Page-Oriented Programming: Subverting Control-Flow Integrity of Commodity Operating System Kernels with Non-Writable Code Pages

33RD USENIX SECURITY SYMPOSIUM

Seunghun Han[1,2], Seong-Joong Kim[1], Wook Shin[1], Byung Joon Kim[1], Jae-Cheol Ryou[2]

1. The Affiliated Institute of ETRI
2. Chungnam National University

Outline

- **Background**

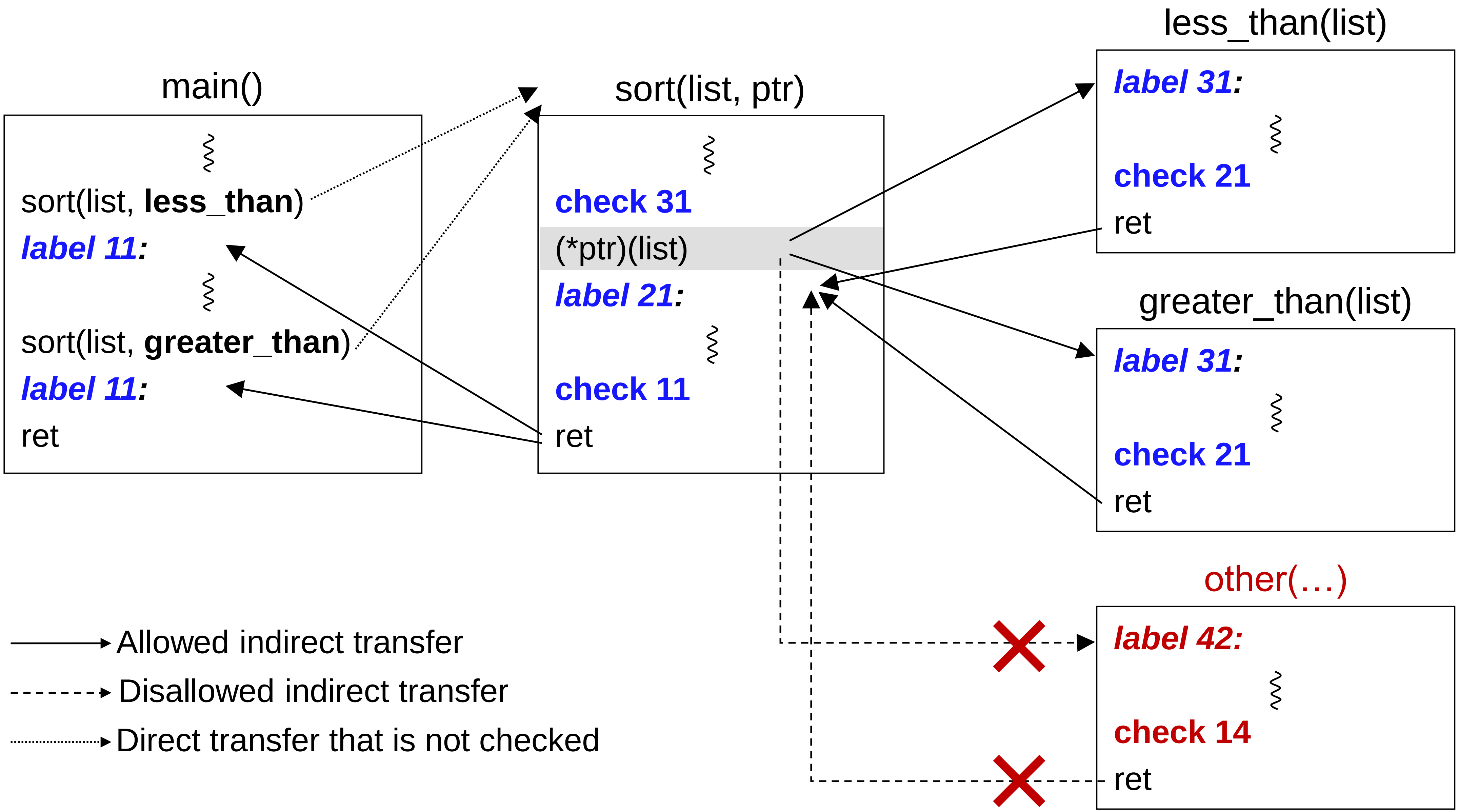
- Threat Model and Motivation

- Page-Oriented Programming (POP)

- Evaluation

- Discussion and Conclusion

Control-Flow Integrity



Practical Implementations for Commodity OSes

- CFI implementations have focused on practicality
  - They integrate with compilation toolchains and generate static CFGs from source code
  - They employ bitmap-based or function type-based verification and create binaries enforced with CFI

- The implementations also adopt hardware-based CFI mechanisms
  - Recent CPUs support CFI-related features, such as Intel Control-flow Enforcement Technology (CET), that restrict indirect branch targets
    - This is known as ***hardware-assisted CFI***

Practical Implementations in Use (for x86 systems)

CFI Implementation	Commodity OS	Forward Edge Policy	Backward Edge Policy
Microsoft Control-Flow Guard (CFG) with CET	Windows	Bitmap-based verification	Hardware-based shadow stack
PaX Reuse Attack Protector (RAP) (open-source version)	Linux	Type-based verification	Type-based verification
GCC CFI (only CET)	Linux	Hardware-based indirect branch tracking	Hardware-based shadow stack
Clang/LLVM CFI with CET	Linux, Windows	Type-based verification with hardware-based indirect branch tracking	Hardware-based shadow stack
FineIBT (integrated with CET)	Linux	Type-based verification with hardware-based indirect branch tracking	Hardware-based shadow stack

Non-Writable Code for CFI

- The original work [1] emphasizes the importance of non-writable code (NWC) for their CFI mechanism
  - If an attacker modifies the CFI enforcement code, the mechanism can be neutralized

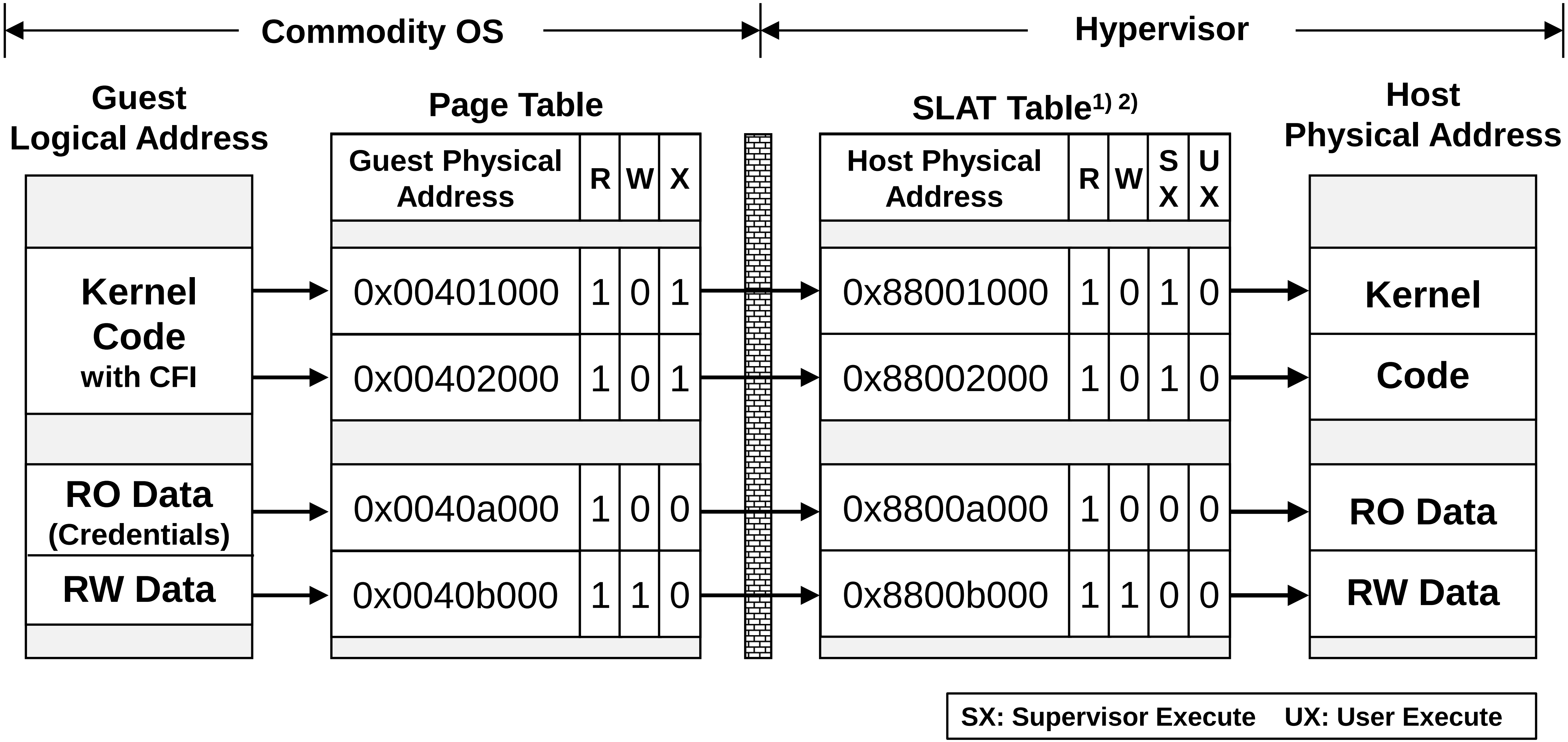
- In commodity OSes, NWC is ensured by the address translation mechanisms of the CPU
  - **Page tables** *in the kernel* for user-level applications
  - **Second-level address translation (SLAT) tables** *in the hypervisor* for the commodity kernels

[1]: Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS), pages 340–353, 2005.

Outline

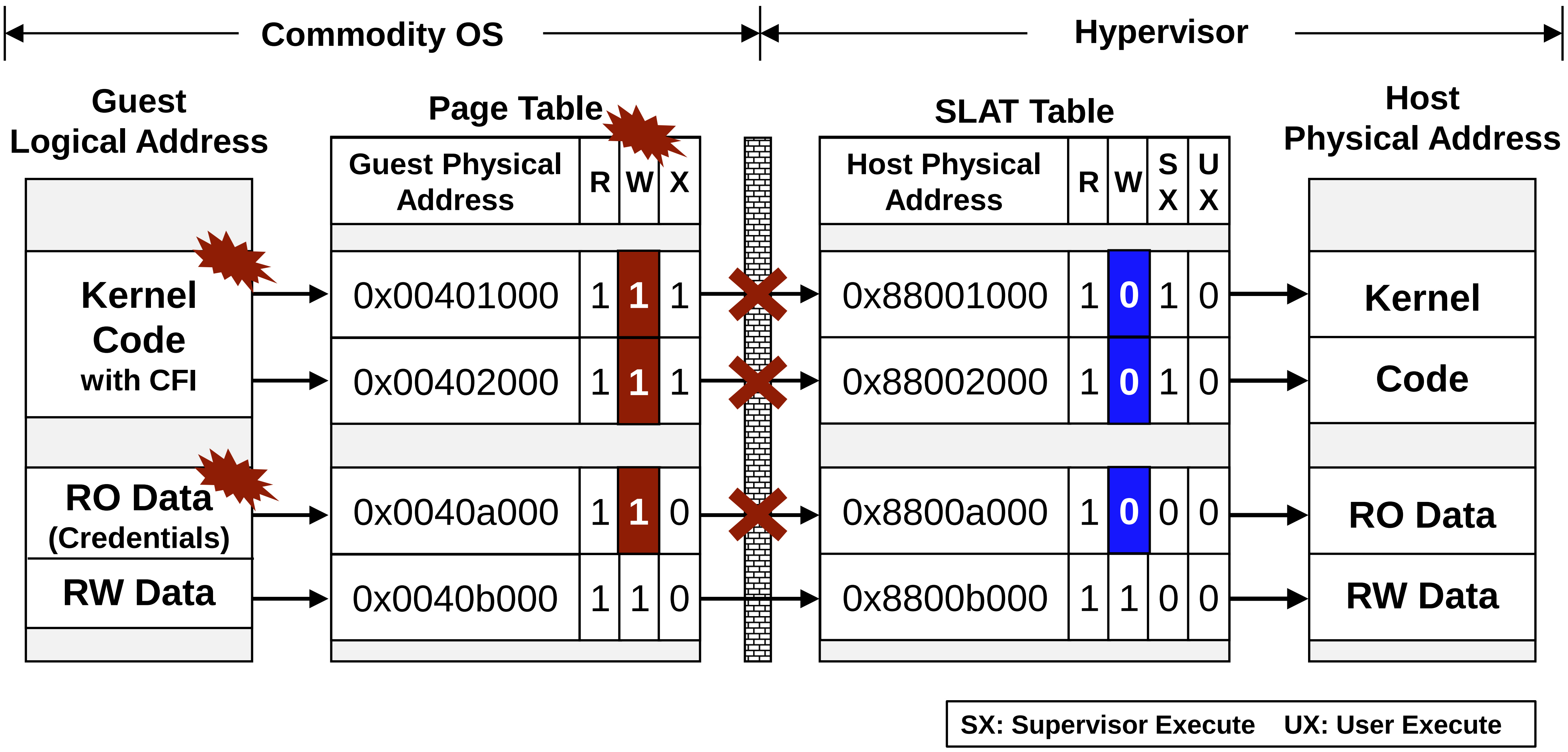
- Background
- **Threat Model and Motivation**
- Page-Oriented Programming (POP)
- Evaluation
- Discussion and Conclusion

Page-Level NWC for Commodity OSes (1)



1) Intel Extended Page Table (EPT) and AMD Rapid Virtualization Indexing (RVI) support the SLAT feature
2) Intel Mode-Based Execution Control (MBEC) and AMD Guest Mode Execution Trap (GMET) support user and supervisor mode-based executions

Page-Level NWC for Commodity OSes (2)



Threat Model and Assumption

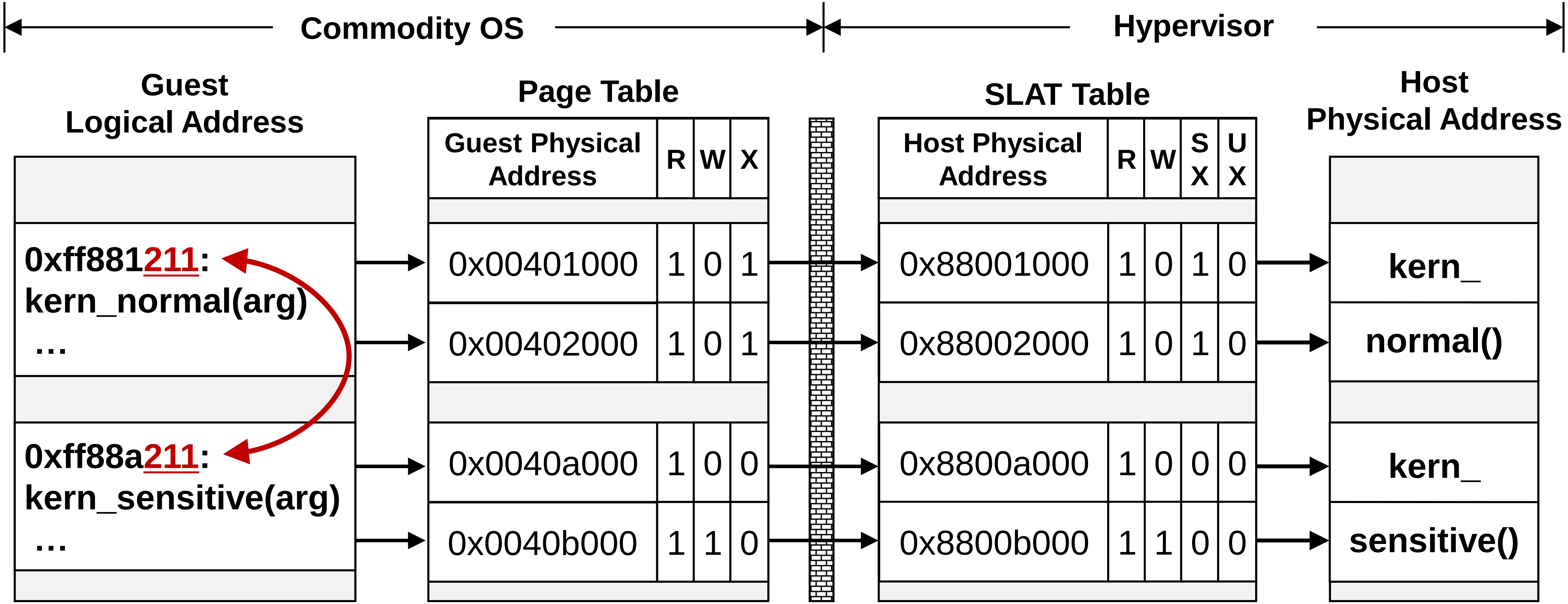
- We assume the target system is fortified with hardware-assisted CFI policies and the page-level NWC mechanism
  - Therefore, the system can thwart typical attack techniques such as unauthorized code alterations, code injections, control-flow hijackings, and direct modifications to kernel credentials

- We assume attackers have an arbitrary kernel memory read and write vulnerability
  - By exploiting it, attackers can leak information from the kernel, manipulate page tables, and bypass kernel ASLR
  - They also have local user privileges and can execute arbitrary programs to exploit it

Motivation

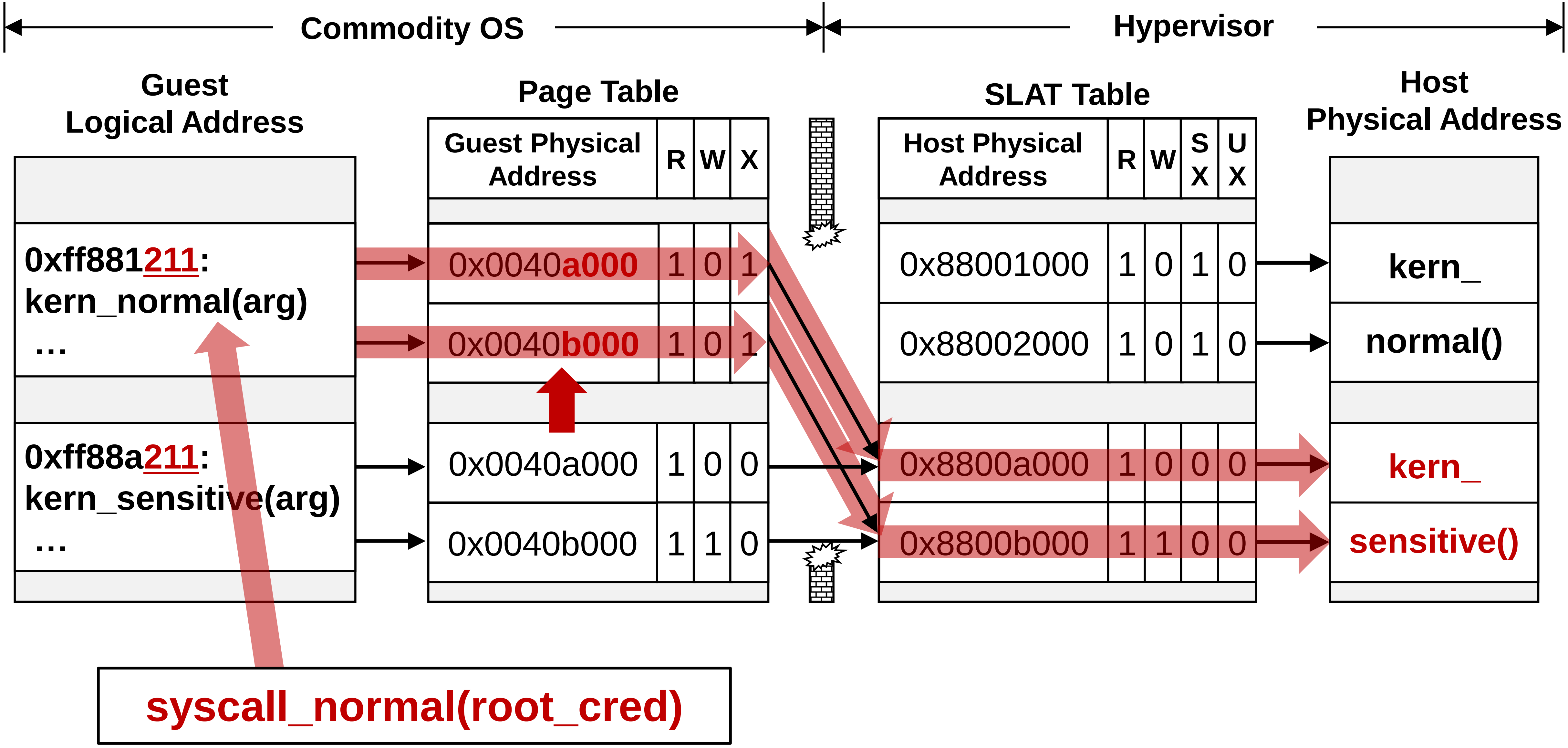
- i) Is page-level protection sufficient to ensure NWC?
  - SLAT tables in the hypervisor only translate guest physical addresses (GPAs) to host physical addresses (HPAs)
  - **The tables do not consider guest logical address (GLA) to GPA mappings**
- ii) Is indirect branch tracking sufficient to detect control-flow deviations?
  - Practical CFI implementations do not monitor direct branches because their target addresses are fixed in the code
  - **However, they are fixed in the GLA space, not the GPA space**

Blind Spots of Page-Level NWC (1)



The page offsets of kern_normal() and kern_sensitive() are identical

Blind Spots of Page-Level NWC (2)



Outline

- Background

- Threat Model and Motivation

- **Page-Oriented Programming (POP)**

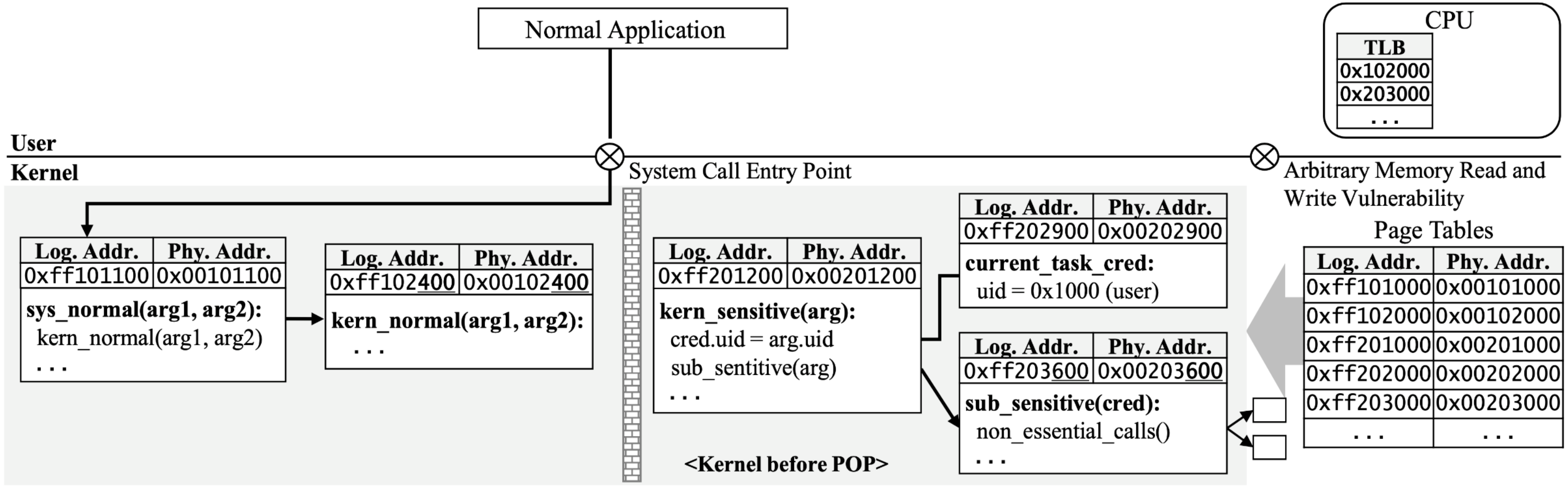
- Evaluation

- Discussion and Conclusion

Page-Oriented Programming (POP)

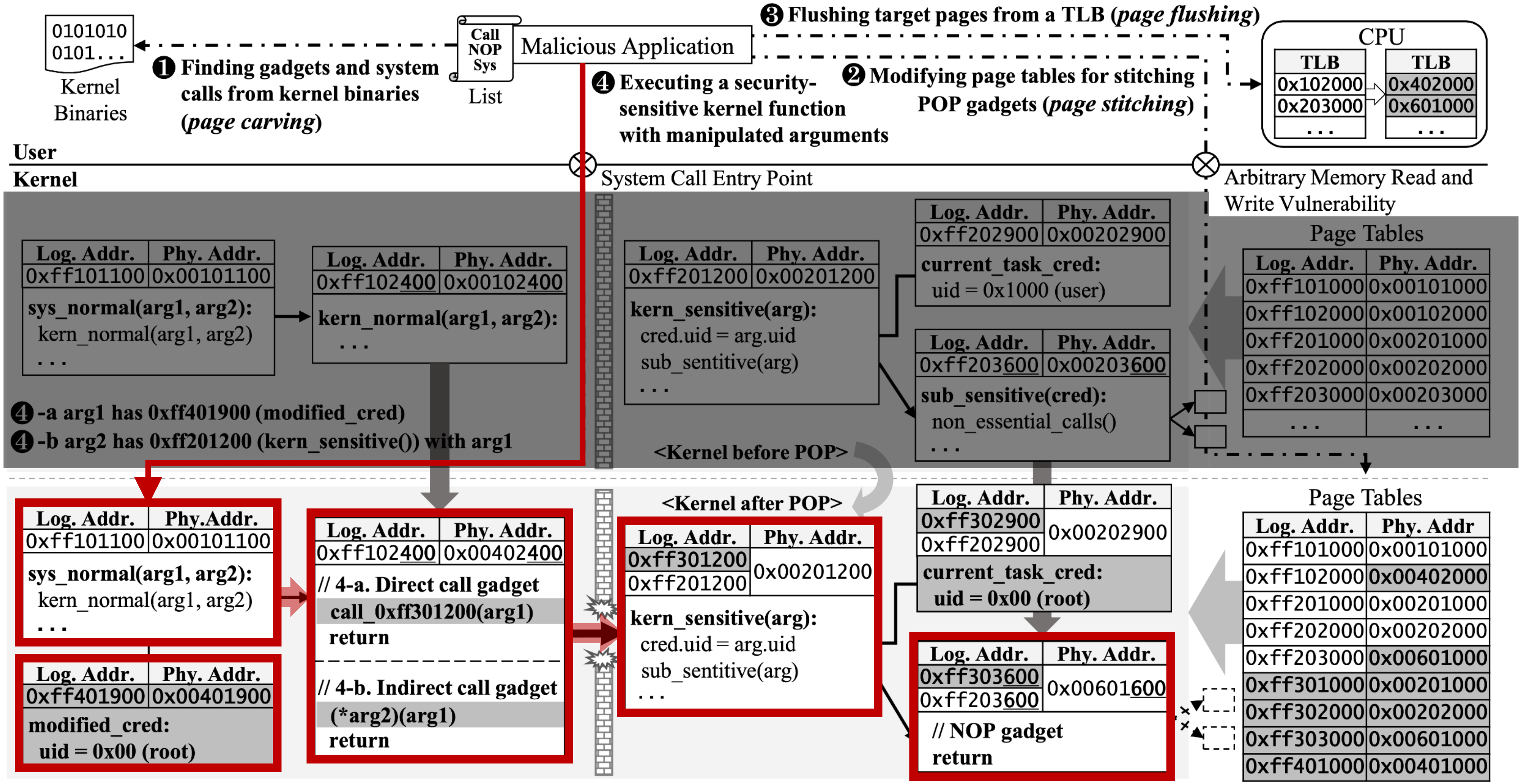
- POP is a novel page-level code reuse attack, similar to ROP and JOP
  - It revisits **page remapping attacks** and exploits the weaknesses in state-of-the-art kernel CFI implementations
  - It programs **page tables** within the kernel using a **kernel memory read and write vulnerability**
- POP can create arbitrary control flows under CFI enforcement
  - It identifies **page-level gadgets** and stitches them for **attacker-controlled execution flows**
  - Page-level NWC and hardware-assisted CFI policies are bypassed

Attack Scenario of POP (1)



: POP flow
: Original control flow
: New control flow
: Original kernel code and data
: POP gadget
: Referenced data
: Replaced function due to identical page offset
: Modified area due to POP
: Unlinked function

Attack Scenario of POP (2)



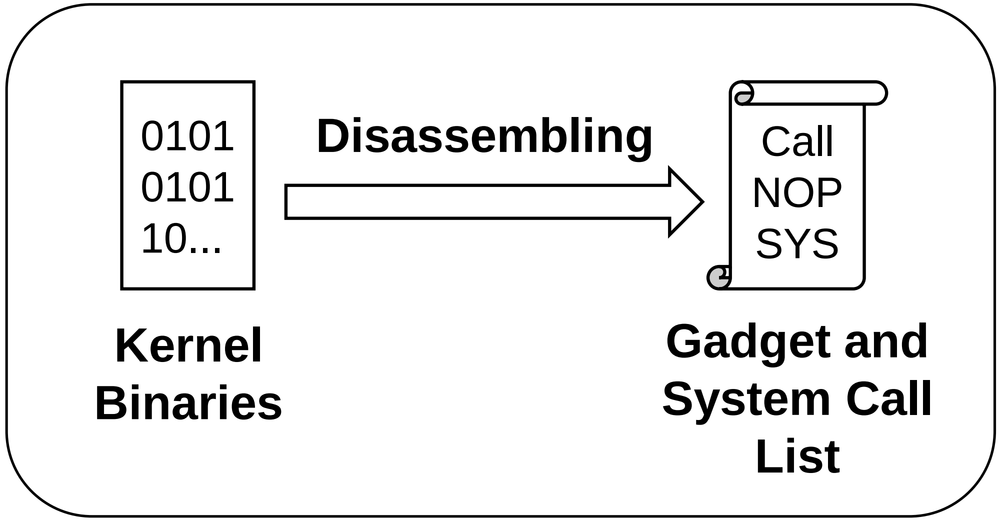
Log. Addr.	Phy. Addr.
0xff101000	0x00101000
0xff102000	0x00102000
0xff201000	0x00201000
0xff202000	0x00202000
0xff203000	0x00203000
...	...

Page Tables

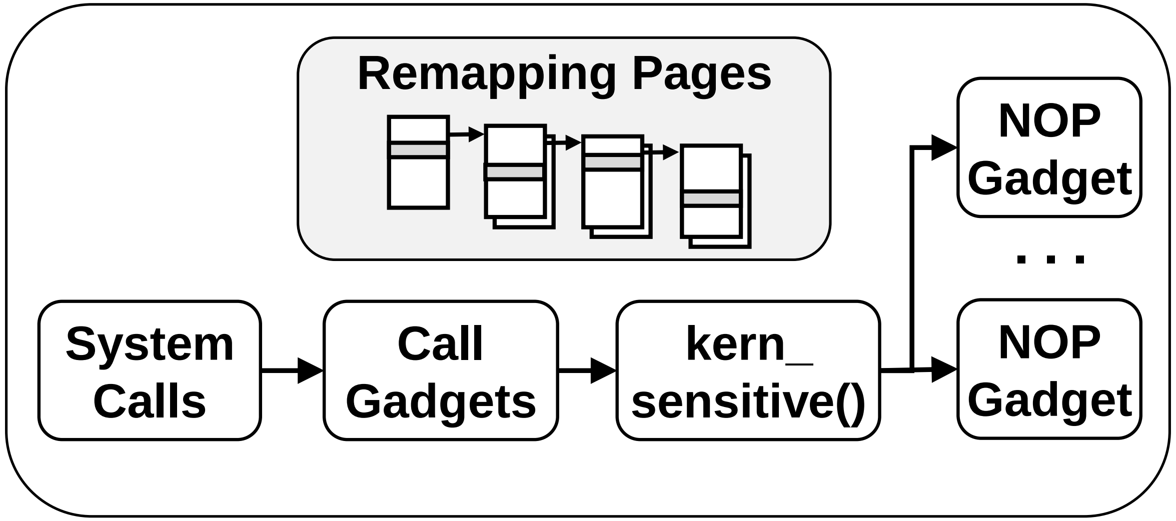
Log. Addr.	Phy. Addr
0xff101000	0x00101000
0xff102000	0x00402000
0xff201000	0x00201000
0xff202000	0x00202000
0xff203000	0x00601000
0xff301000	0x00201000
0xff302000	0x00202000
0xff303000	0x00601000
0xff401000	0x00401000

- - → : POP flow — → : Original control flow — ➡ : New control flow — □ : Original kernel code and data — ▣ : POP gadget
— : Referenced data — ➡ : Replaced function due to identical page offset — ▨ : Modified area due to POP — ⬚ : Unlinked function

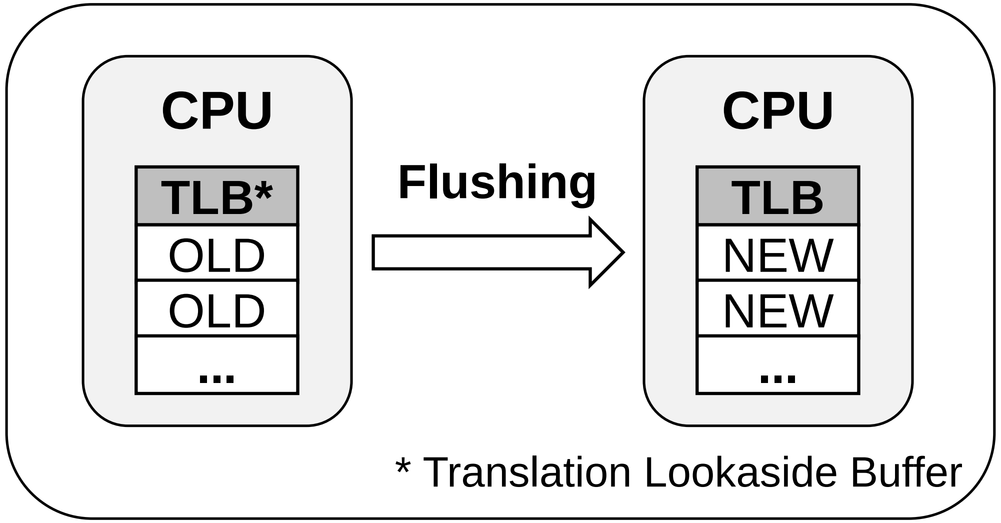
Stage and Challenge of POP



(1) Page carving



(2) Page stitching

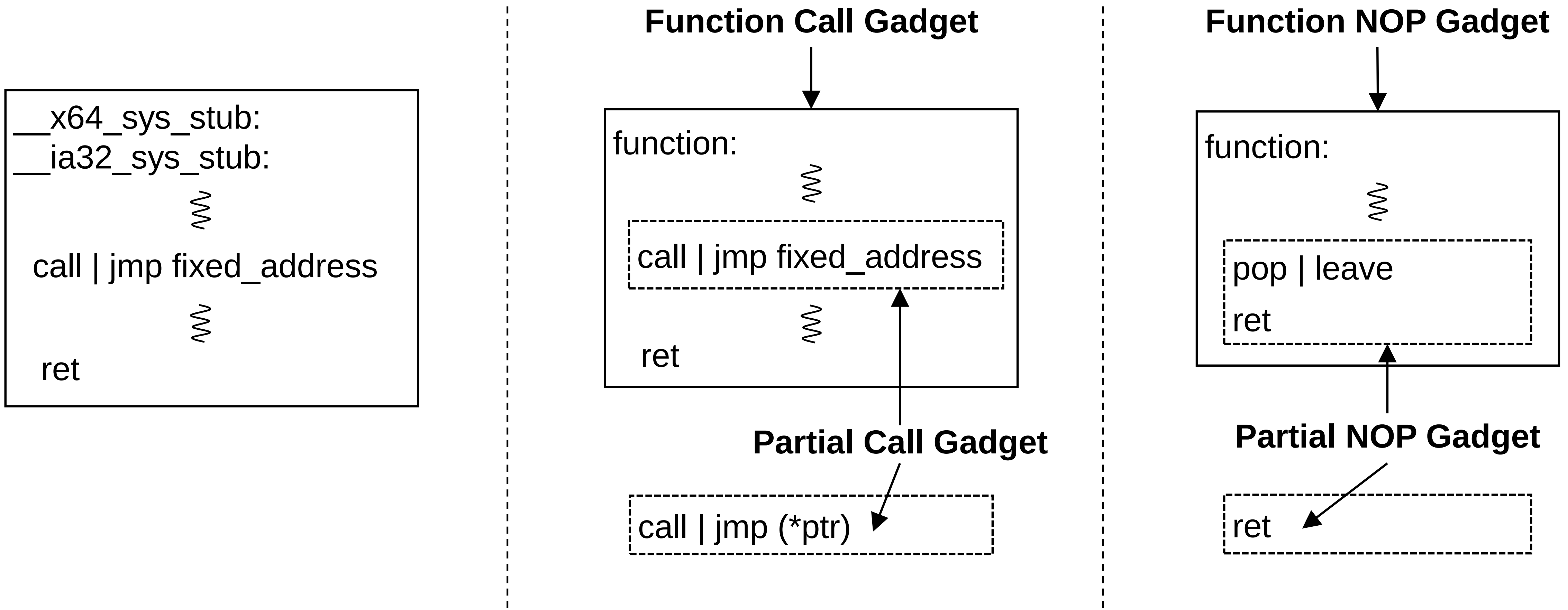


(3) Page flushing

```
; Syscall number for exploitation
mov $syscall_number, %rax

; Argument for the kern_sensitive() function
mov $0xdeadbeaf, %rdi or %rbx

; Execute the new control flow
syscall or int $0x80

; Malicious behaviors with root privileges …
```

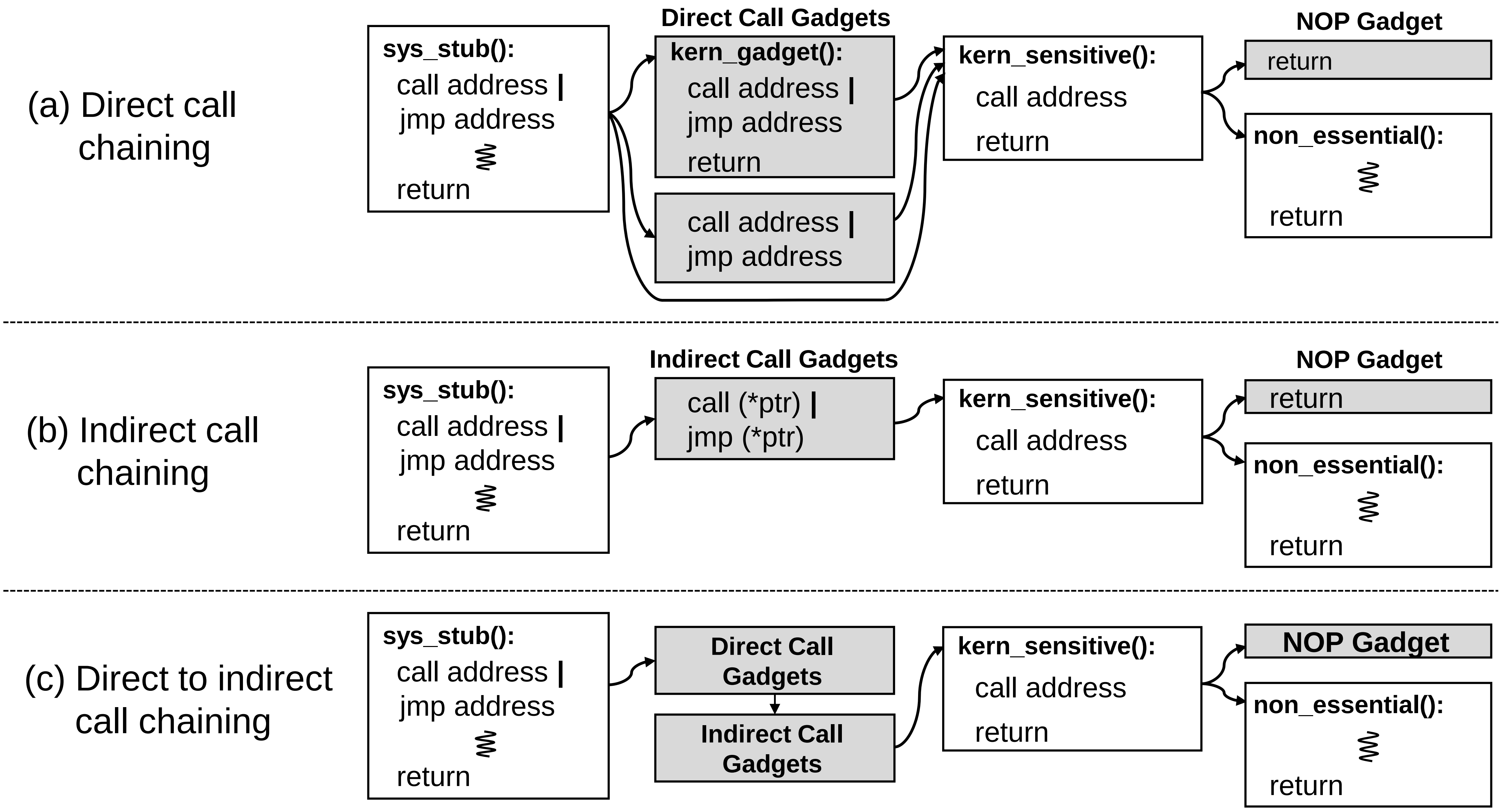
Exploitation

Stage 1 - Page Carving



(a) System call candidate　(b) Call gadget　(c) NOP gadget

Stage 2 - Page Stitching



Stage 3 - Page Flushing

- **Page flushing wipes out stale mappings** in the TLB to replace them with new ones
  - Modern CPUs manage TLB data to accelerate the translation from logical to physical addresses
  - Remapped physical pages are not accessed until the old mappings in the TLB are flushed

- This stage removes **global bits from page tables** and waits for a sufficient time
  - Non-global pages have the same priority as user-level pages
  - The TLB has limited space, so non-global pages are flushed more frequently than kernel pages

Outline

- Background

- Threat Model and Motivation

- Page-Oriented Programming (POP)

- **Evaluation**

- Discussion and Conclusion

Environment

- Machine: HP Victus 16 laptop
  - Intel i7-12700H with the CET technology and 16 GB RAM

- Operating system, compilation toolchain, and kernel CFI implementations
  - ***Ubuntu 22.04.2*** and ***LLVM 6.0.0***
  - Clang/LLVM kernel CFI with ***Linux kernel 6.1.12***
  - FineIBT with ***Linux kernel 6.2.8***

- Hypervisor-based page-level protection
  - Open-source hypervisor, ***Shadow-box***, with CET and MBEC extensions from Intel

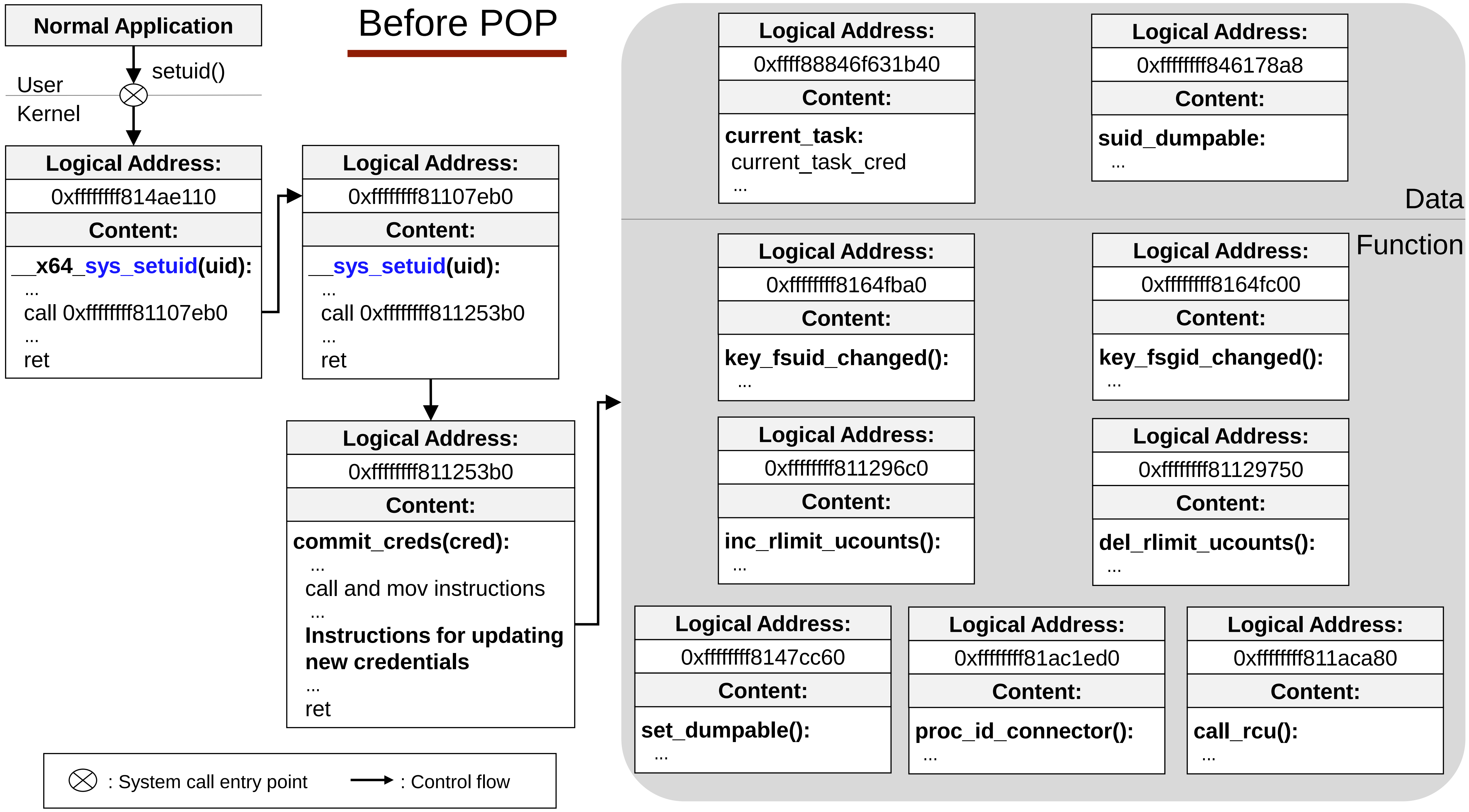
Evaluation

- i) Proof-of-concept (PoC) exploitation
  - We developed PoC exploit code for FineIBT using a real-world vulnerability
    - *CVE-2013-2595*: Page remapping capability
- ii) Analysis of branch and gadget distributions
  - We analyzed the distributions of system call candidates, direct branches, and indirect branches

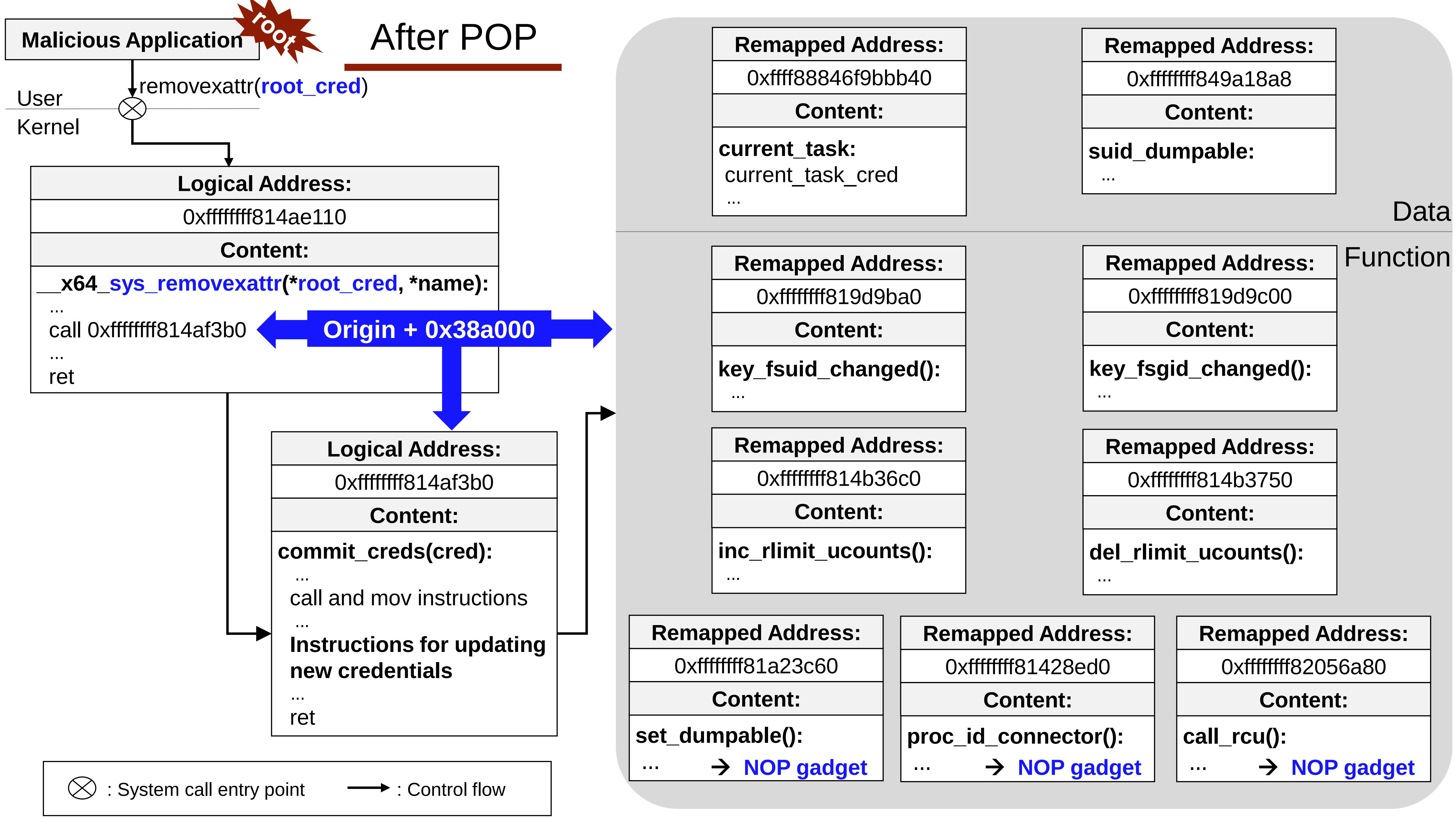
PoC - Essential Symbols for Exploitation

Symbol Name	Offset in Kernel Code	Usage
sys_call_table	0x1400400	**Breaking Kernel Address Space Layout Randomization (KASLR)**
__x64_sys_read	0x46fda0	
clear_tasks_mm_cpumask()	0xeb800	**Identifying kernel data structures such as task_struct, mm_struct, and cred**
prepare_kernel_cred()	0x1257f0	
__set_task_comm()	0x47bff0	
pgd_alloc()	0xc6840	
init_task	0x201bb00	**Performing POP**
page_offset_base	0x19d7008	
__per_cpu_offset	0x19dd9e0	
commit_creds()	0x1253b0	

Before POP


Normal Application
setuid()
User
Kernel
Logical Address:
0xffffffff814ae110
Content:
__x64_sys_setuid(uid):
...
call 0xffffffff81107eb0
...
ret
Logical Address:
0xffffffff81107eb0
Content:
__sys_setuid(uid):
...
call 0xffffffff811253b0
...
ret
Logical Address:
0xffffffff811253b0
Content:
commit_creds(cred):
...
call and mov instructions
...
Instructions for updating new credentials
...
ret
Logical Address:
0xffff88846f631b40
Content:
current_task:
current_task_cred
...
Logical Address:
0xffffffff846178a8
Content:
suid_dumpable:
...
Data
Function
Logical Address:
0xffffffff8164fba0
Content:
key_fsuid_changed():
...
Logical Address:
0xffffffff8164fc00
Content:
key_fsgid_changed():
...
Logical Address:
0xffffffff811296c0
Content:
inc_rlimit_ucounts():
...
Logical Address:
0xffffffff81129750
Content:
del_rlimit_ucounts():
...
Logical Address:
0xffffffff8147cc60
Content:
set_dumpable():
...
Logical Address:
0xffffffff81ac1ed0
Content:
proc_id_connector():
...
Logical Address:
0xffffffff811aca80
Content:
call_rcu():
...
: System call entry point
: Control flow

After POP


Malicious Application
root
removexattr(root_cred)
User
Kernel
Logical Address:
0xffffffff814ae110
Content:
__x64_sys_removexattr(*root_cred, *name):
...
call 0xffffffff814af3b0
...
ret
Origin + 0x38a000
Logical Address:
0xffffffff814af3b0
Content:
commit_creds(cred):
...
call and mov instructions
...
Instructions for updating new credentials
...
ret
: System call entry point
: Control flow
Remapped Address:
0xffff88846f9bbb40
Content:
current_task:
current_task_cred
...
Remapped Address:
0xffffffff849a18a8
Content:
suid_dumpable:
...
Data
Function
Remapped Address:
0xffffffff819d9ba0
Content:
key_fsuid_changed():
...
Remapped Address:
0xffffffff819d9c00
Content:
key_fsgid_changed():
...
Remapped Address:
0xffffffff814b36c0
Content:
inc_rlimit_ucounts():
...
Remapped Address:
0xffffffff814b3750
Content:
del_rlimit_ucounts():
...
Remapped Address:
0xffffffff81a23c60
Content:
set_dumpable():
... → NOP gadget
Remapped Address:
0xffffffff81428ed0
Content:
proc_id_connector():
... → NOP gadget
Remapped Address:
0xffffffff82056a80
Content:
call_rcu():
... → NOP gadget

Distributions - System Call Candidates

Kernel Version	Configuration	System Call	
		Total (x32 and x64)	Candidate
6.1.12 (Clang/LLVM CFI with CET)	Commodity	992	252[1)]
	Kernel Default	992	220
6.2.8 (FineIBT)	Commodity	992	257
	Kernel Default	992	229

1) Branch targets of system call candidates were aligned by 16 bytes

Distributions - Function Call and NOP Gadgets

Kernel Version	Config.	Code Size[1] (KB)	Function Gadgets		
			Direct Call		NOP
			Call	Jump	
6.1.12 (Clang/LLVM CFI with CET)	Commodity	18,440.6	6,447 **(6,466)**[2]	-	6,088 **(6,126)**
	Kernel Default	18,444.8	5,495 **(5,503)**	2 **(2)**	6,542 **(6,571)**
6.2.8 (FineIBT)	Commodity	20,480.0	6,500 **(6,507)**	-	6,230 **(6,247)**
	Kernel Default	18,432.0	5,504 **(5,506)**	2 **(2)**	6,604 **(6,625)**

1) Code size indicates the .text section size of the kernel binary
2) The numbers at the top of function and partial gadgets represent the number of aligned gadgets. The bold numbers in parentheses represent the sum of 16-bytes aligned and unaligned gadgets.

Distributions - Partial Call and NOP Gadgets

Kernel Version	Config.	Partial Gadgets				
		Direct Call		Indirect Call		NOP
		Call	Jump	Call	Jump	
6.1.12 (Clang/LLVM CFI with CET)	Commodity	67,356 [1)] **(1,073,721)**	4,428 **(68,080)**	60 **(1,313)**	570 **(6,759)**	63,301 **(1,030,371)**
	Kernel Default	61,639 **(1,005,609)**	7,249 **(107,949)**	42 **(759)**	708 **(8,500)**	43,333 **(680,008)**
6.2.8 (FineIBT)	Commodity	69,448 **(1,100,737)**	4,897 **(75,282)**	80 **(1,640)**	604 **(7,095)**	64,498 **(1,045,240)**
	Kernel Default	61,977 **(1,011,125)**	6,799 **(99,514)**	44 **(825)**	733 **(8,816)**	42,125 **(659,266)**

1) Aligned direct call gadgets have unaligned branch targets

Outline

- Background

- Threat Model and Motivation

- Page-Oriented Programming (POP)

- Evaluation

- **Discussion and Conclusion**

Mitigations

- Page table protection and randomization
  - SecVisor, HyperSafe, and kCoFI introduce page table protection techniques with escorting page updates
  - PT-Rand and Microsoft Windows employ page table randomization techniques to conceal page table information
- Compartmentalization and domain isolation
  - SeCage and xMP can impede POP by isolating page tables from unrelated kernel components
- Data-flow integrity (DFI) and software fault isolation (SFI)
  - DFI and SFI can prevent POP by limiting arbitrary memory read and write vulnerabilities

Mitigations

- Intel Virtualization Technology-Redirect Protection (VT-rp)
  - The **Hypervisor-managed Linear Address Translation (HLAT)** feature of VT-rp specifically aims to mitigate page remapping attacks
  - When the feature is enabled, it translates GLAs to GPAs instead of relying on page tables within the guest OS

Conclusion

- We analyzed blind spots in kernel CFI implementations for commodity OSes
  - Their focus was on ensuring page-level NWC and verifying the targets of indirect branches

- We introduced a novel POP technique capable of bypassing state-of-the-art kernel CFI implementations
  - We exploited these blind spots and evaluated POP

- We proposed potential mitigations against POP
  - POP can be hindered by various software- and hardware-based methods

Questions?

Seunghun Han

✉ hanseunghun@nsr.re.kr