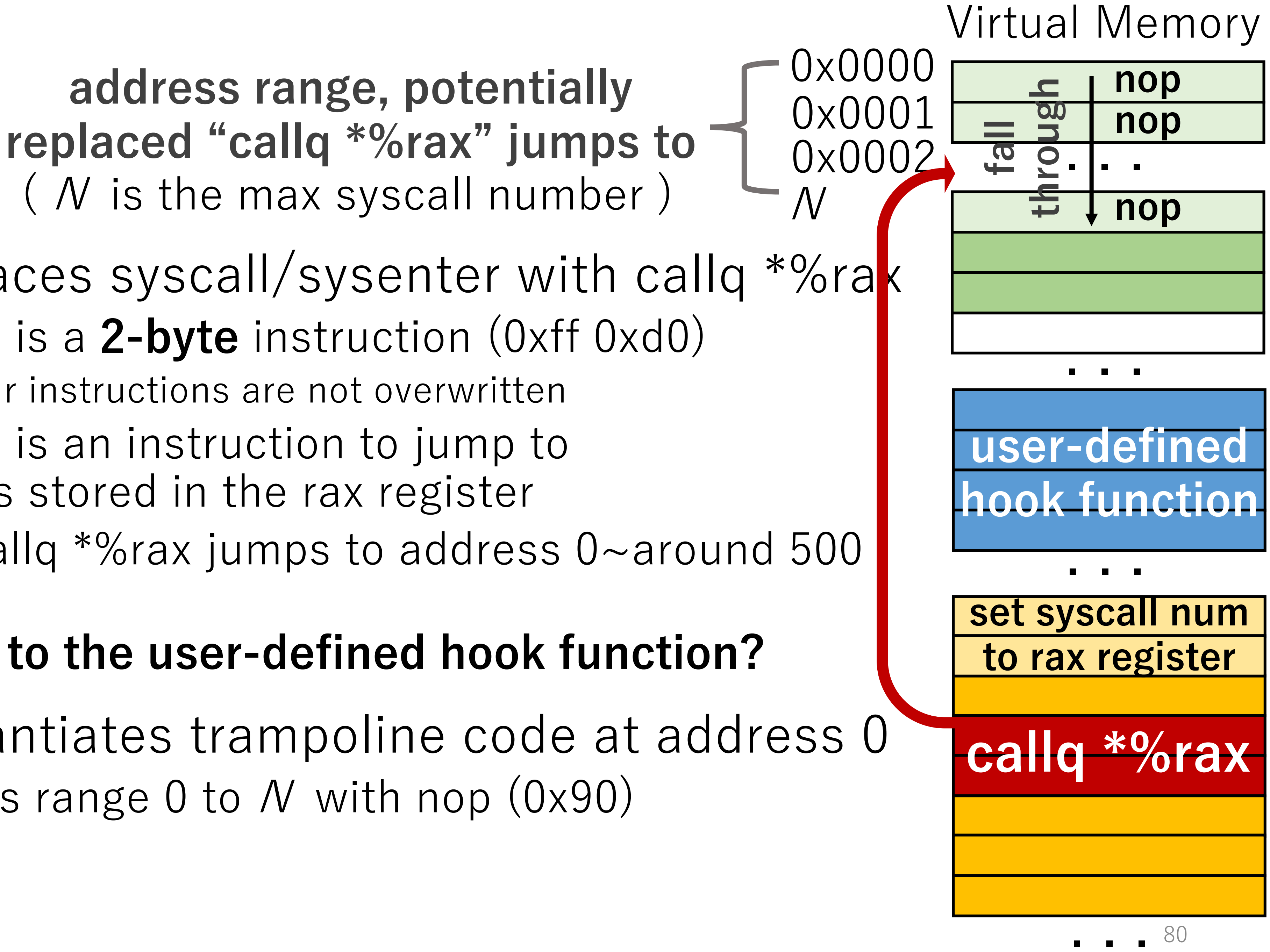
Virtual Memory
address range, potentially replaced "callq *%rax" jumps to
(N is the max syscall number)
0x0000
0x0001
0x0002
N
nop
nop
. . .
nop
. . .
user-defined hook function
. . .
set syscall num to rax register
callq *%rax
. . .

zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register
  - replaced callq *%rax jumps to address 0~around 500

How to redirect to the user-defined hook function?

- zpoline instantiates trampoline code at address 0
  - fills address range 0 to N with nop (0x90)



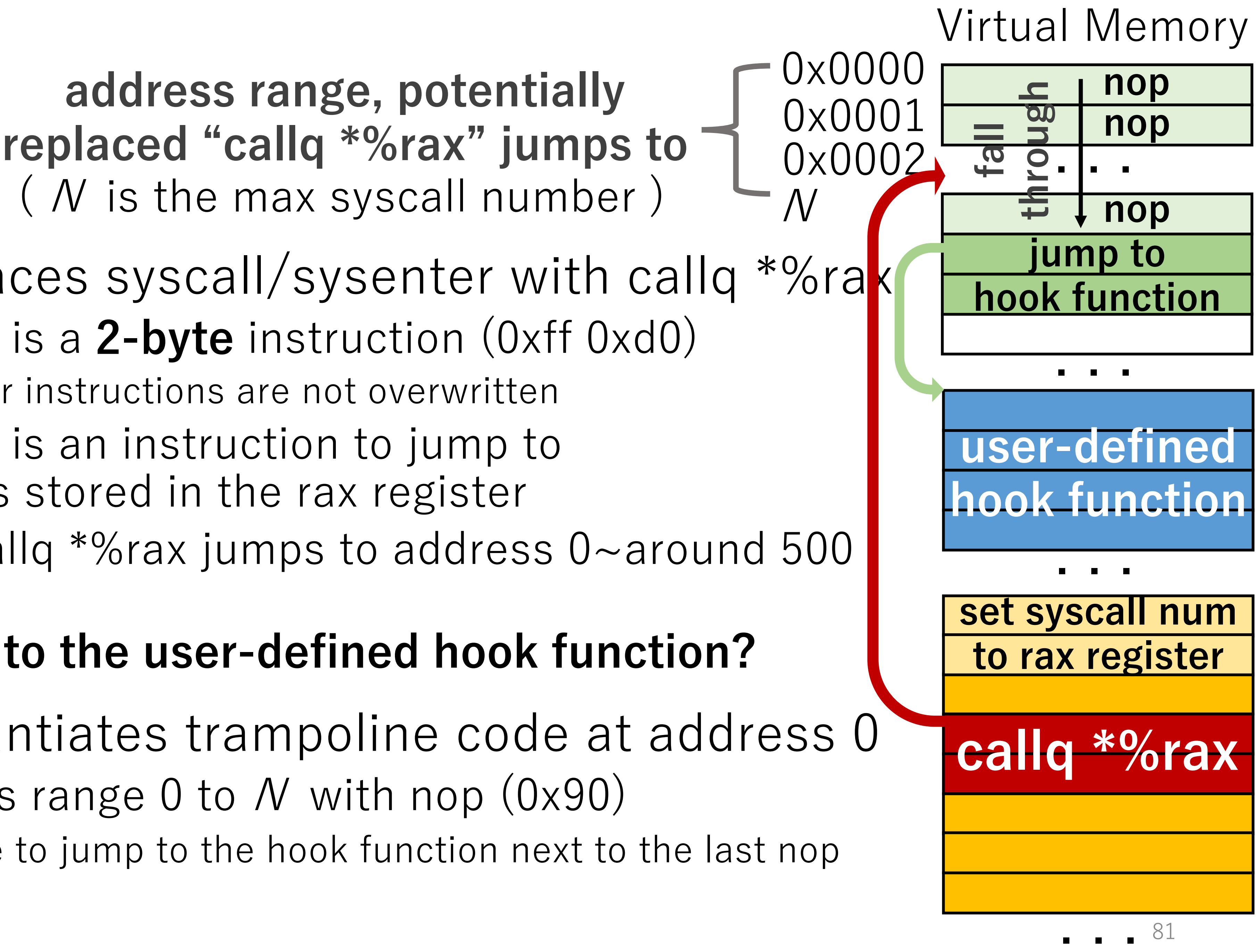
address range, potentially
replaced "callq *%rax" jumps to
(N is the max syscall number)
Virtual Memory
0x0000
0x0001
0x0002
N
nop
nop
. . .
nop
fall through
. . .
user-defined hook function
. . .
set syscall num to rax register
callq *%rax
. . .

zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register
  - replaced callq *%rax jumps to address 0~around 500

How to redirect to the user-defined hook function?

- zpoline instantiates trampoline code at address 0
  - fills address range 0 to N with nop (0x90)
    - puts code to jump to the hook function next to the last nop



address range, potentially replaced "callq *%rax" jumps to
(N is the max syscall number)
Virtual Memory
0x0000
0x0001
0x0002
N
nop
nop
. . .
nop
fall through
jump to hook function
. . .
user-defined hook function
. . .
set syscall num to rax register
callq *%rax
. . .

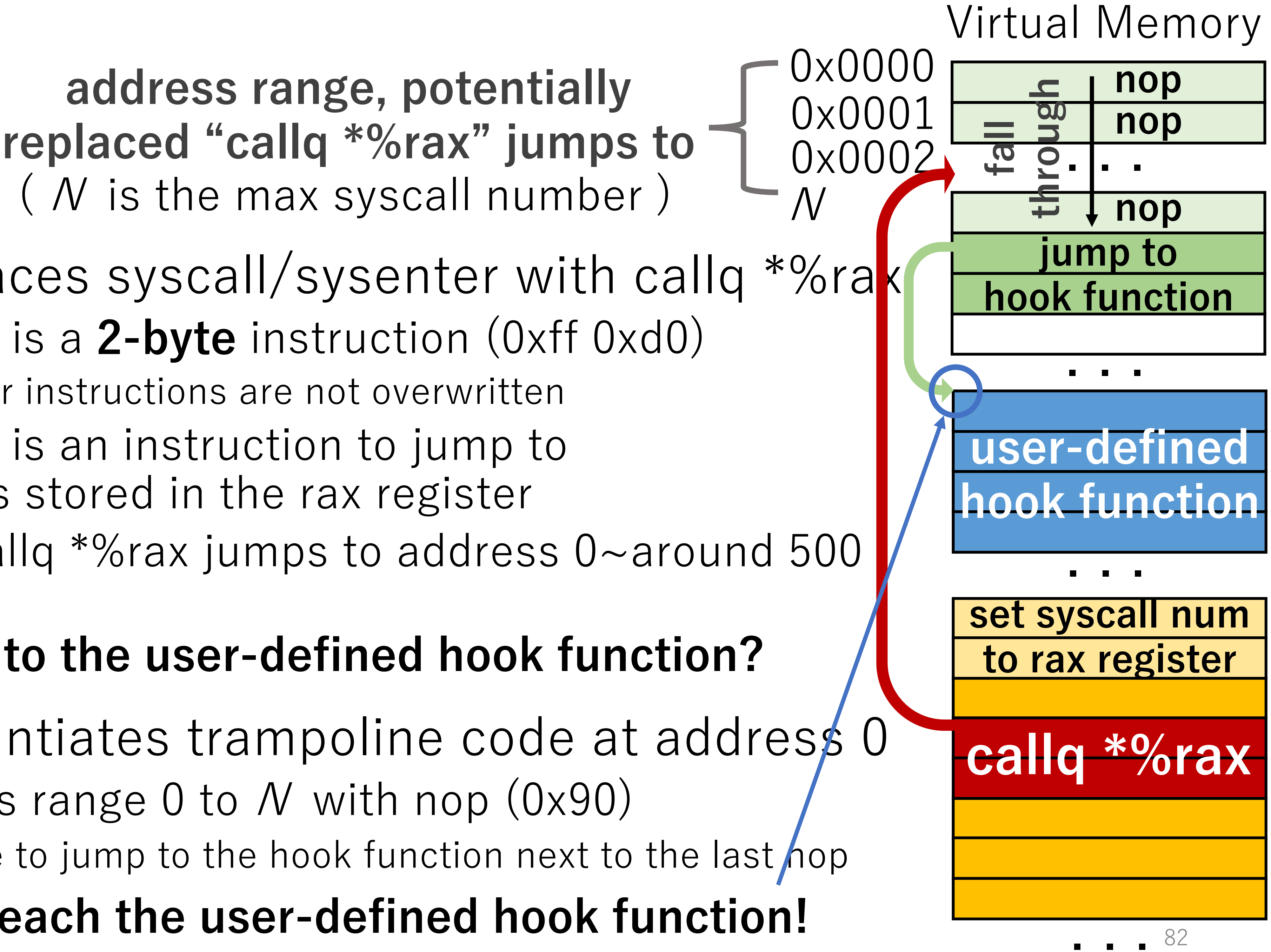
zpoline

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register
  - replaced callq *%rax jumps to address 0~around 500

How to redirect to the user-defined hook function?

- zpoline instantiates trampoline code at address 0
  - fills address range 0 to *N* with nop (0x90)
    - puts code to jump to the hook function next to the last nop

We could reach the user-defined hook function!



Virtual Memory
address range, potentially replaced "callq *%rax" jumps to
(*N* is the max syscall number)
0x0000
0x0001
0x0002
N
nop
nop
fall through
. . .
nop
jump to hook function
. . .
user-defined hook function
. . .
set syscall num to rax register
callq *%rax
. . .

zpoline

← trampoline code at address 0 (zero)

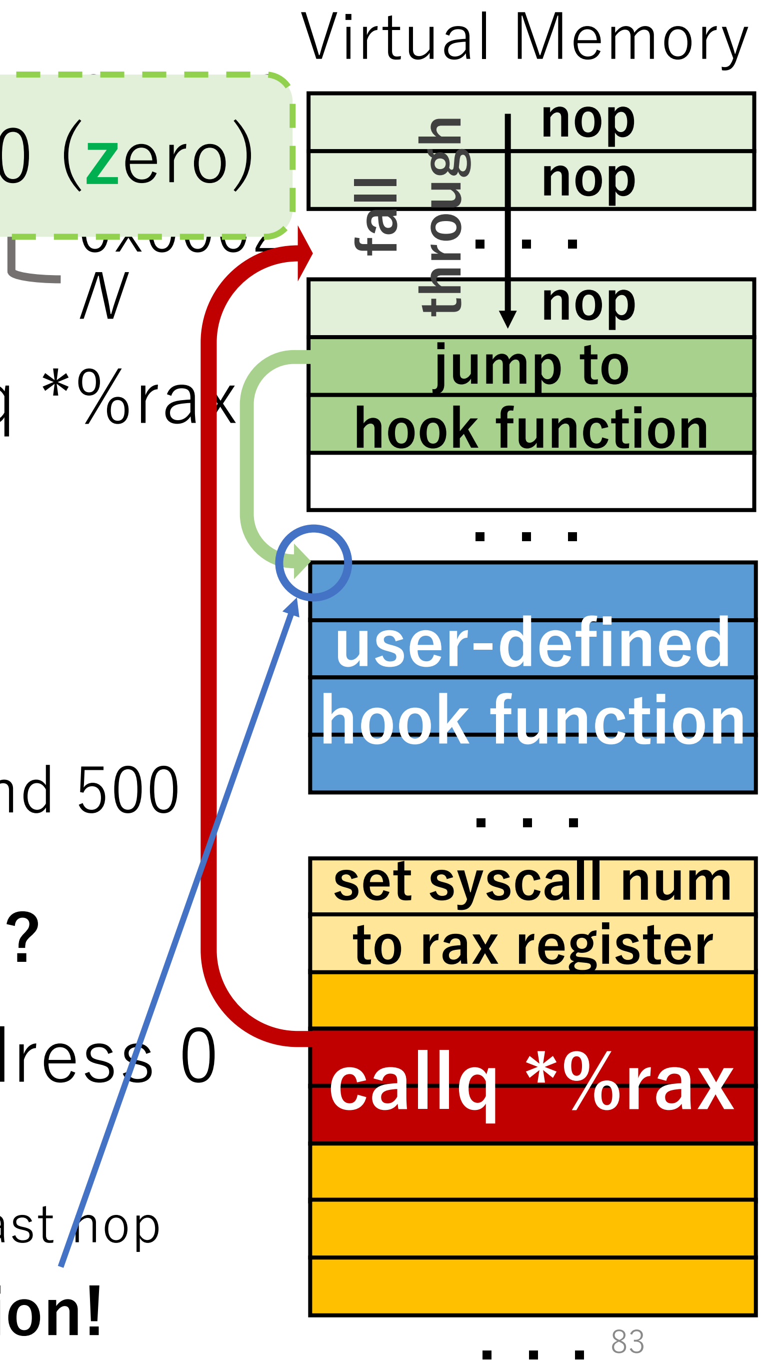
(*N* is the max syscall number)

- zpoline replaces syscall/sysenter with callq *%rax
  - callq *%rax is a **2-byte** instruction (0xff 0xd0)
    - Neighbour instructions are not overwritten
  - callq *%rax is an instruction to jump to the address stored in the rax register
  - replaced callq *%rax jumps to address 0~around 500

How to redirect to the user-defined hook function?

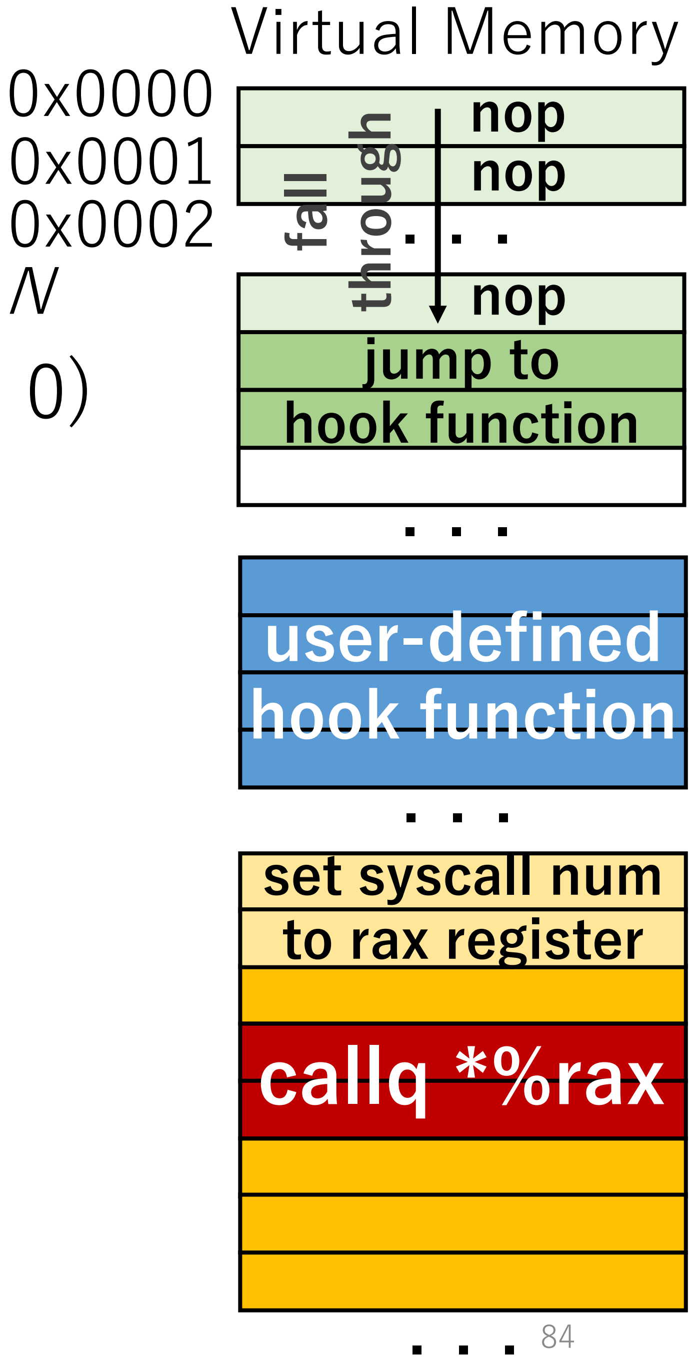
- zpoline instantiates trampoline code at address 0
  - fills address range 0 to *N* with nop (0x90)
    - puts code to jump to the hook function next to the last nop

We could reach the user-defined hook function!



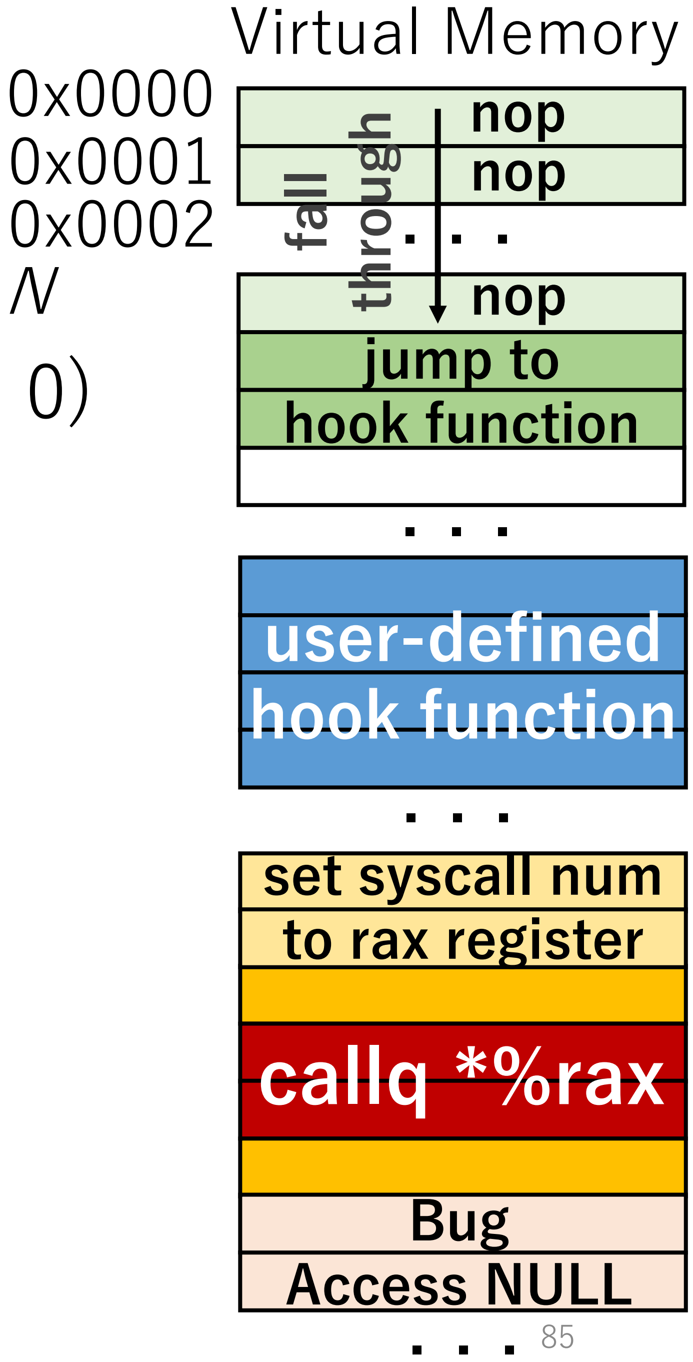
NULL Access Termination

- A buggy program may access NULL (address 0)



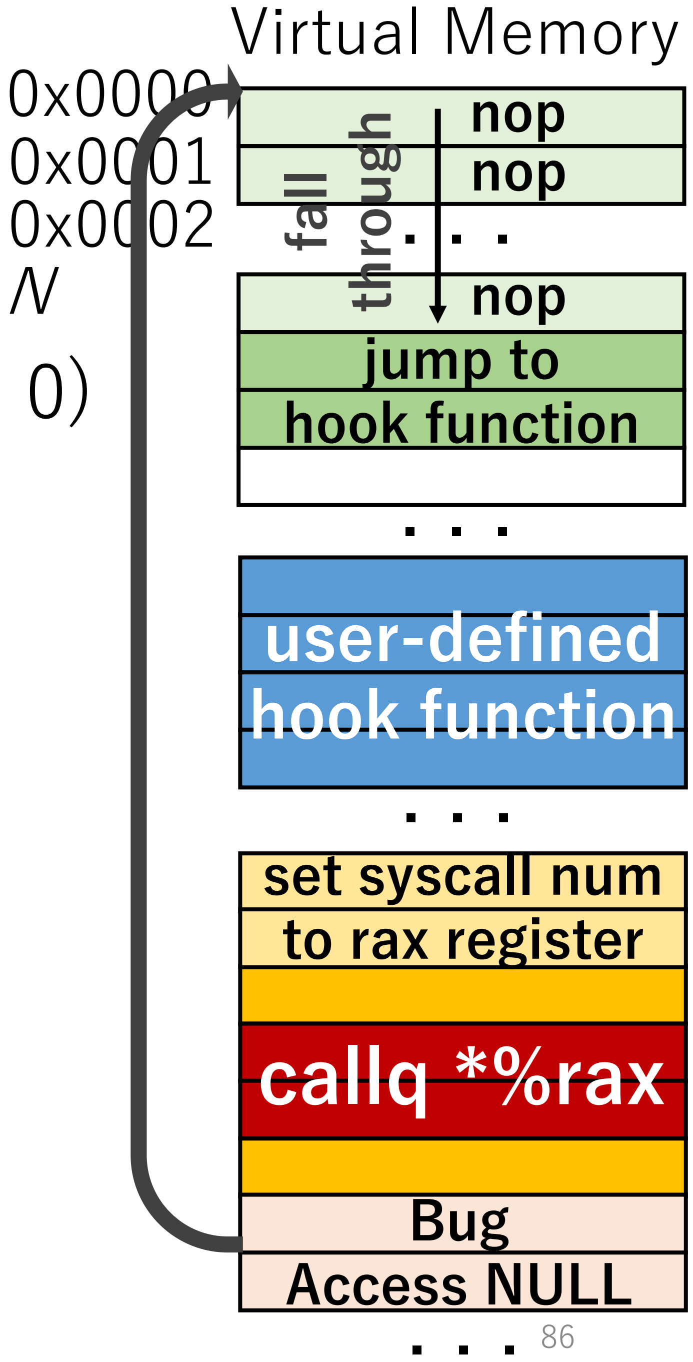
NULL Access Termination

- A buggy program may access NULL (address 0)



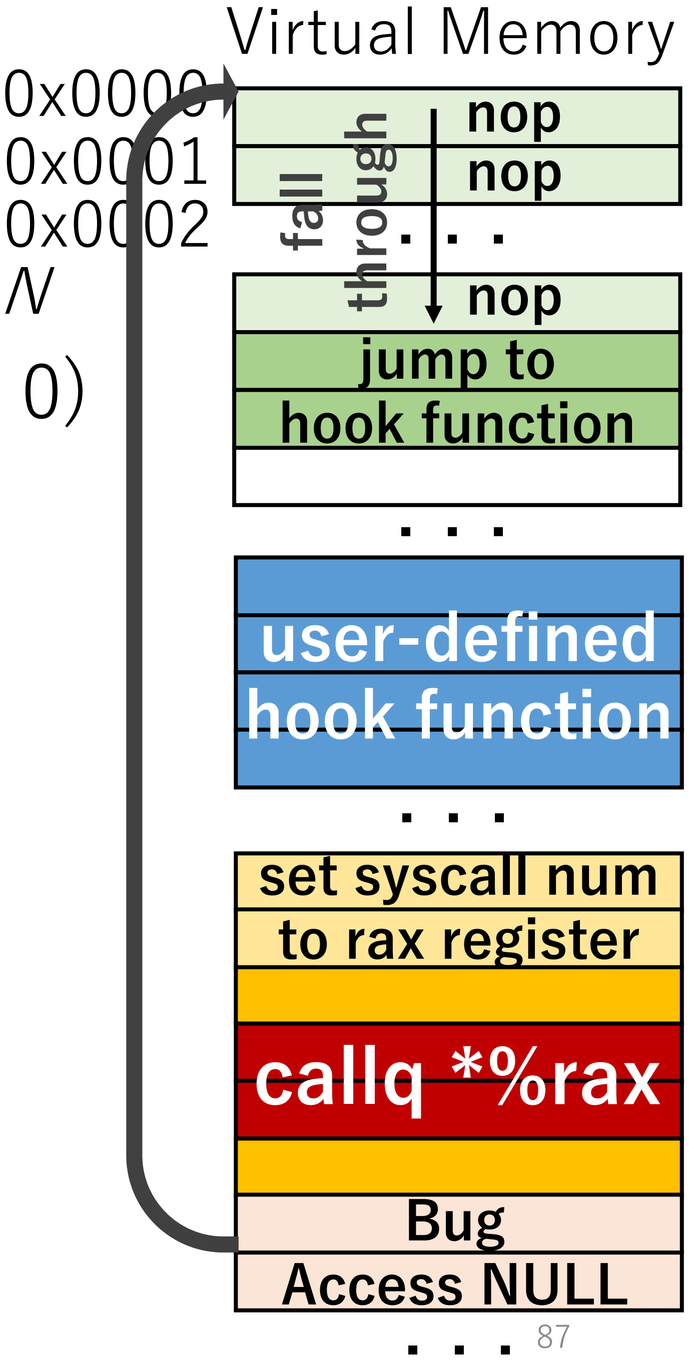
NULL Access Termination

- A buggy program may access NULL (address 0)



NULL Access Termination

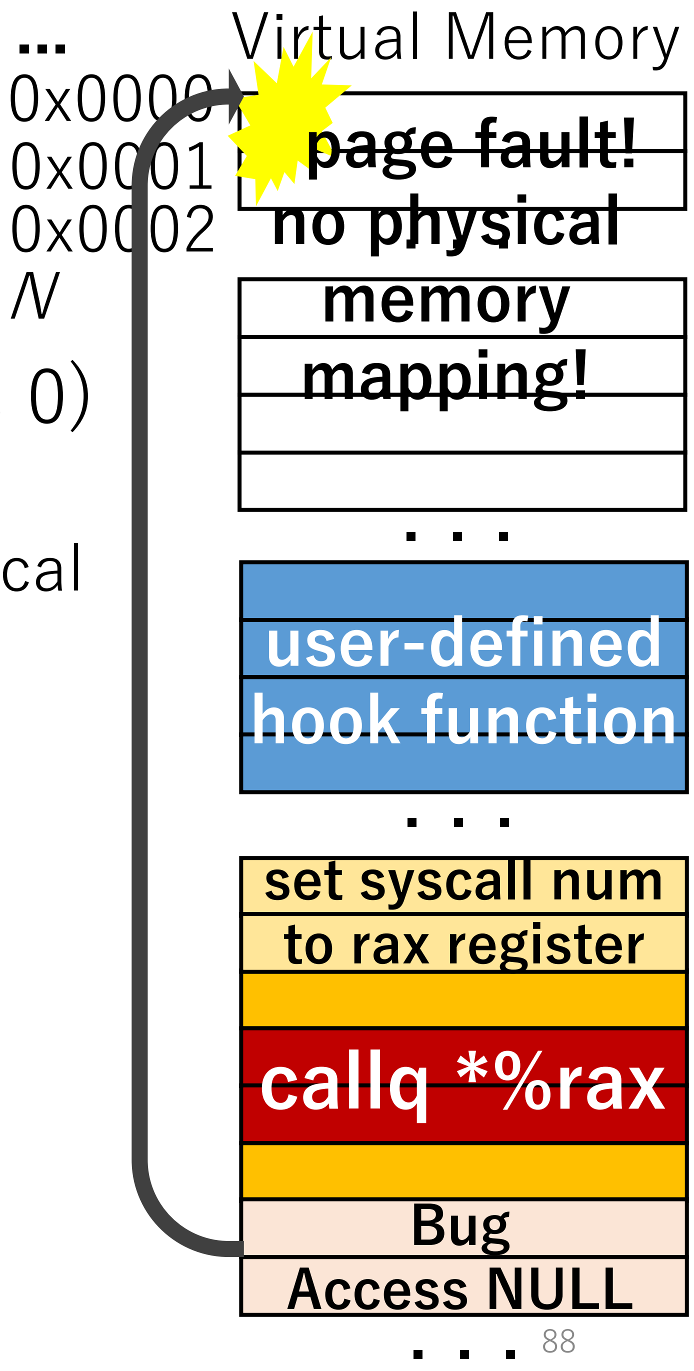
- A buggy program may access NULL (address 0)
  - In principle, NULL access has to be terminated



NULL Access Termination

- A buggy program may access NULL (address 0)
  - In principle, NULL access has to be terminated
  - Normally, a page fault happens because no physical memory is mapped to virtual address 0

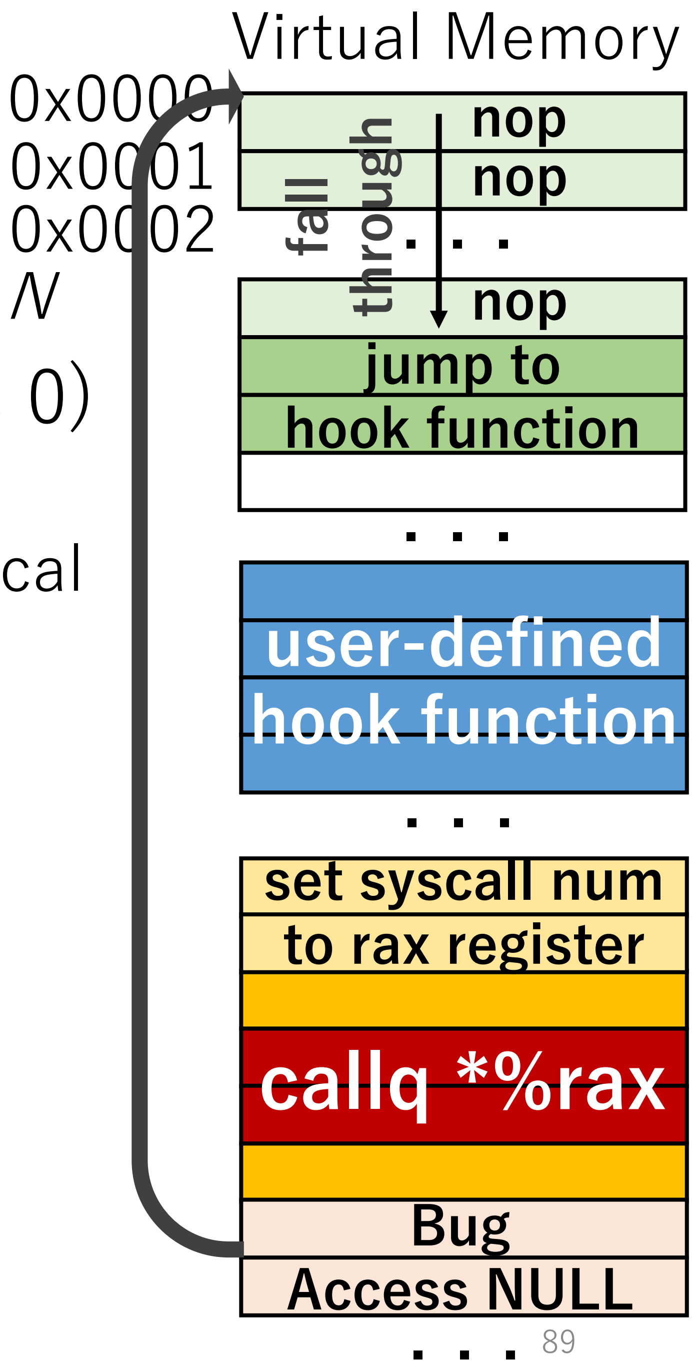
Normally, ...



Virtual Memory
0x0000
0x0001
0x0002
N
page fault!
no physical memory mapping!
. . .
user-defined hook function
. . .
set syscall num to rax register
callq *%rax
Bug
Access NULL
. . .

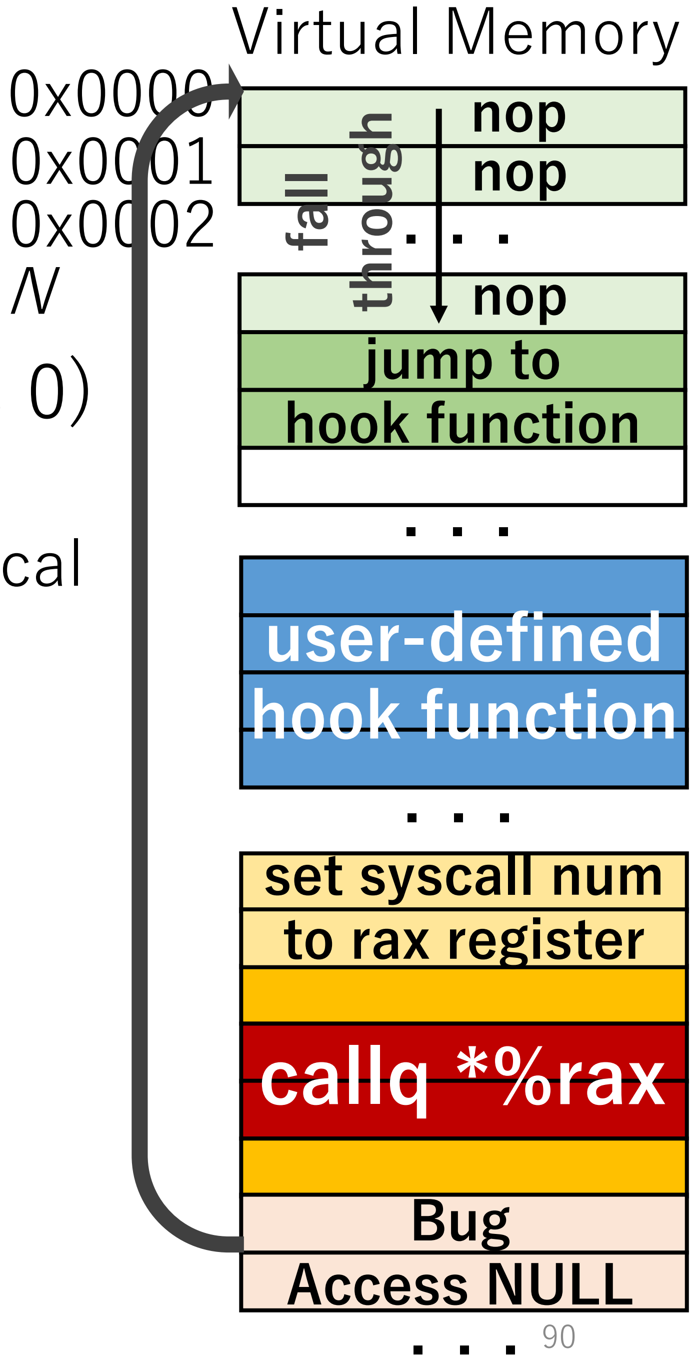
NULL Access Termination

- A buggy program may access NULL (address 0)
  - In principle, NULL access has to be terminated
  - Normally, a page fault happens because no physical memory is mapped to virtual address 0
  - zpoline uses virtual address 0, therefore, the page fault does not happen



NULL Access Termination

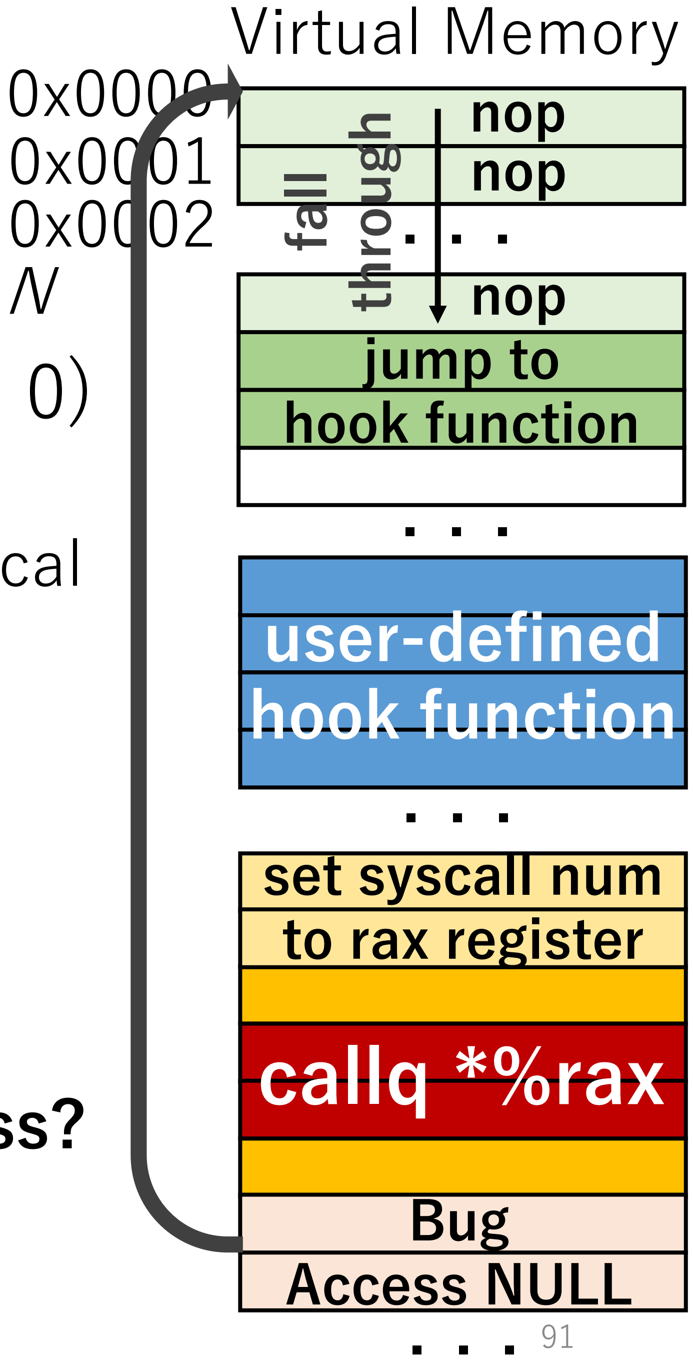
- A buggy program may access NULL (address 0)
  - In principle, NULL access has to be terminated
  - Normally, a page fault happens because no physical memory is mapped to virtual address 0
  - zpoline uses virtual address 0, therefore, the page fault does not happen
  - The buggy program continues to run



NULL Access Termination

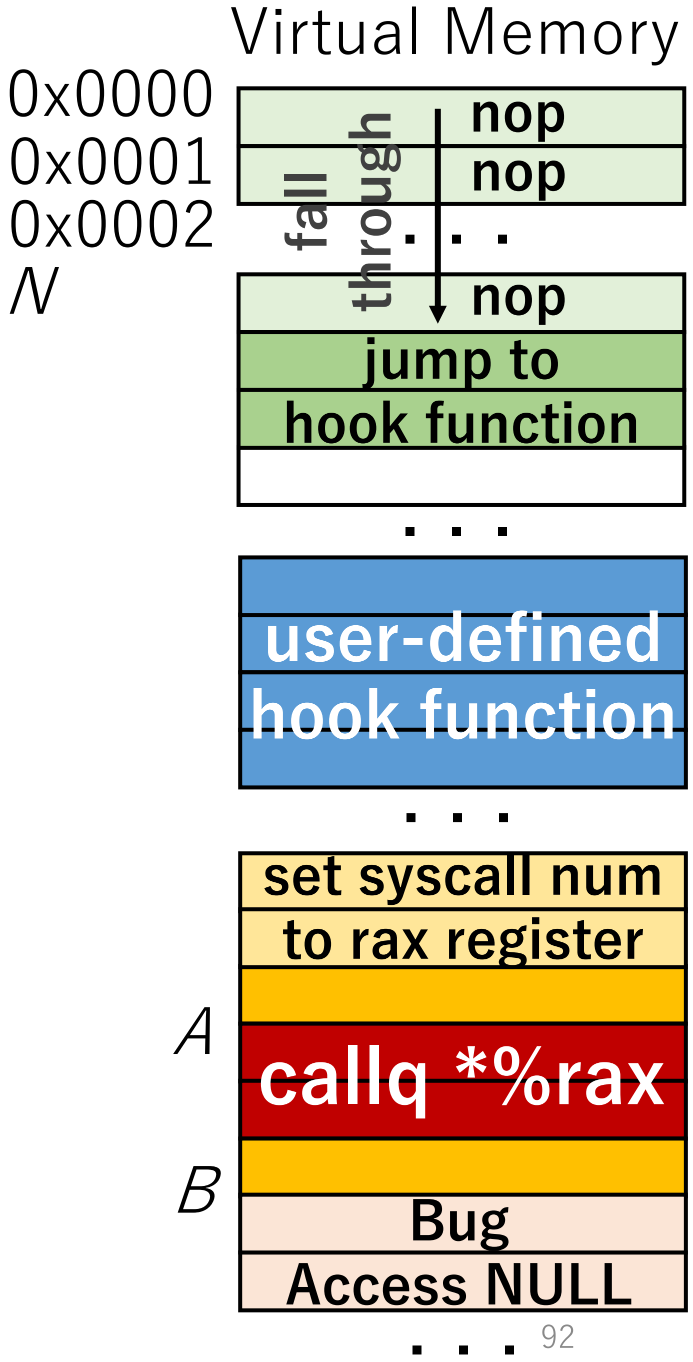
- A buggy program may access NULL (address 0)
  - In principle, NULL access has to be terminated
  - Normally, a page fault happens because no physical memory is mapped to virtual address 0
  - zpoline uses virtual address 0, therefore, the page fault does not happen
  - The buggy program continues to run

How can we detect and terminate a buggy NULL access?



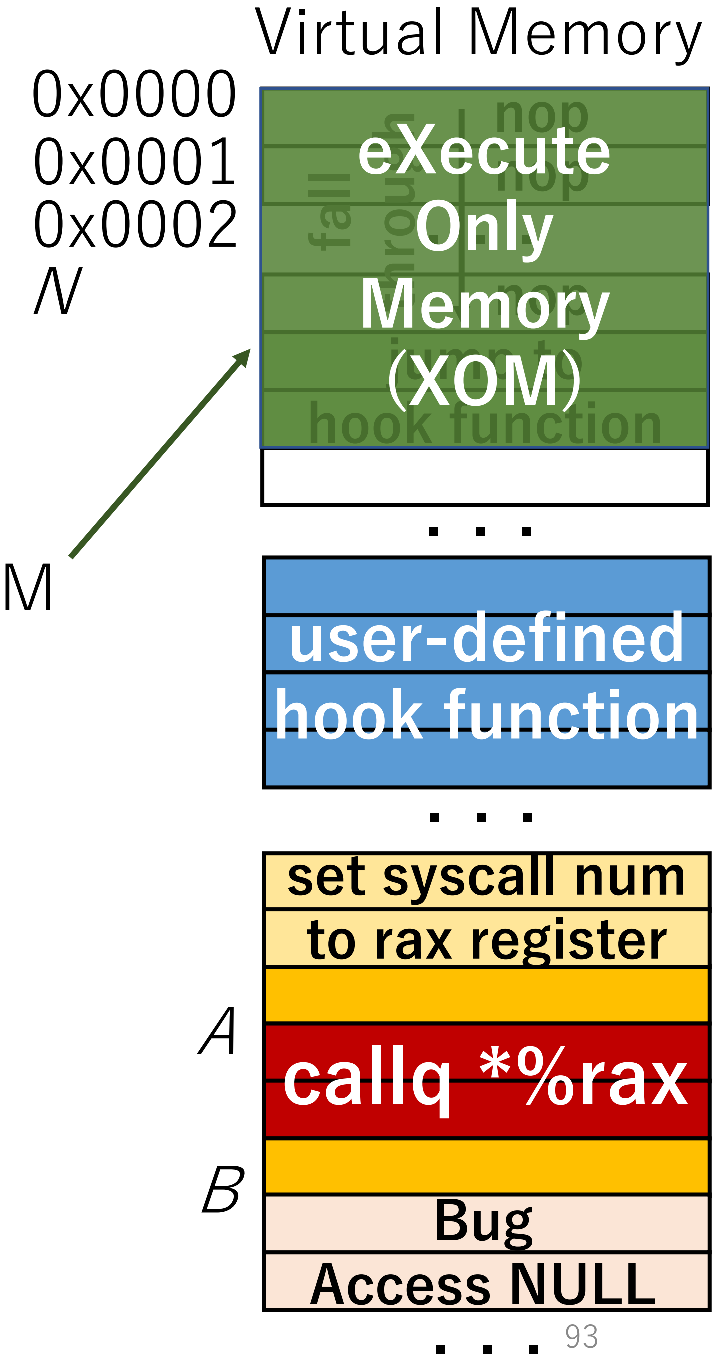
NULL Access Termination

- Memory access: read / write / execute



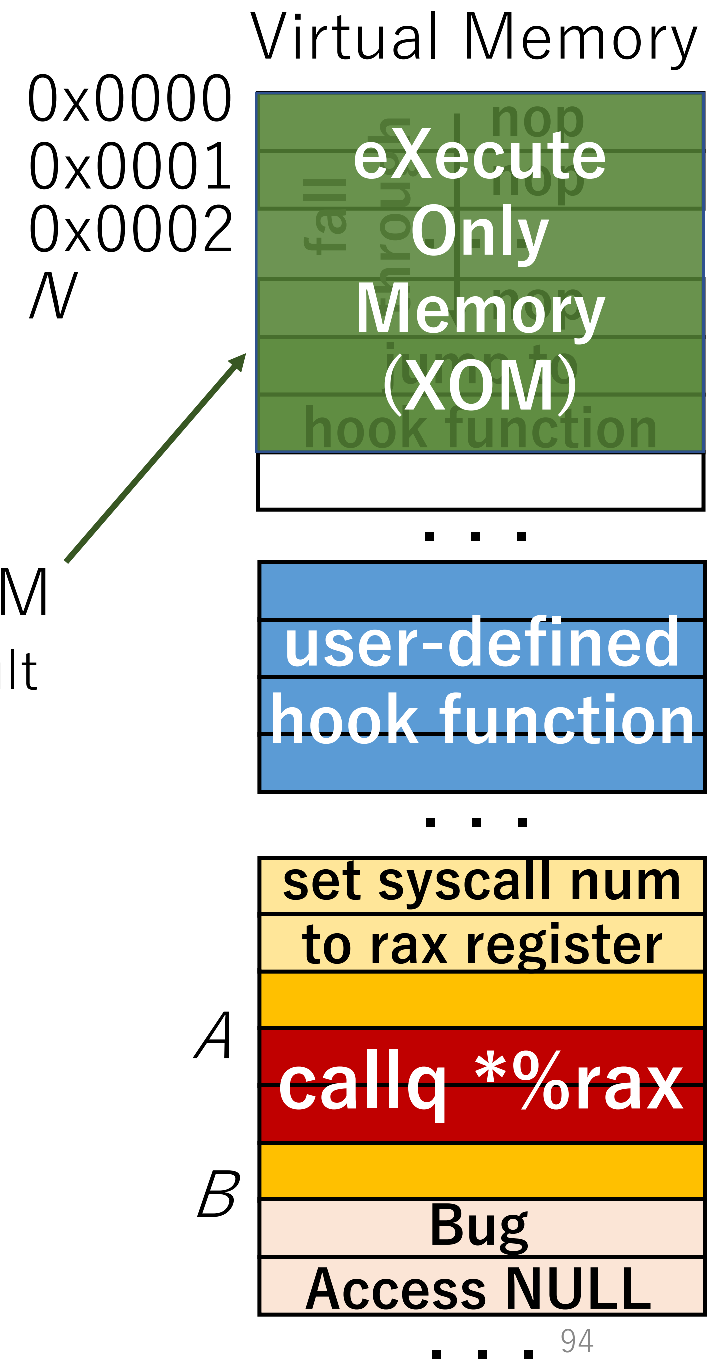
NULL Access Termination

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM



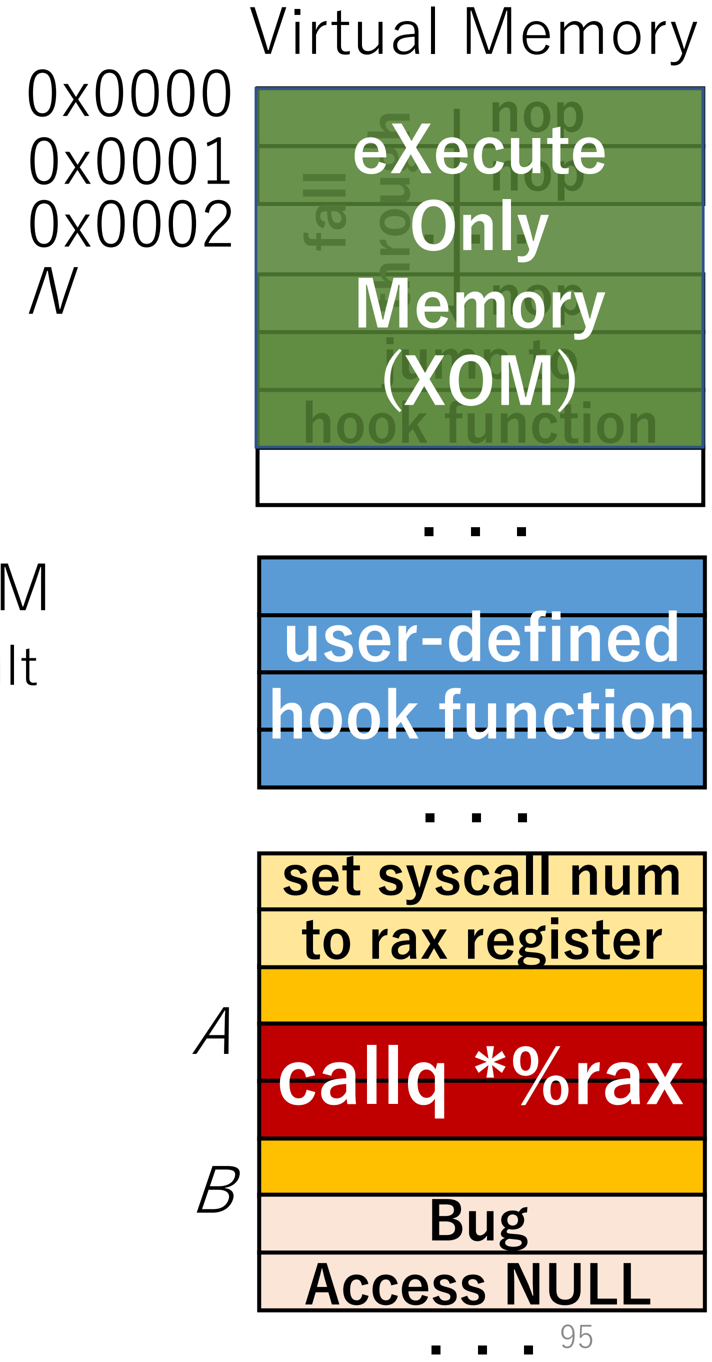
NULL Access Termination

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call



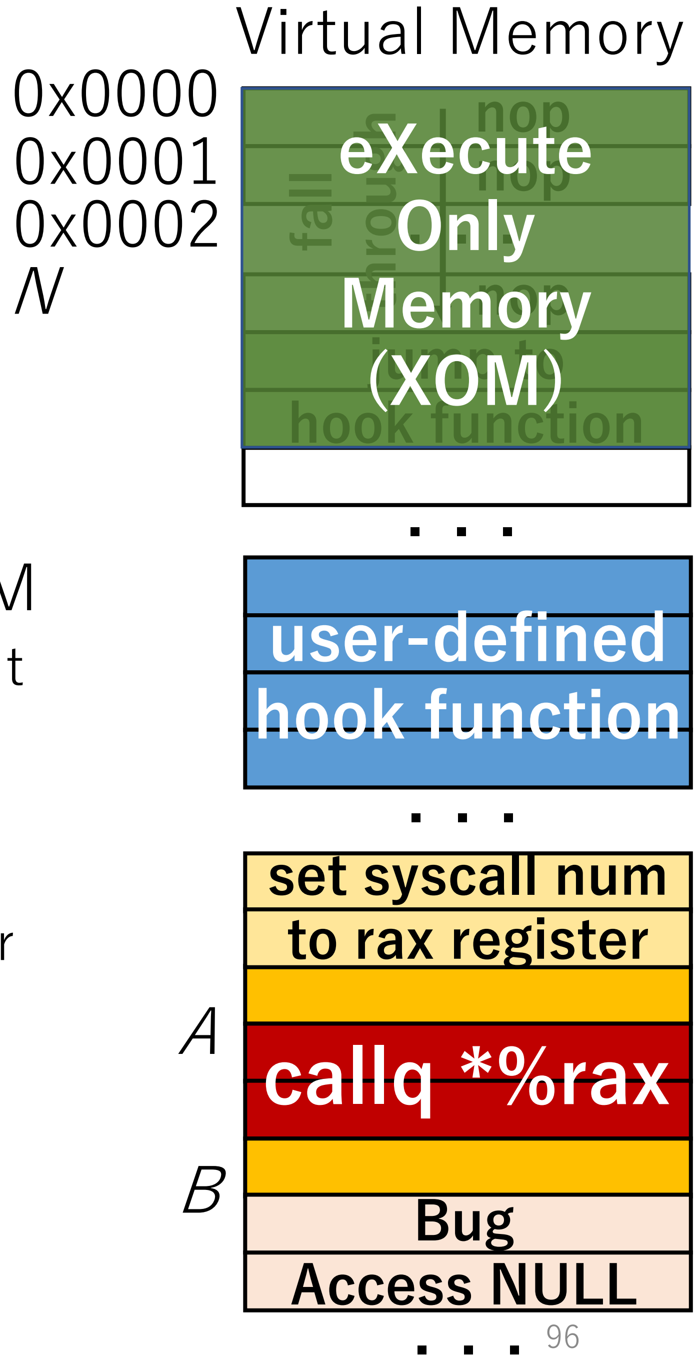
NULL Access Termination

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address



NULL Access Termination

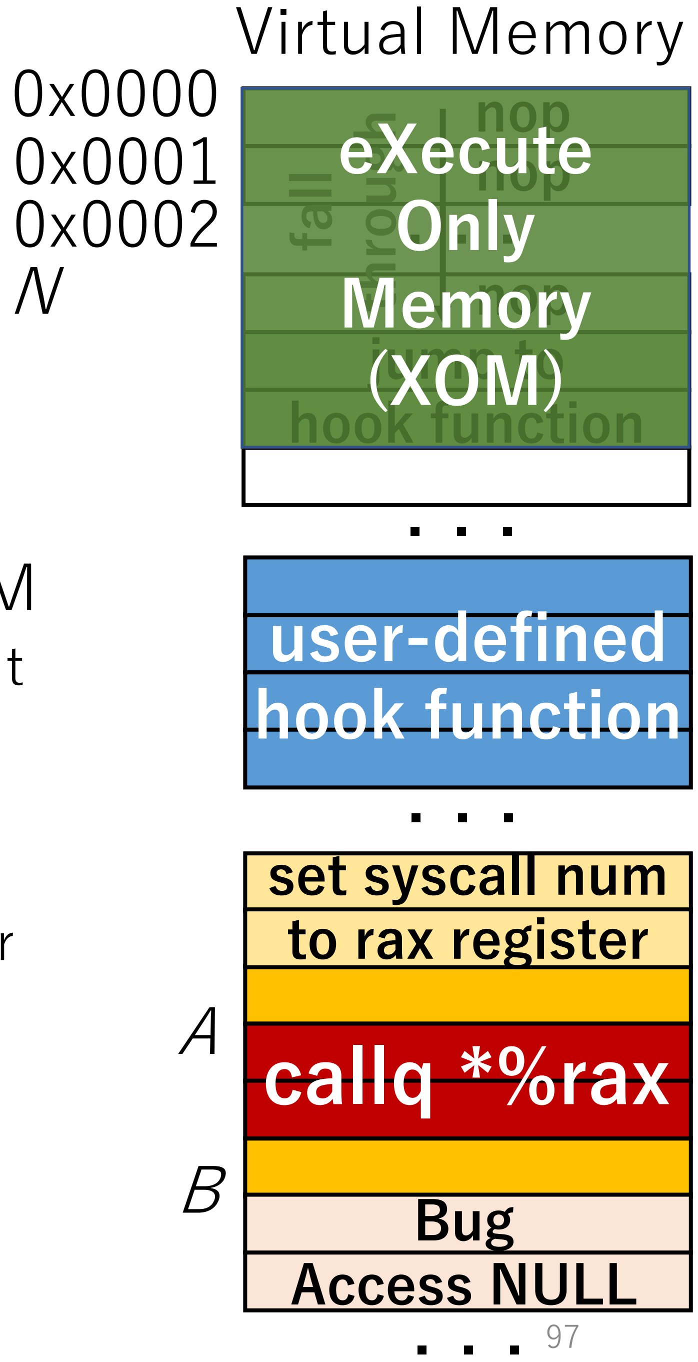
- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase,
       we collect the addresses of replaced syscall/sysenter



NULL Access Termination

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase,
       we collect the addresses of replaced syscall/sysenter

During binary rewriting phase …

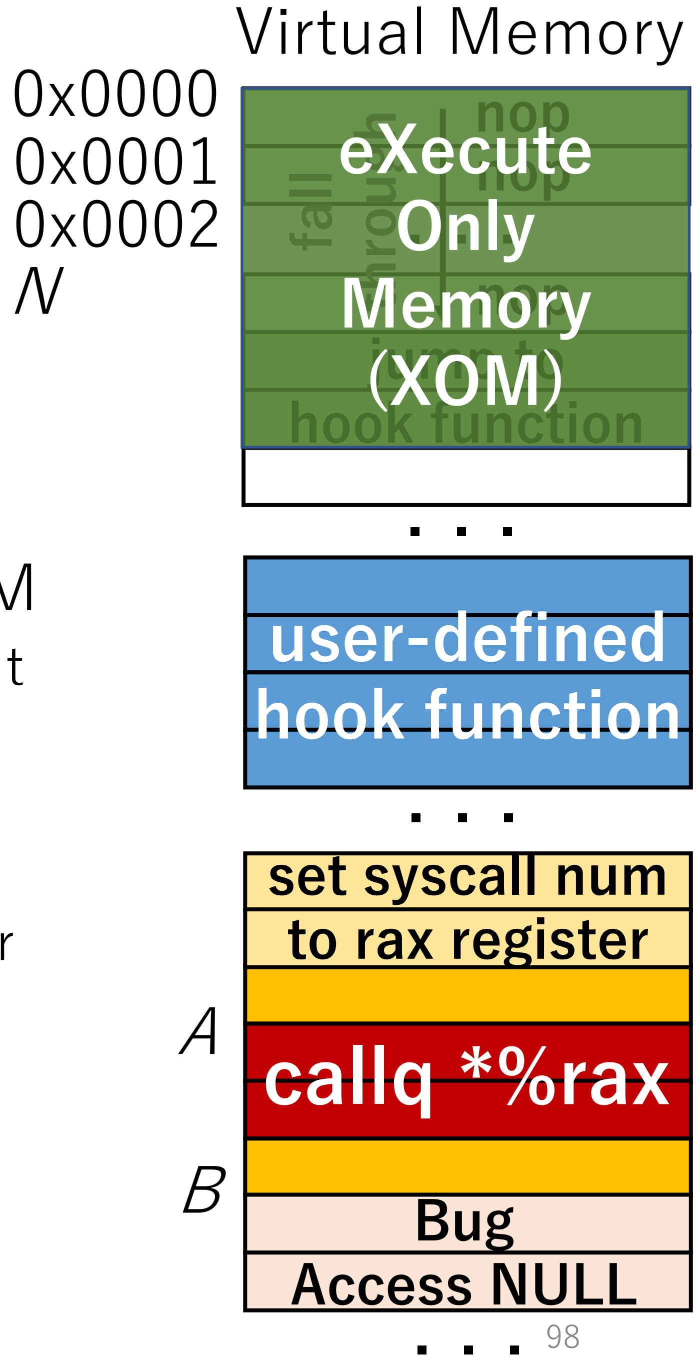


NULL Access Termination

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase,
       we collect the addresses of replaced syscall/sysenter

During binary rewriting phase …

List of replaced addresses : […]

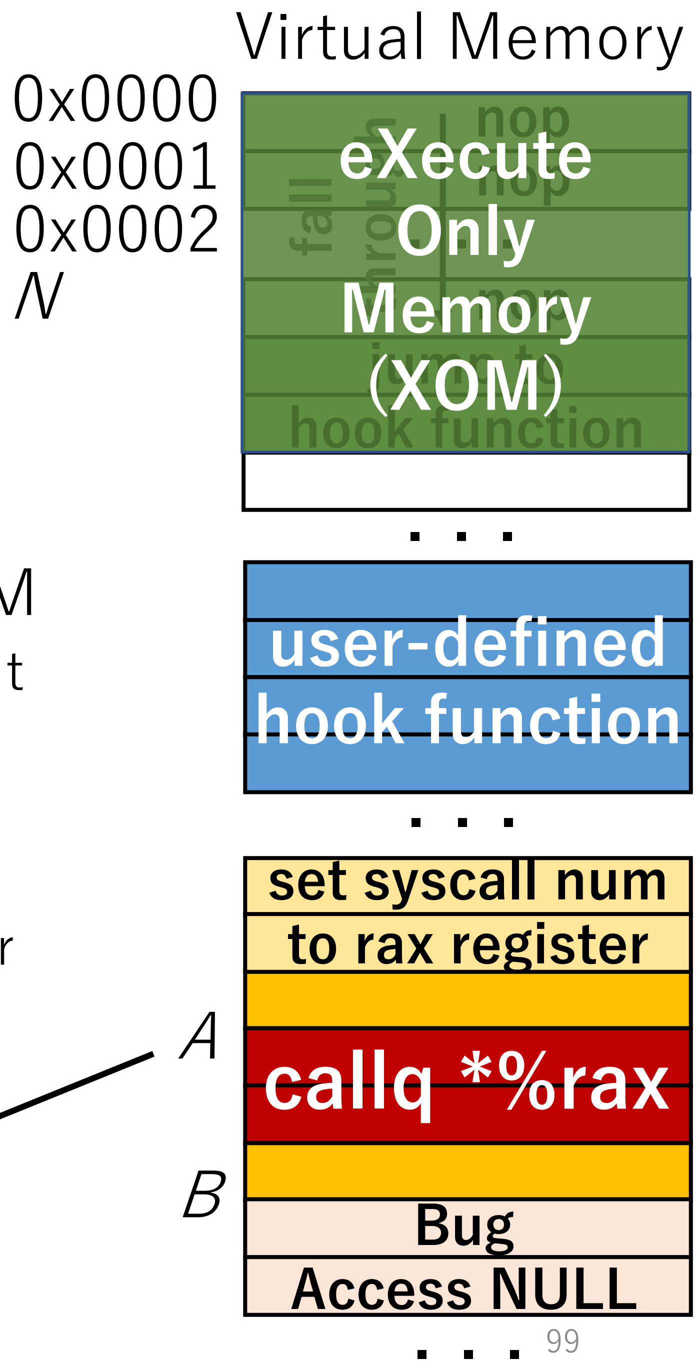


NULL Access Termination

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase,
       we collect the addresses of replaced syscall/sysenter

During binary rewriting phase …

List of replaced addresses : […]

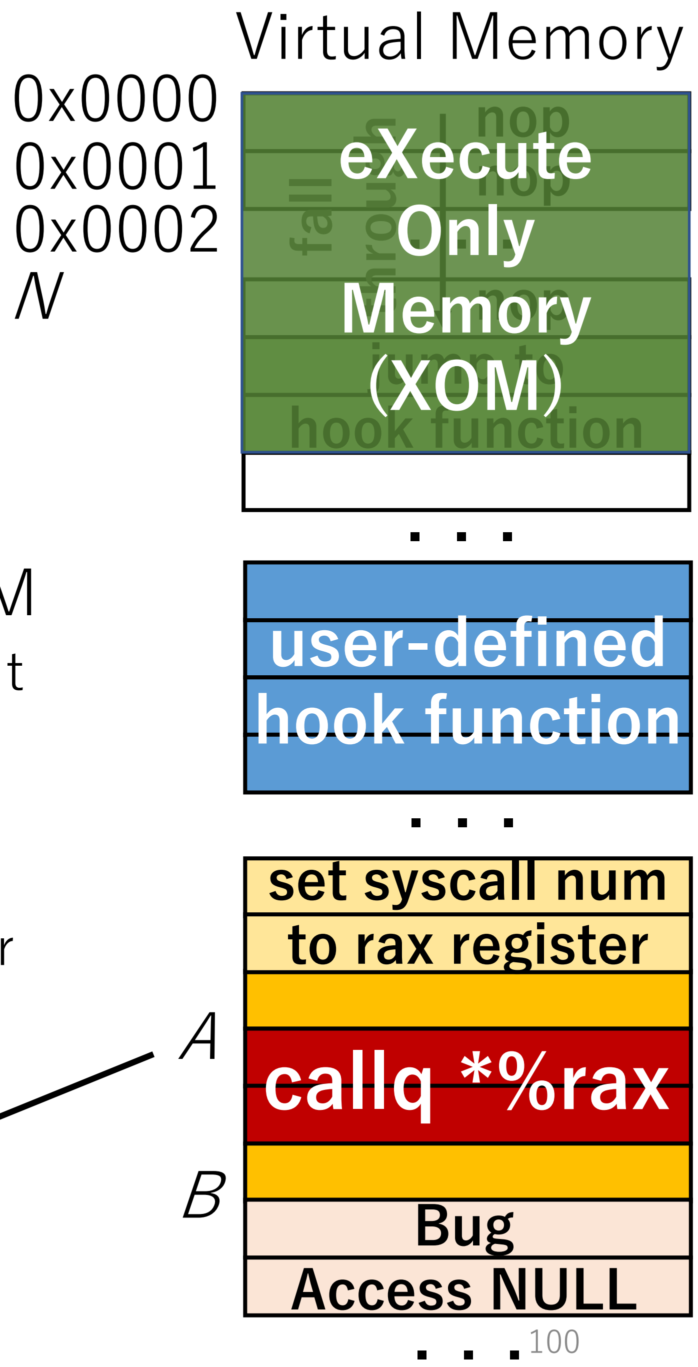


NULL Access Termination

- Memory access: read / write / execute
- Solution
    - read/write: configure the trampoline code as XOM
        - read/write access to the trampoline code causes a fault
            - This can be done by mprotect() system call
    - execute: check the caller address
        1. during the binary rewriting phase,
           we collect the addresses of replaced syscall/sysenter

During binary rewriting phase ...

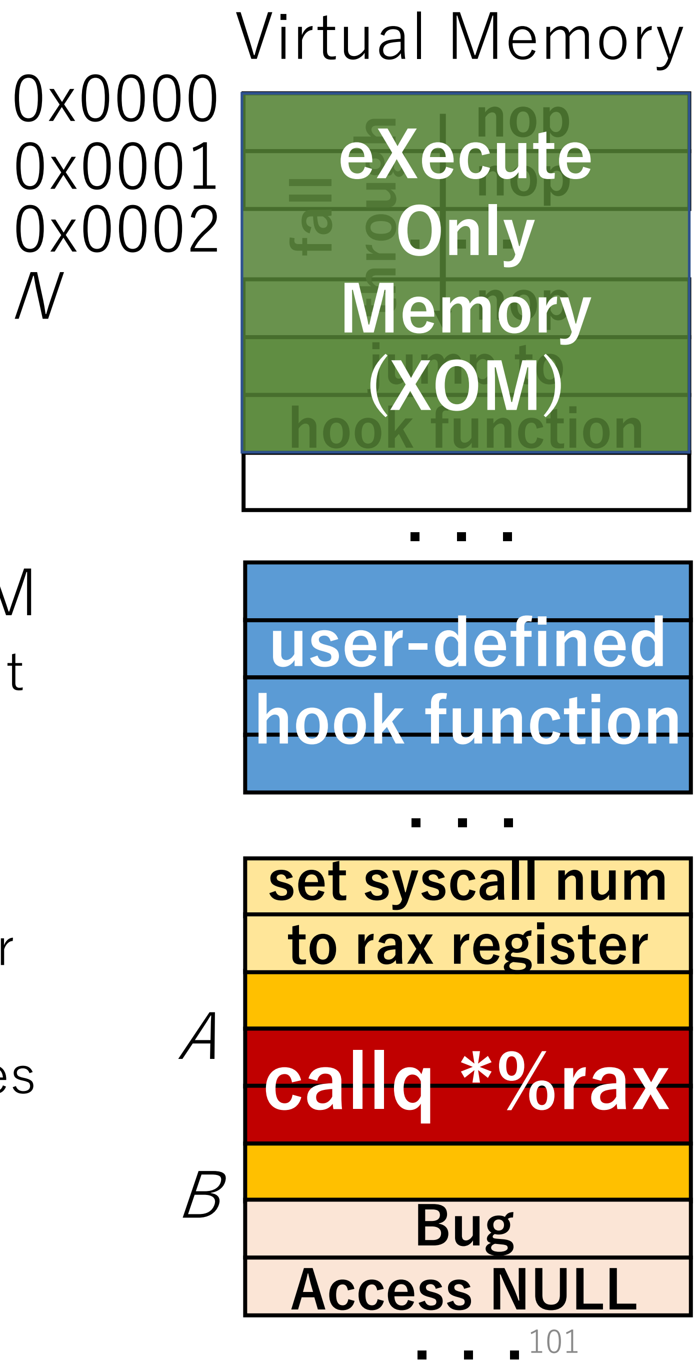
List of replaced addresses : [*A* , ...]



NULL Access Termination

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase,
       we collect the addresses of replaced syscall/sysenter
    2. at runtime, in the hook function,
       we check if the caller is one of the replaced addresses

List of replaced addresses : [*A* , ...]



Virtual Memory

0x0000
0x0001
0x0002
N

nop
nop
fall through
nop
jump to
hook function

eXecute Only Memory (XOM)

. . .

user-defined hook function

. . .

set syscall num to rax register

A

callq *%rax

B

Bug
Access NULL

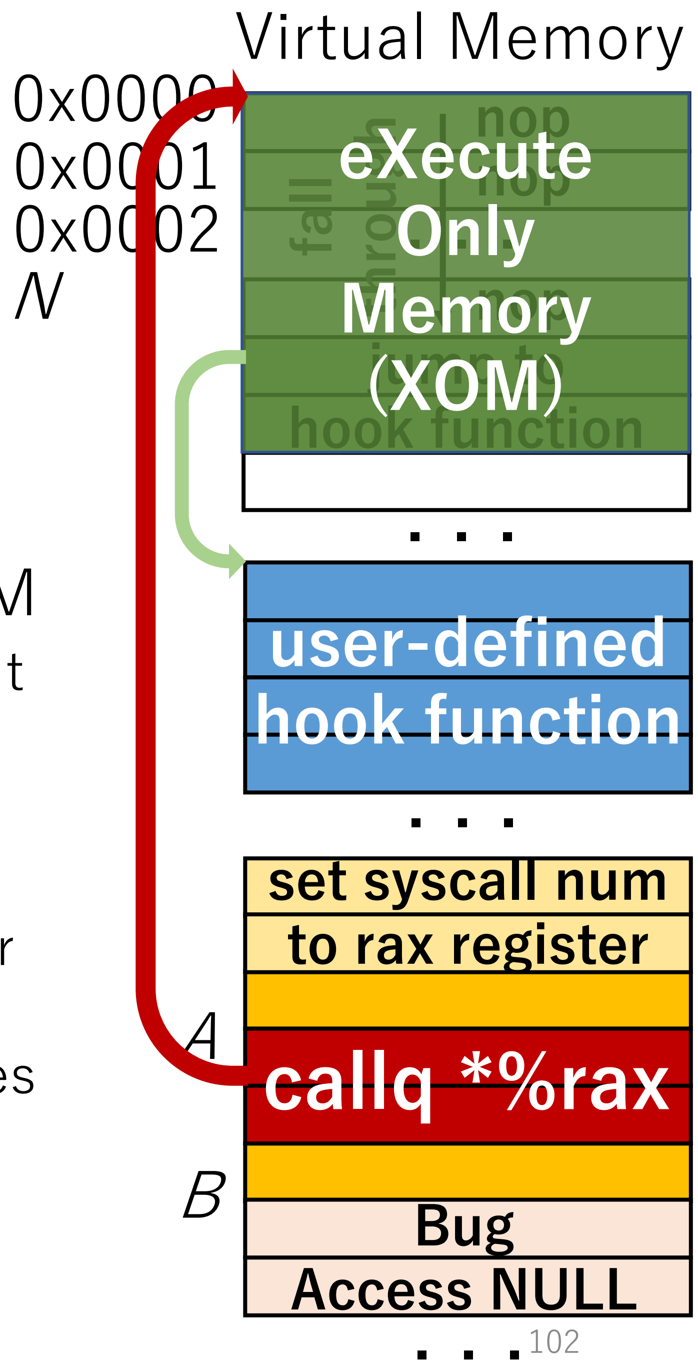
. . .

NULL Access Termination

At runtime ...

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase,
       we collect the addresses of replaced syscall/sysenter
    2. at runtime, in the hook function,
       we check if the caller is one of the replaced addresses

List of replaced addresses : [*A* , ...]



Virtual Memory

0x0000
0x0001
0x0002
N

nop
nop
nop
jump to
hook function
fall through

eXecute Only Memory (XOM)

. . .

user-defined hook function

. . .

set syscall num to rax register

A

callq *%rax

B

Bug
Access NULL

. . .

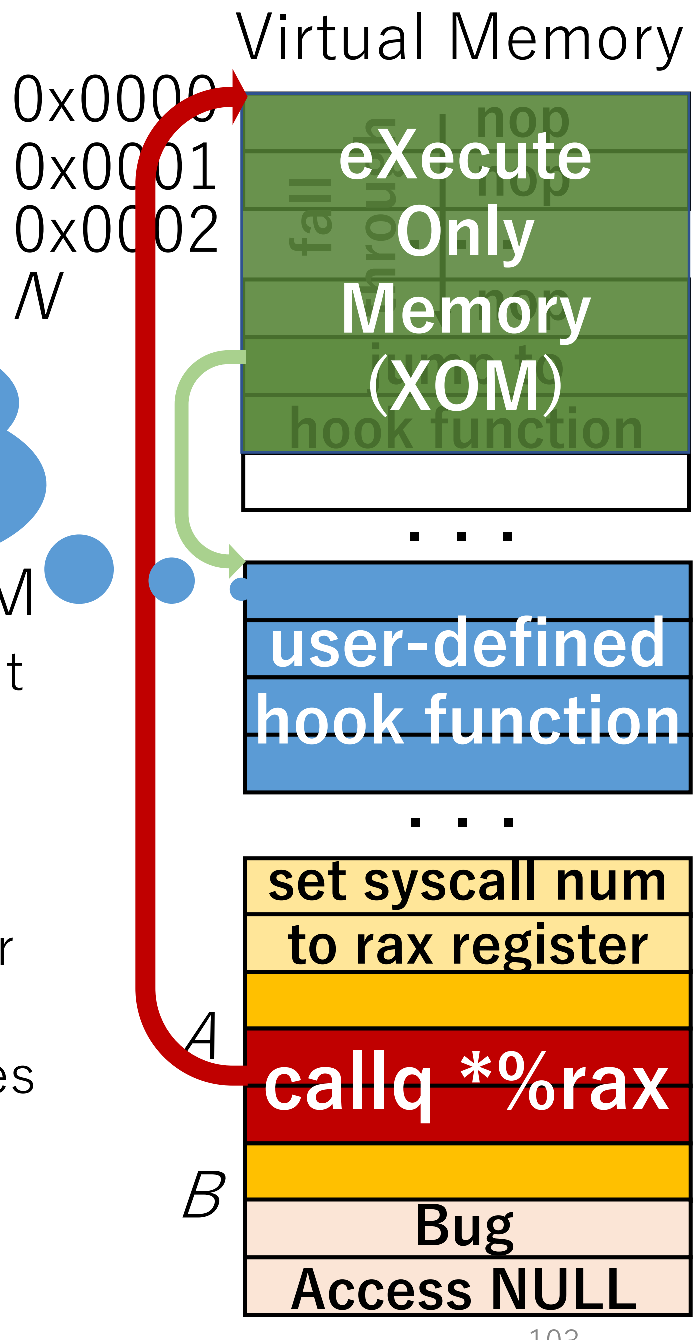
NULL Access Termination

At runtime ...

- Memory acces
- Solution
  - read/write. XOM
    - read/write access to the tra code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase, we collect the addresses of replaced syscall/sysenter
    2. at runtime, in the hook function, we check if the caller is one of the replaced addresses

The caller address is *A*
*A* is in the list, so
this is a _valid_ access

List of replaced addresses : [*A* , ...]

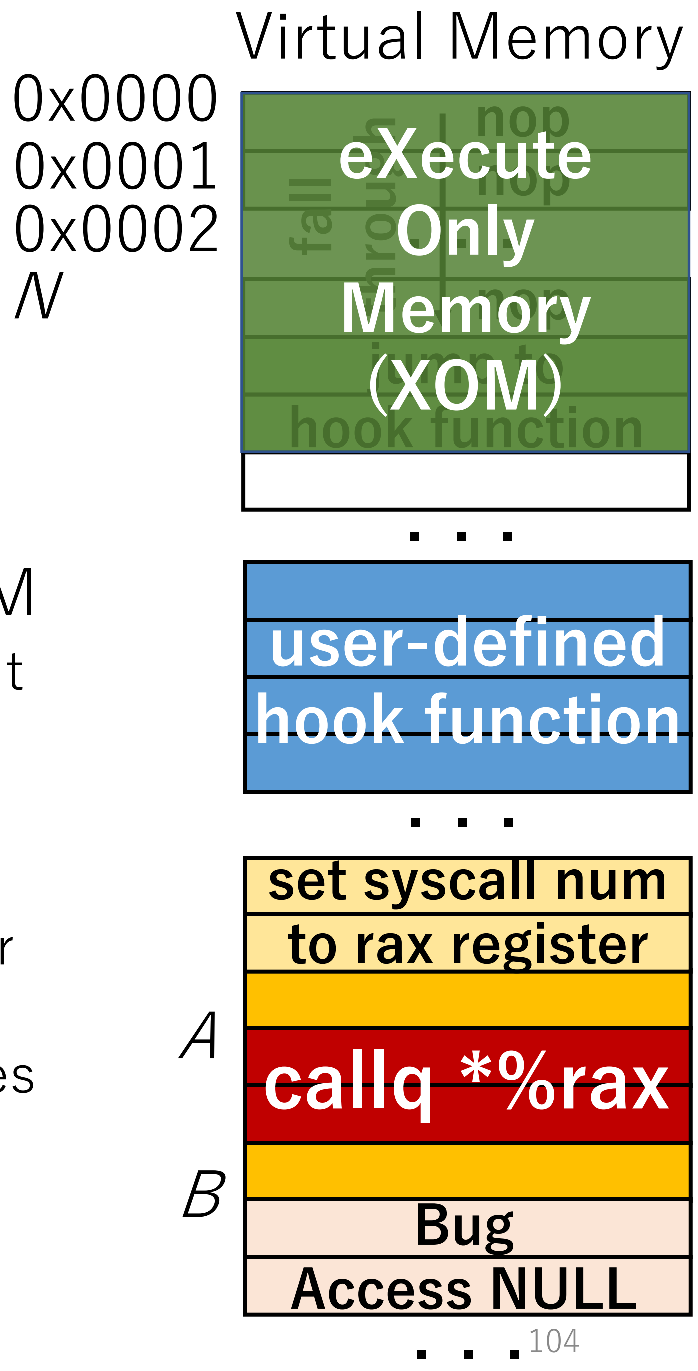


NULL Access Termination

At runtime …

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase,
       we collect the addresses of replaced syscall/sysenter
    2. at runtime, in the hook function,
       we check if the caller is one of the replaced addresses

List of replaced addresses : [*A* , ...]

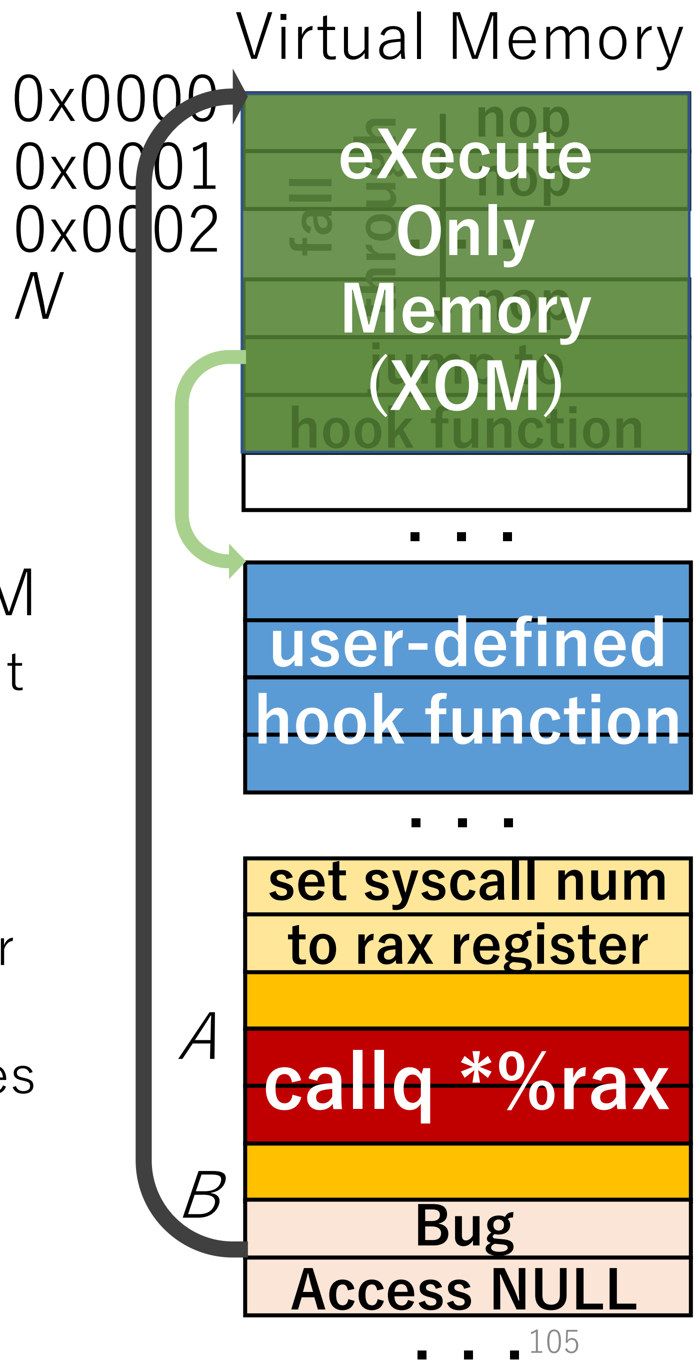


NULL Access Termination

At runtime …

- Memory access: read / write / execute
- Solution
  - read/write: configure the trampoline code as XOM
    - read/write access to the trampoline code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase,
       we collect the addresses of replaced syscall/sysenter
    2. at runtime, in the hook function,
       we check if the caller is one of the replaced addresses

List of replaced addresses : [*A* , …]



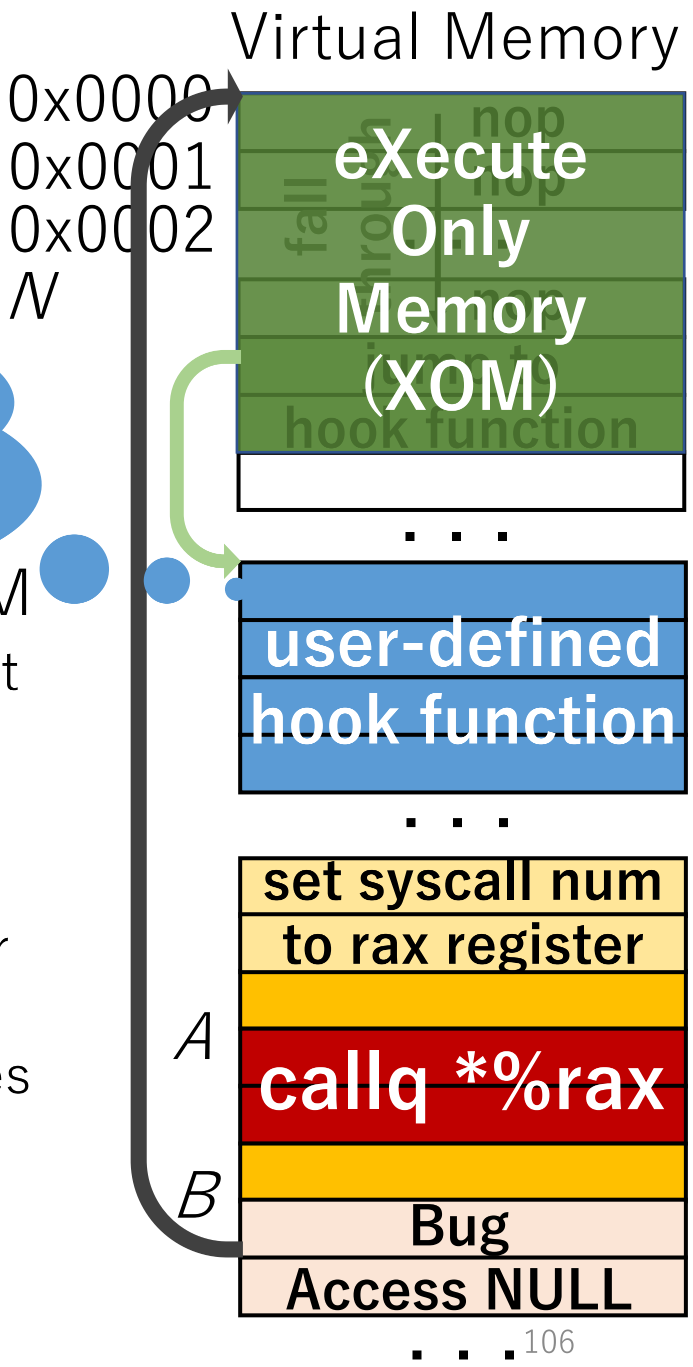
NULL Access Termination

At runtime ...

- Memory acces
- Solution
  - read/write: XOM
    - read/write access to the tra ode causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase,
       we collect the addresses of replaced syscall/sysenter
    2. at runtime, in the hook function,
       we check if the caller is one of the replaced addresses

The caller address is *B*
<u>*B* is NOT in the list</u>, so
this is an <u>invalid</u> access

List of replaced addresses : [*A* , ...]



NULL Access Termination

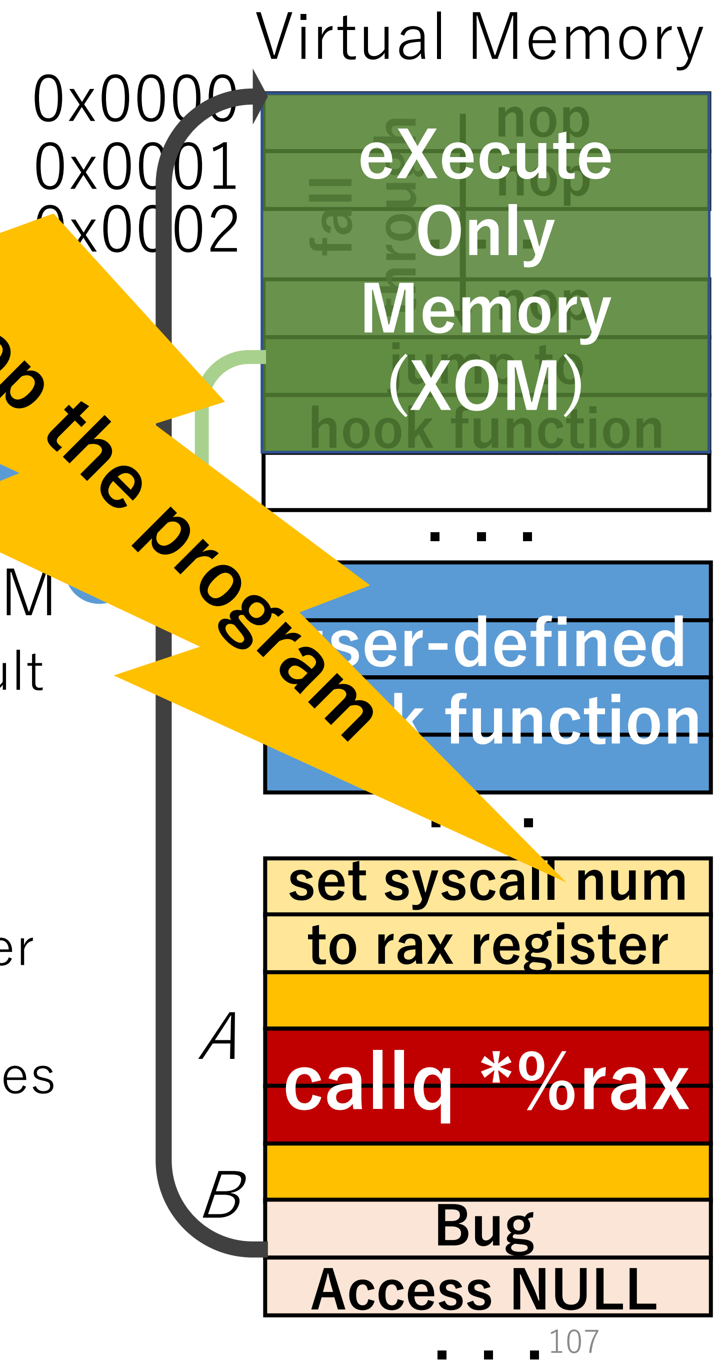
At runtime ...

- Memory acces
- Solution
  - read/write. XOM
    - read/write access to the tra ode causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase,
       we collect the addresses of replaced syscall/sysenter
    2. at runtime, in the hook function,
       we check if the caller is one of the replaced addresses

The caller address is *B*
B is NOT in the list, so
this is an invalid access

stop the program

List of replaced addresses : [*A* , ...]



NULL Access Termination

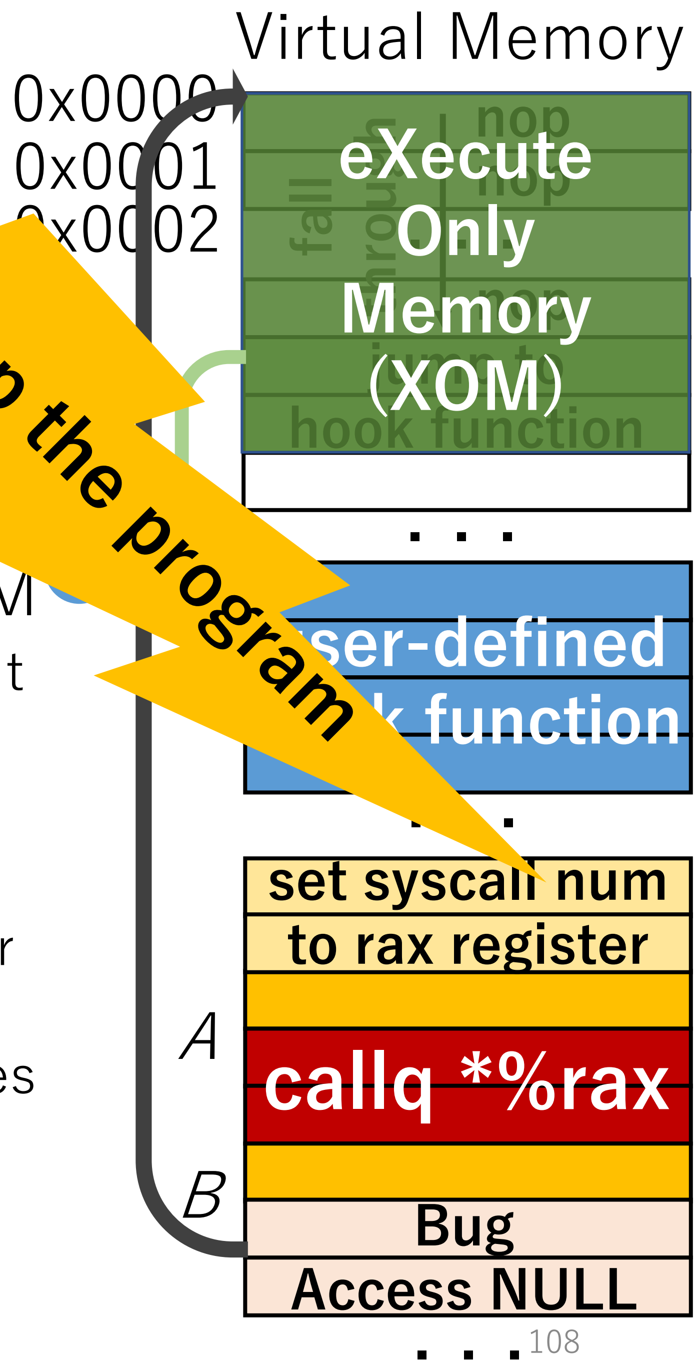
At runtime ...

- Memory acces
- Solution
  - read/write. XOM
    - read/write access to the tra code causes a fault
      - This can be done by mprotect() system call
  - execute: check the caller address
    1. during the binary rewriting phase, we collect the addresses of replaced syscall/sysenter
    2. at runtime, in the hook function, we check if the caller is one of the replaced addresses
       - Current prototype uses bitmap to implement this check

List of replaced addresses : [*A* , ...]

The caller address is *B*
B is NOT in the list, so
this is an invalid access

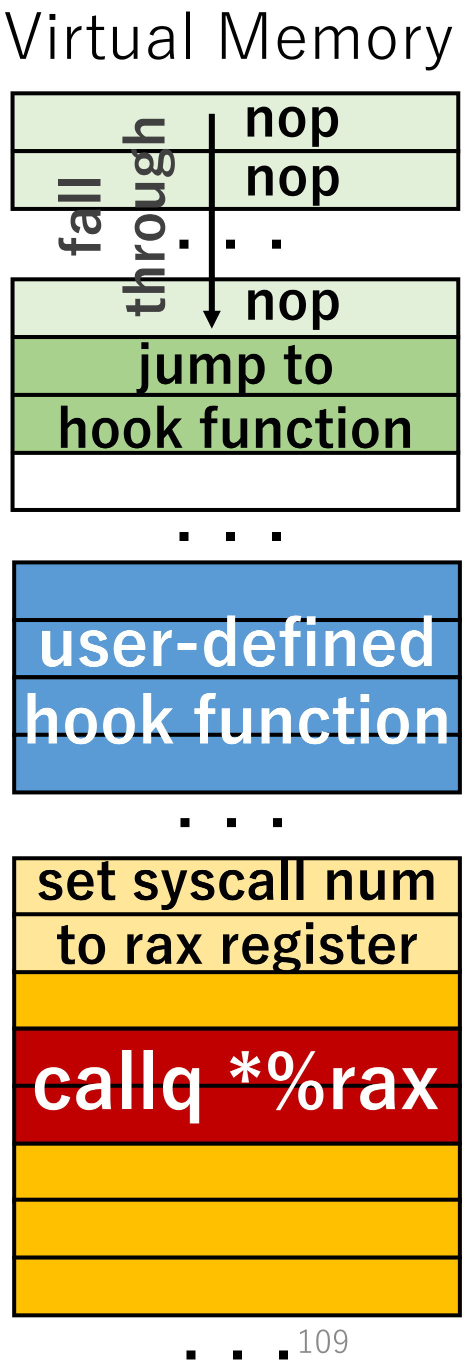
stop the program



System Call Hook Overhead

- Time to hook getpid() and return a dummy value

Mechanism	Time [ns]
ptrace	31201
int3 signaling	1342
SUD	1156
zpoline	41
LD_PRELOAD	6



System Call Hook Overhead

- Time to hook getpid() and return a dummy value

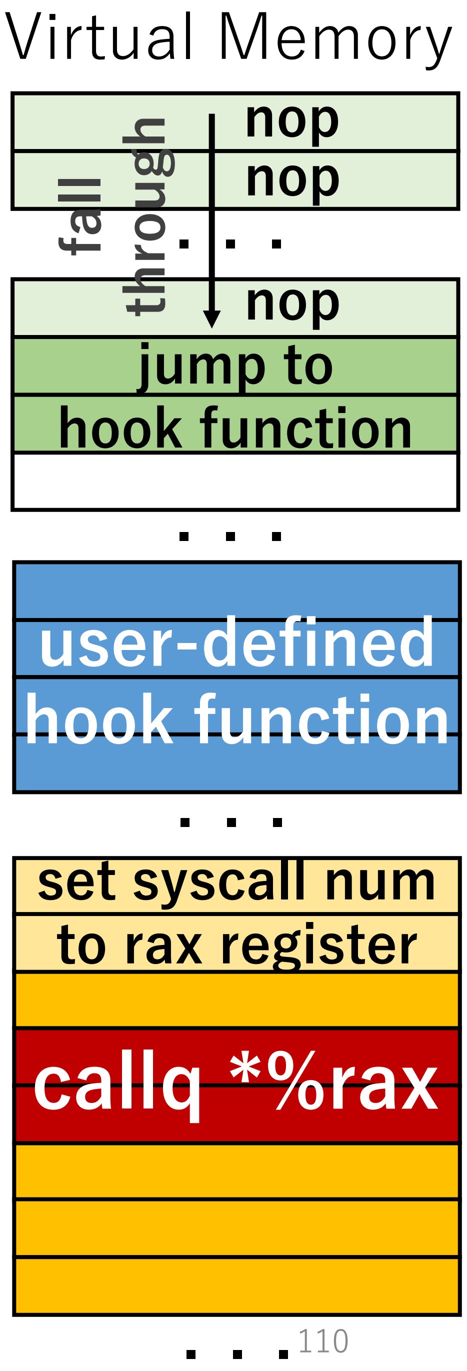
Mechanism	Time [ns]
ptrace	31201
int3 signaling	1342
SUD	1156
zpoline	41
LD_PRELOAD	6

716x

32.7x

28.1x

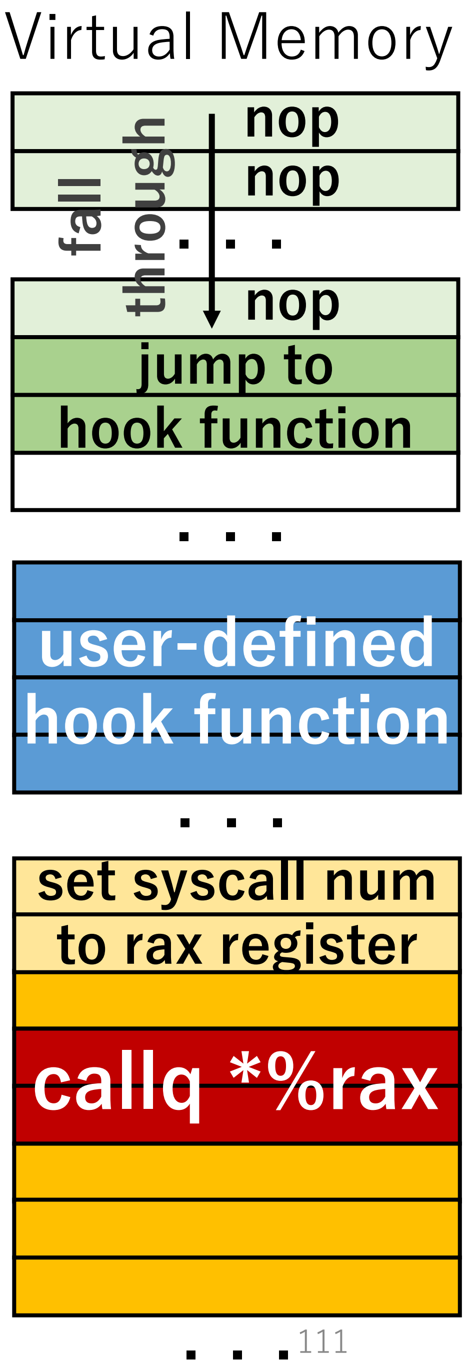
improvement



System Call Hook Overhead

- Time to hook getpid() and return a dummy value

Mechanism	Time [ns]
ptrace	31201
int3 signaling	1342
SUD	1156
zpoline NULL exec check: 1 ns out of	41
LD_PRELOAD	6

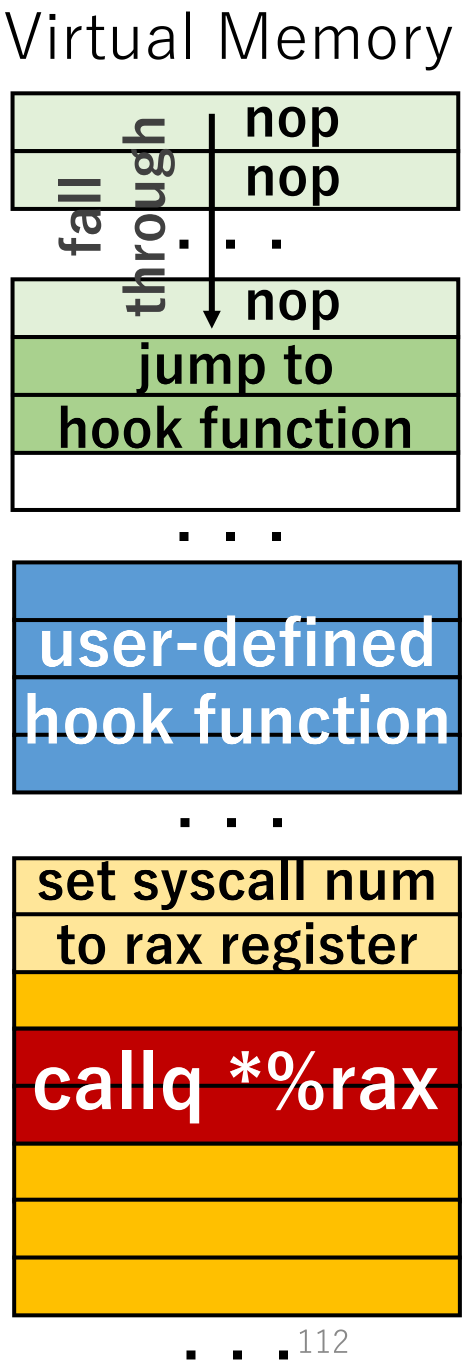


System Call Hook Overhead

- Time to hook getpid() and return a dummy value

Mechanism	Time [ns]
ptrace	31201
int3 signaling	1342
SUD	1156
zpoline	41
LD_PRELOAD	6

+35ns overhead

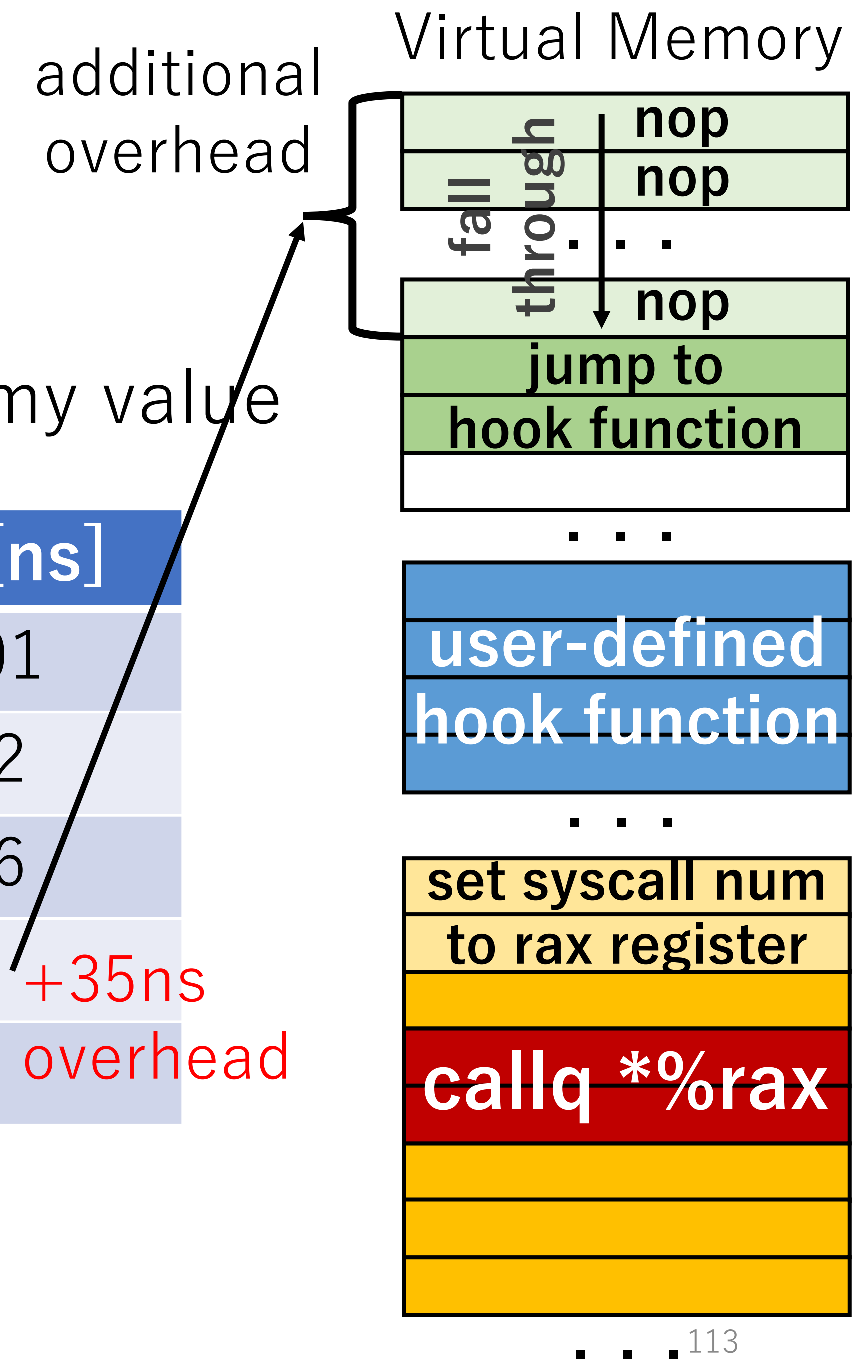


System Call Hook Overhead

- Time to hook getpid() and return a dummy value

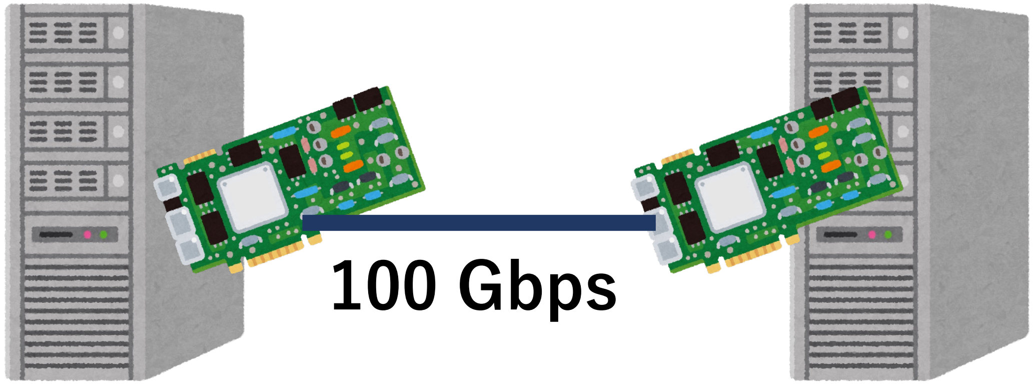
Mechanism	Time [ns]
ptrace	31201
int3 signaling	1342
SUD	1156
zpoline	41
LD_PRELOAD	6

+35ns overhead



Application Performance

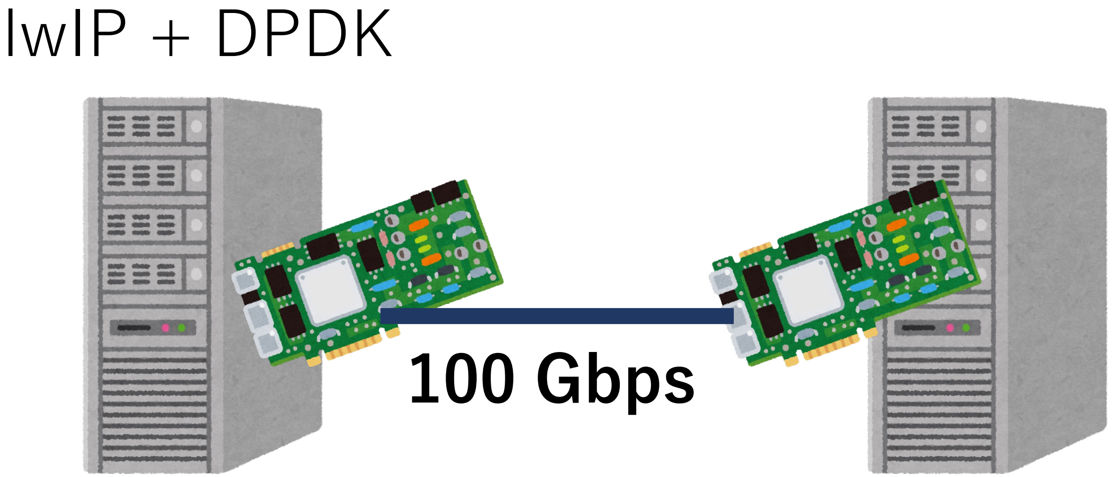
- We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms



Application Performance

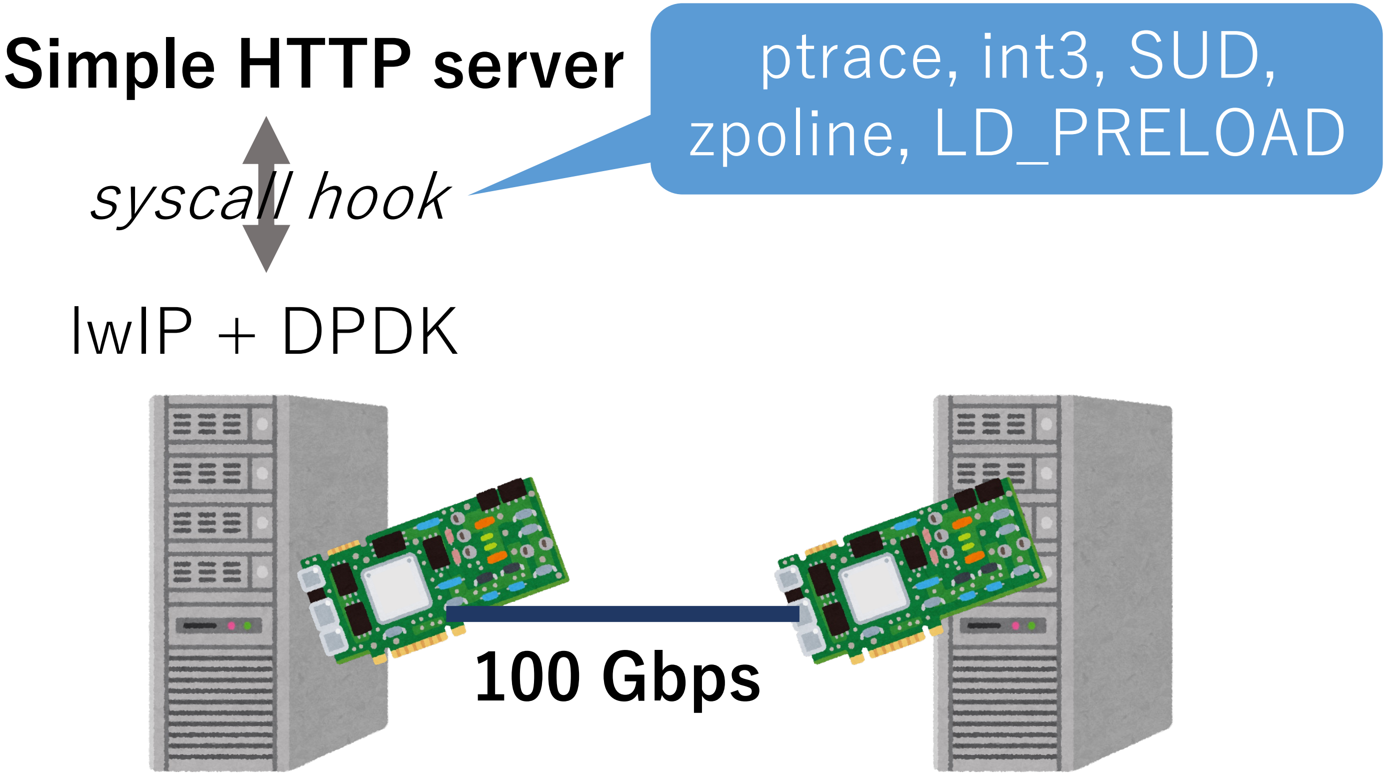
- We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms

Simple HTTP server



Application Performance

- We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms



Simple HTTP server

syscall hook

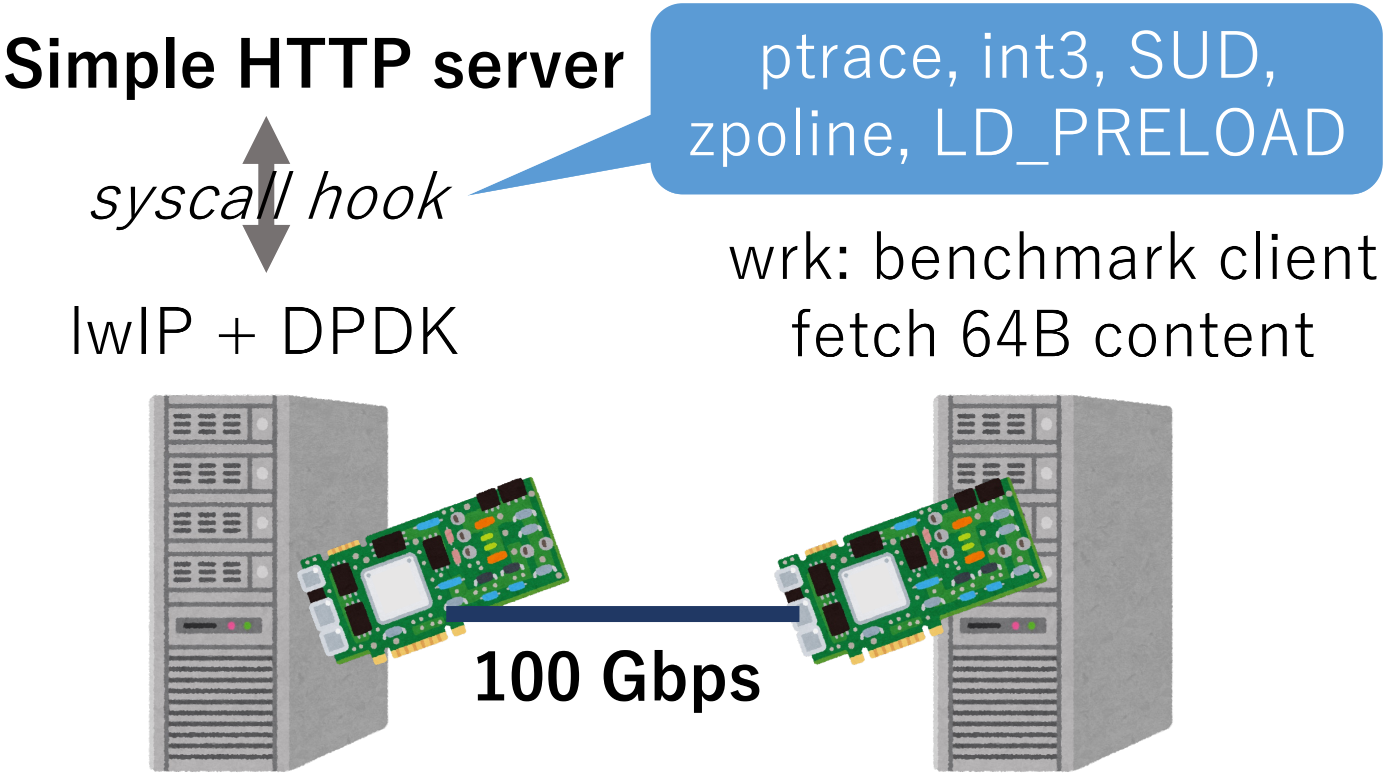
ptrace, int3, SUD, zpoline, LD_PRELOAD

lwIP + DPDK

100 Gbps

Application Performance

- We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms



Simple HTTP server

syscall hook

lwIP + DPDK

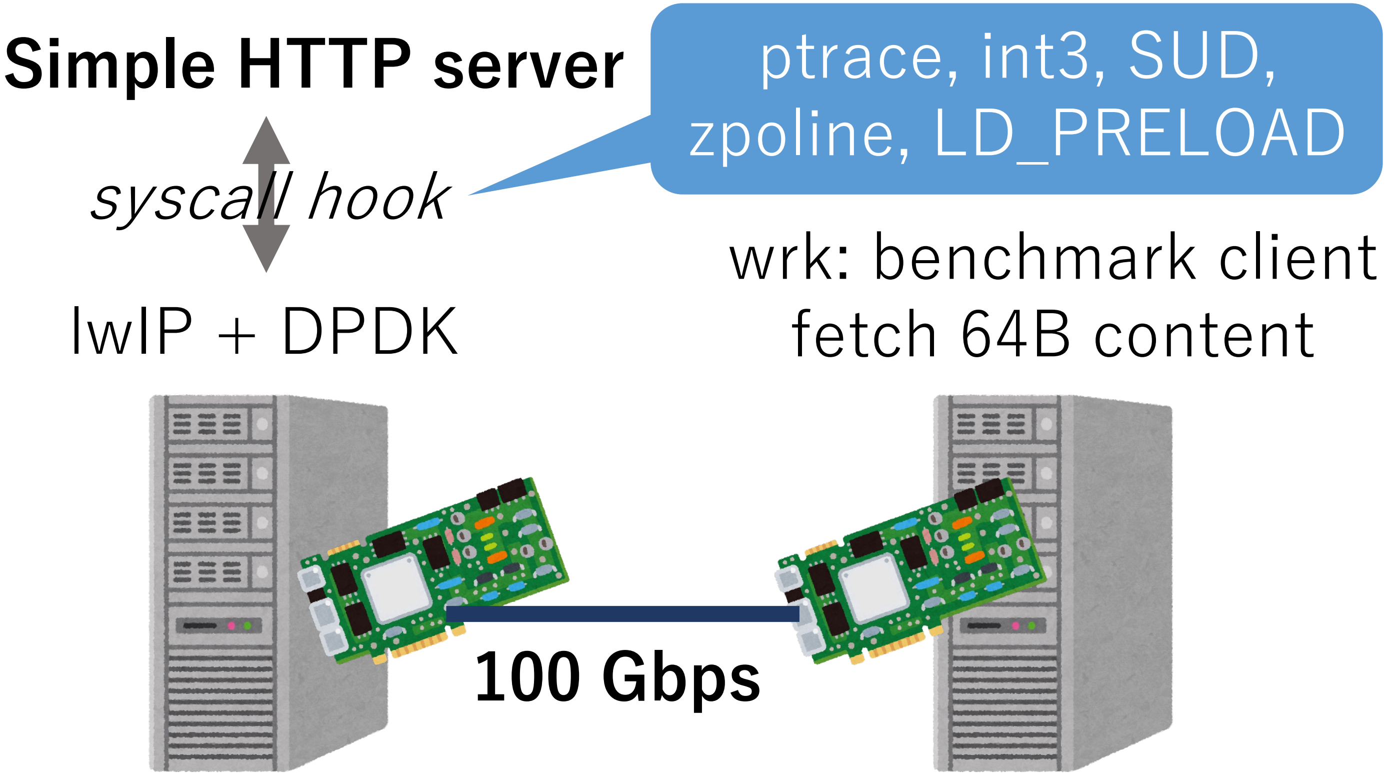
ptrace, int3, SUD, zpoline, LD_PRELOAD

wrk: benchmark client
fetch 64B content

100 Gbps

Application Performance

- We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms



Simple HTTP server

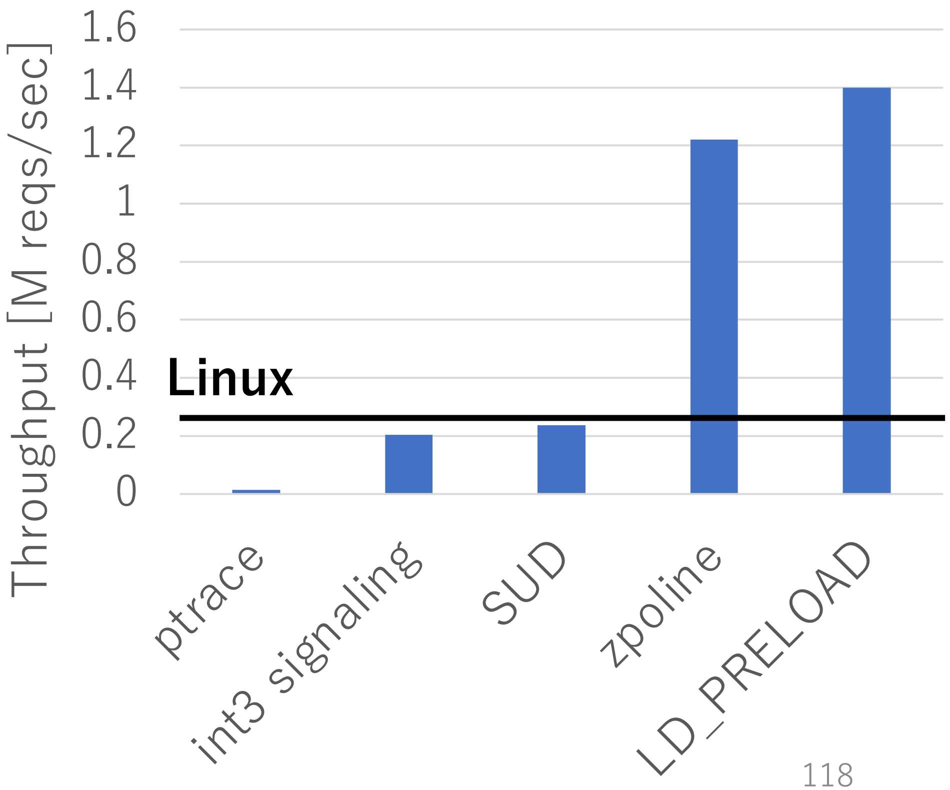
syscall hook

lwIP + DPDK

ptrace, int3, SUD, zpoline, LD_PRELOAD

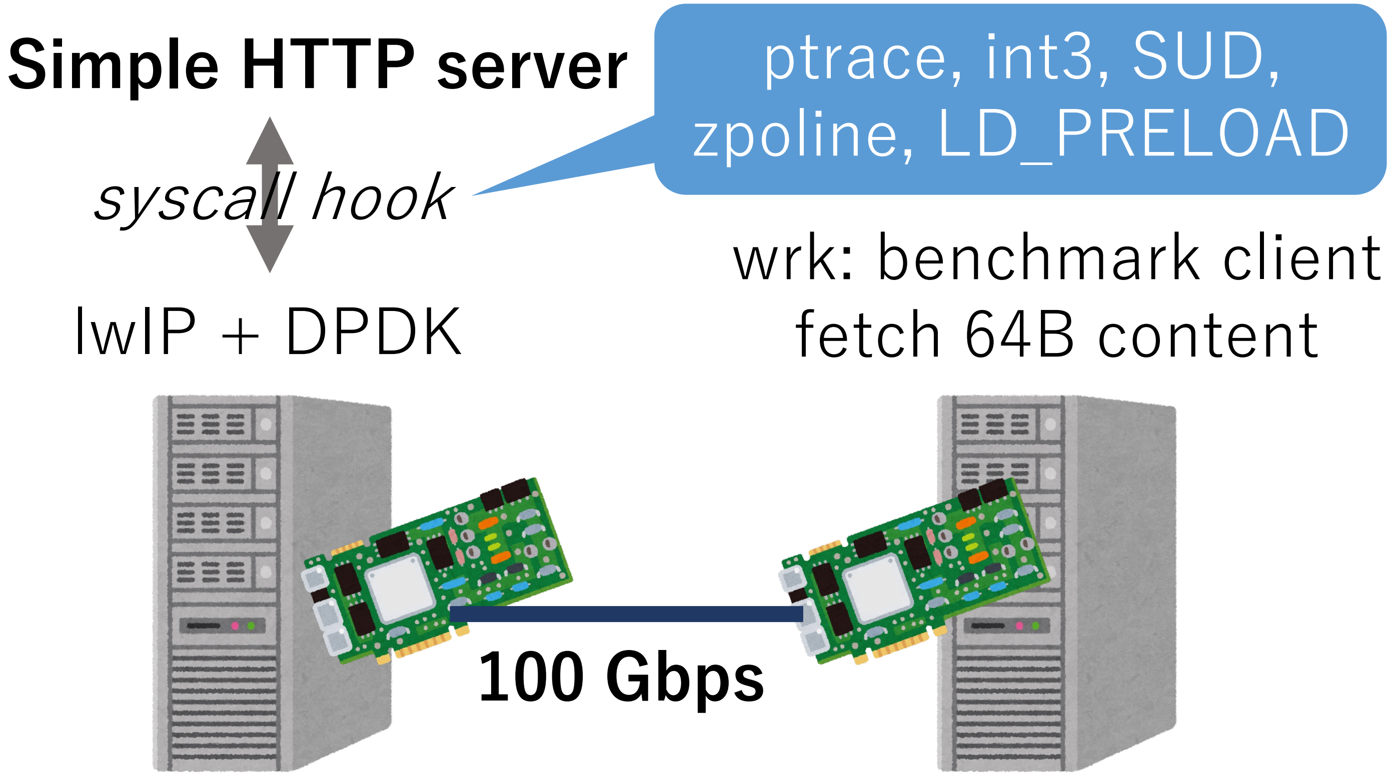
wrk: benchmark client
fetch 64B content

100 Gbps



Application Performance

- We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms



Simple HTTP server

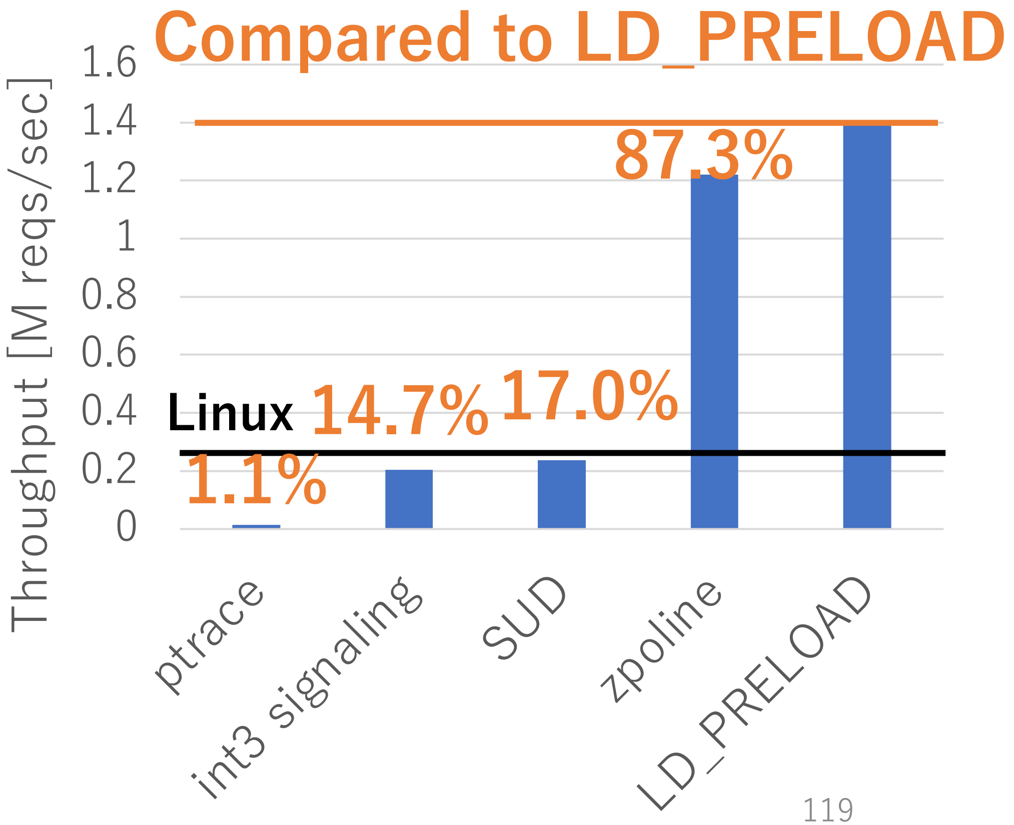
syscall hook

lwIP + DPDK

ptrace, int3, SUD, zpoline, LD_PRELOAD

wrk: benchmark client
fetch 64B content

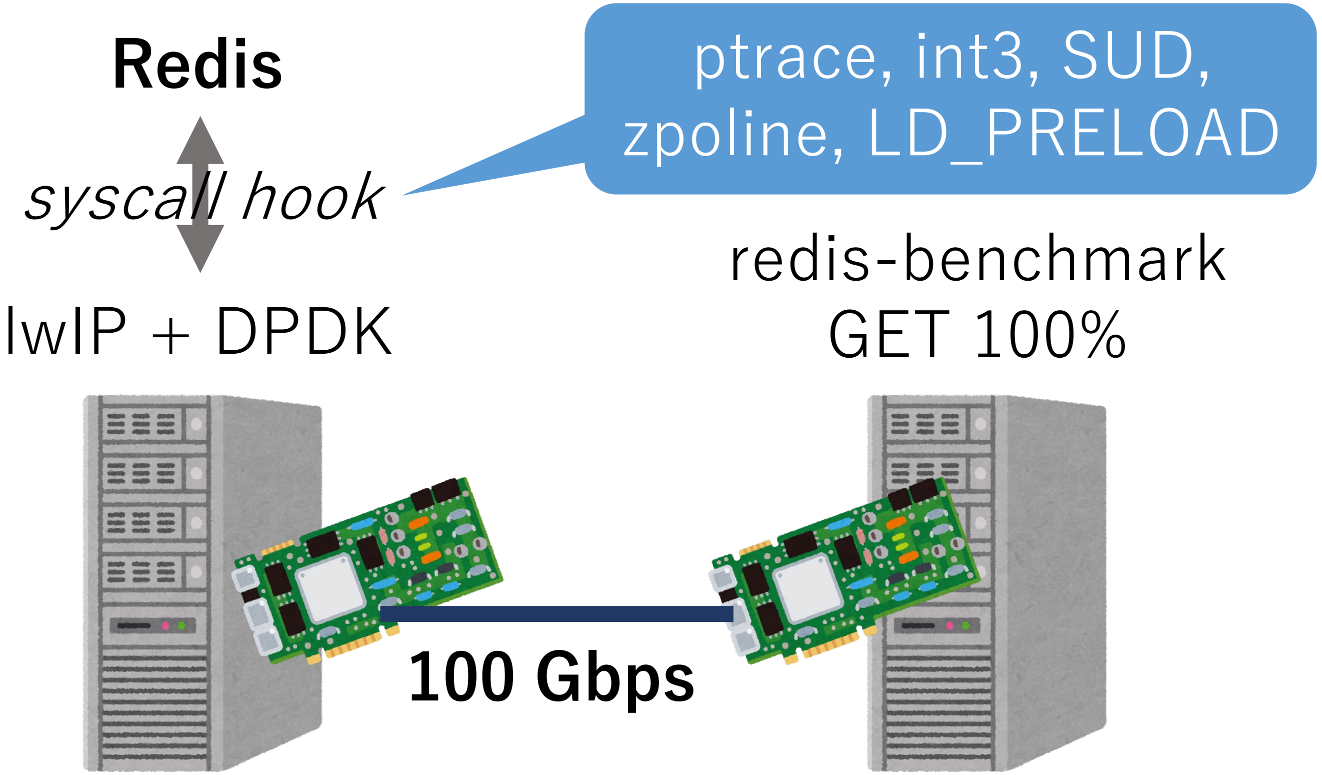
100 Gbps



Compared to LD_PRELOAD

Application Performance

- We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms



Redis

syscall hook

ptrace, int3, SUD, zpoline, LD_PRELOAD

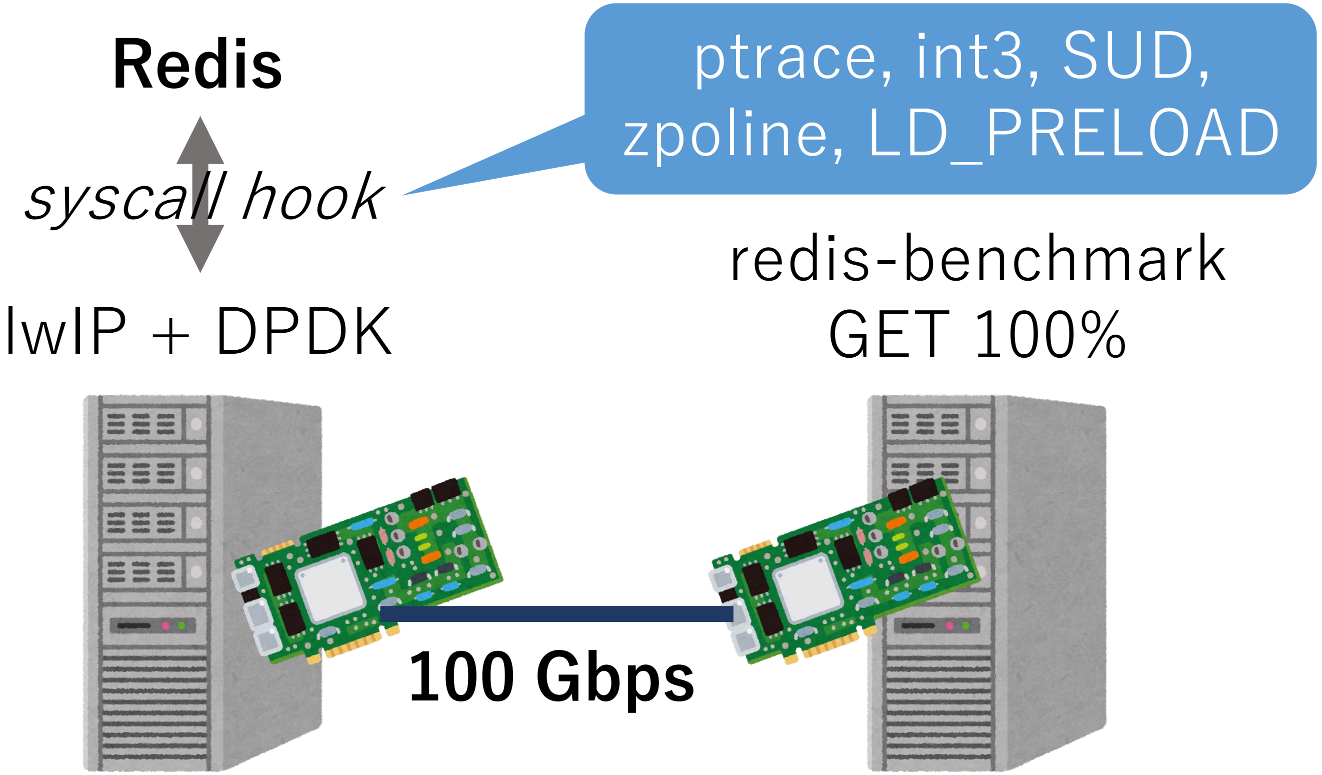
lwIP + DPDK

redis-benchmark GET 100%

100 Gbps

Application Performance

- We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms



Redis

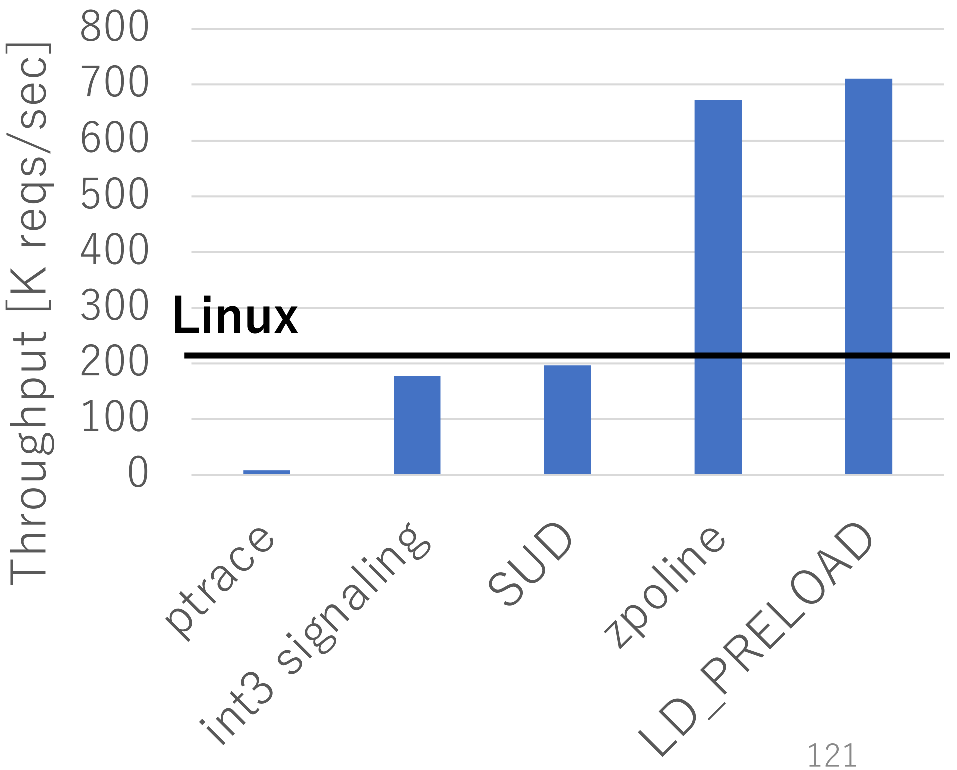
syscall hook

lwIP + DPDK

ptrace, int3, SUD, zpoline, LD_PRELOAD

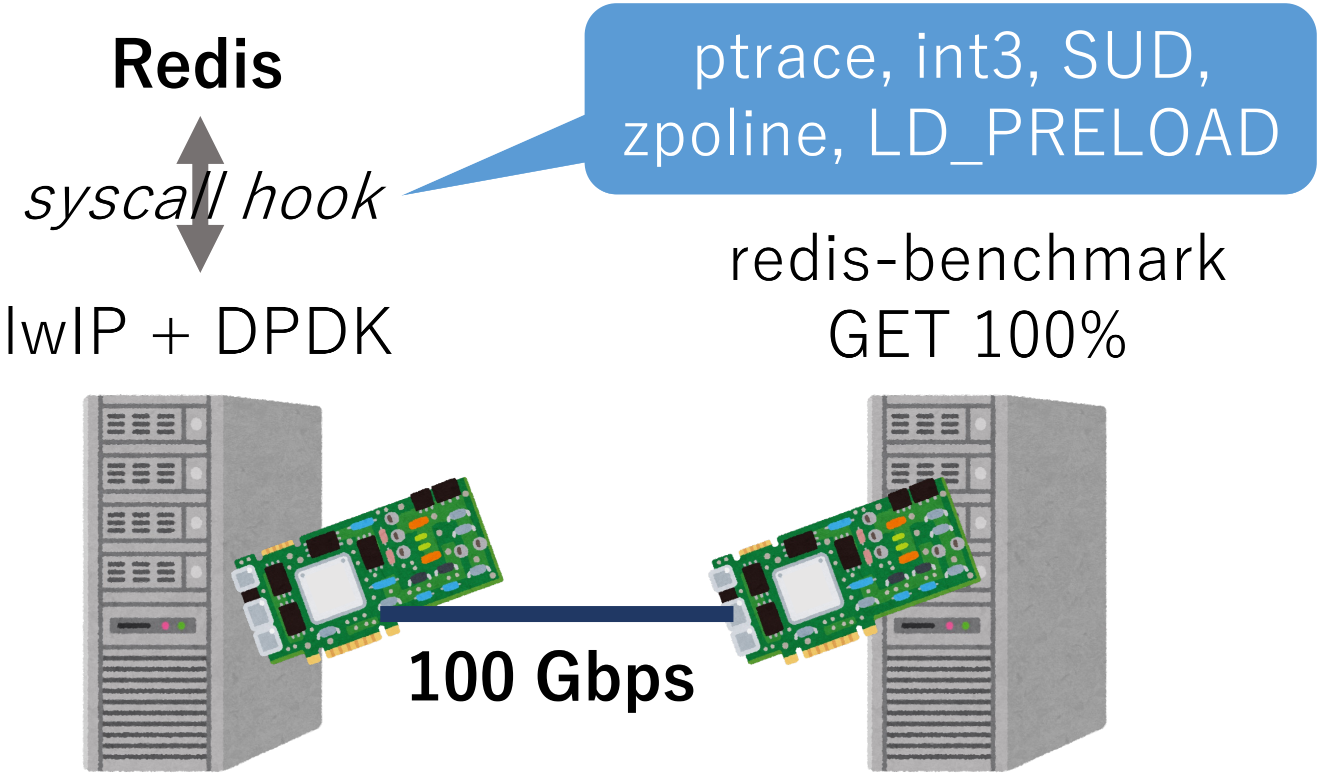
redis-benchmark
GET 100%

100 Gbps



Application Performance

- We **transparently** apply lwIP + DPDK to an application using different system call hook mechanisms



Redis

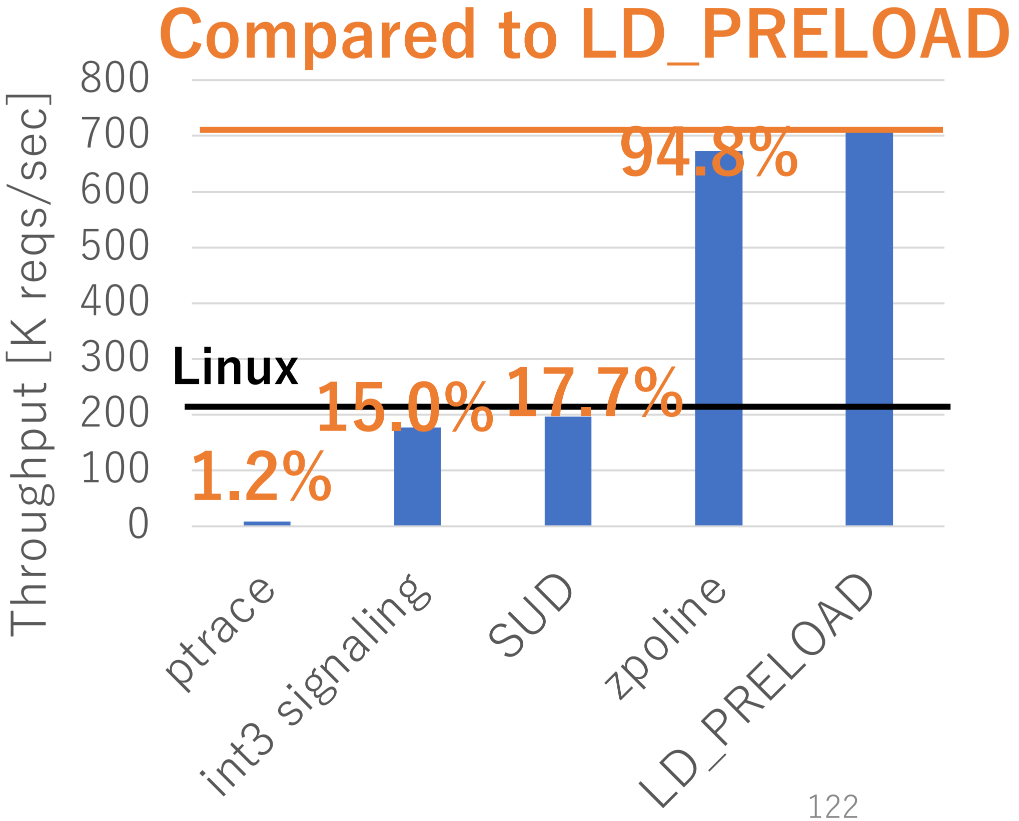
syscall hook

lwIP + DPDK

ptrace, int3, SUD, zpoline, LD_PRELOAD

redis-benchmark
GET 100%

100 Gbps



Compared to LD_PRELOAD

Summary

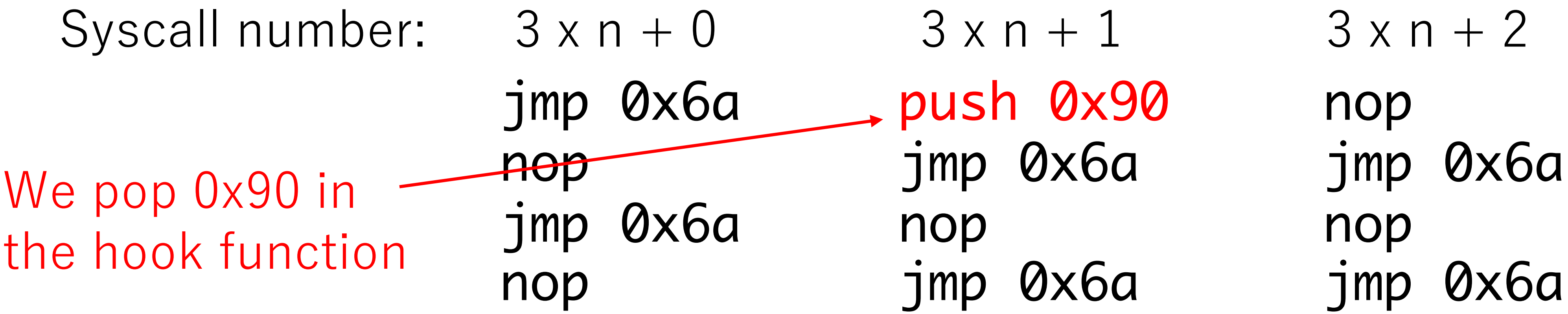
- zpoline: a system call hook mechanism for x86-64 CPUs
    - based on binary rewriting
        - replaces syscall/sysenter with callq *%rax
        - instantiates the trampoline code at virtual address 0 (zero)
    - free from the drawbacks of the pervious mechanisms
    - keeps the performance benefit of user-space OS subsystems

- Source code: https://github.com/yasukata/zpoline
    - since October 2021

Speeding up the Trampoline Code

- Inspired from USENIX ATC’23 reviewers who suggested to employ a one-byte short jump instruction for speeding up
  - Put it on the addresses corresponding to obsolete system calls

- Optimization: repeat **0xeb 0x6a 0x90** instead of nops
  - Hook overhead reduction from 41 ns to 10 ns



Syscall number:	3 x n + 0	3 x n + 1	3 x n + 2
	jmp 0x6a	push 0x90	nop
	nop	jmp 0x6a	jmp 0x6a
	jmp 0x6a	nop	nop
	nop	jmp 0x6a	jmp 0x6a

We pop 0x90 in the hook function