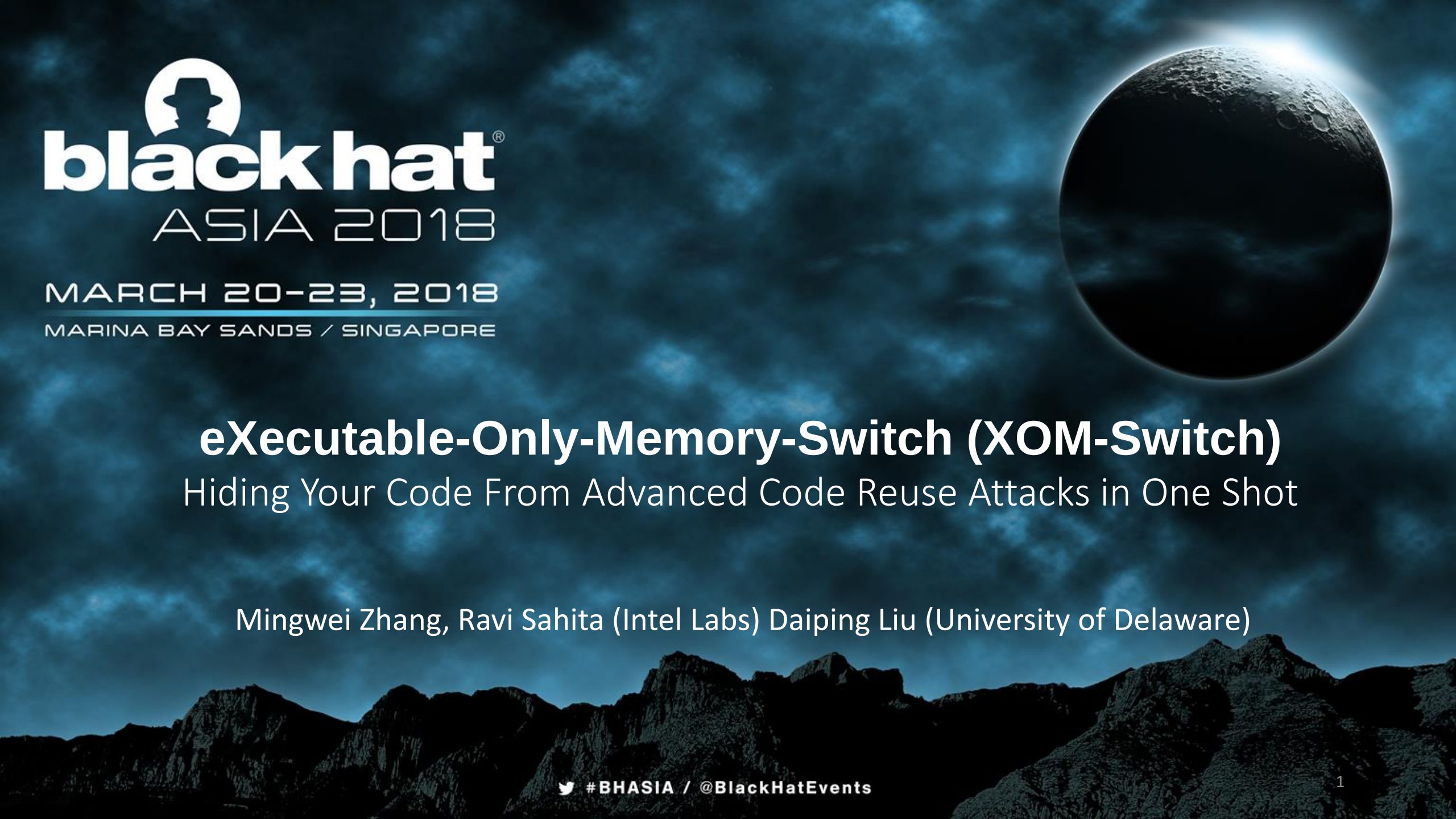


eXecutable-Only-Memory-Switch (XOM-Switch)

Hiding Your Code From Advanced Code Reuse Attacks in One Shot

Mingwei Zhang, Ravi Sahita (Intel Labs) Daiping Liu (University of Delaware)

[Short BIO of Speaker]

Mingwei Zhang is currently a research scientist in Anti-Malware Team in Intel Labs.

His current research areas span a wide range of topics including program hardening using Intel hardware features, anti-malware techniques, dynamic sandboxing for Android and etc. Mingwei received his Ph.D of computer science from Stony Brook University in 2015. His research in the Ph.D program was focused on software security protection via binary rewriting and program analysis.

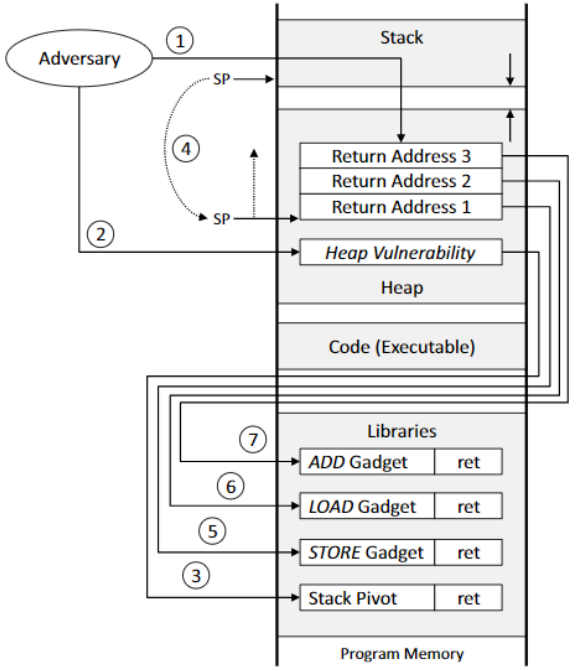

https://www.linkedin.com/in/mingwayzhang

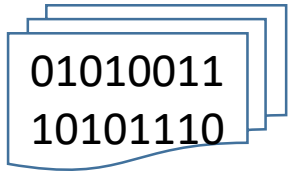

https://scholar.google.com/citations?user=llCSAtwAAAAJ&hl=en


http://www3.cs.stonybrook.edu/~mizhang/

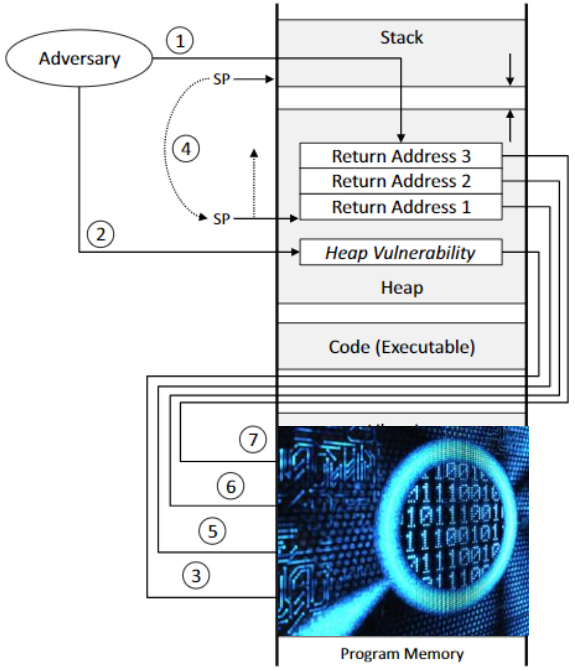
- Agenda
  - Problem Statement and Motivation
  - Intel’s protection keys design and implementation.
  - Binary rewriting in program loader
  - Evaluation
  - Conclusion

- Code Reuse Attacks
  - Effective way to bypass DEP, i.e., executing code without code injection.
  - Code reuse attacks requires accurate locations of “gadgets”, which may
  - Suffer from code diversity and availability.
- Advanced Code Reuse Attacks.
  - **Designed to solve the problem of gadget availability**



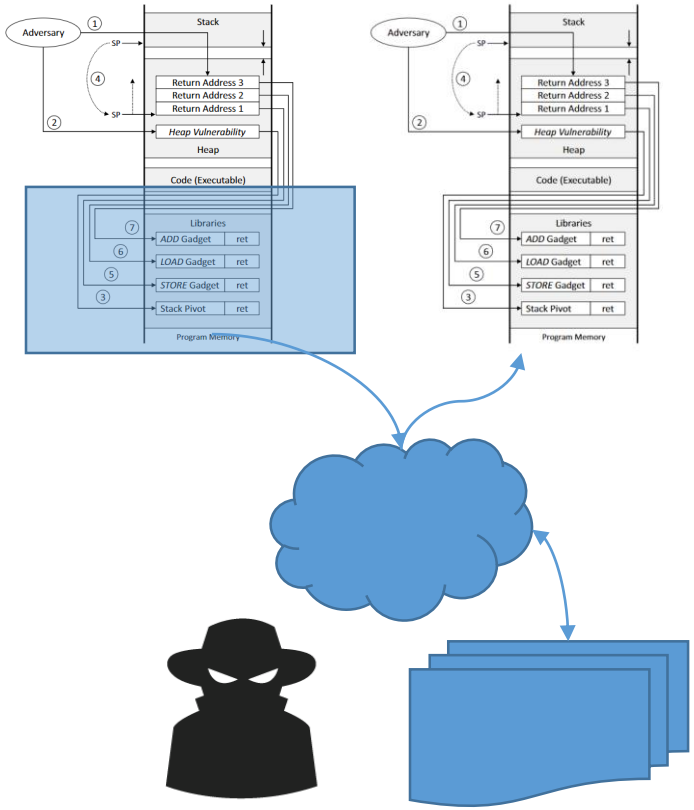


Traditional Code Reuse Attacks



Just-In-Time Code Reuse Attacks

1st ROP harvest code pages **2st ROP launch real attack**



Blind Code Reuse Attacks

- Reason of Advanced ROP:
  - **Convenience**:
    - Robustness of attacks on binaries in many versions.
    - Robustness on defenses like fine grained code randomizations.
  - **Larger attack vector**:
    - Both JIT-ROP and BROP makes significant threat to close-source/private-distribute binaries.

Advanced ROP attacks require code reading capability. This is why defenses on eXecutable Only Memory.

- Prevention on advanced code reuse focuses on eXecutable Only Memory.
  - Using page fault handler: XnR (CCS ’14)
  - Using Extended Page Table: X^R
  - Using side effects in micro-architecture: HideM
  - Using hardware support in ARM: NORAX
- **They are all beautiful, but they have their own drawbacks☺**
  - High runtime overhead.
  - Require Hypervisor support (nested virtualization in cloud?)
  - Significant effort on Code Refactoring/Rewriting/Recompilation.
  - Not available on x86 architecture (not available for cloud apps)

eXecutable Only Memory could be easily achieved using new Intel hardware capability

Intel Protection Keys

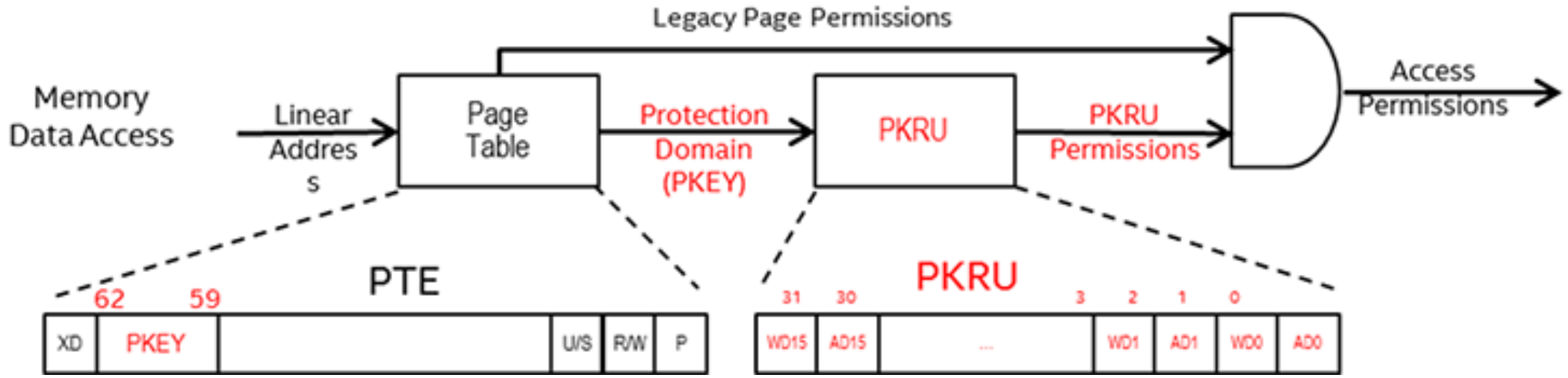
- Memory Protection Keys (MPK)
- *"Intel's memory protection keys feature works by making use of four page-table bits to assign one of sixteen key values to each page. A separate register then allows the assertion of 'write-disable' and 'access-disable' bits for each key value. Setting the write-disable bit for key seven, for example, will cause all pages marked with that key as being read-only, even if the protection bits on those pages would otherwise allow write access. The write- and access-disable bits are local to each thread, and they can be modified without privilege. Since keys are assigned to pages in the page-table entries, only the kernel can change those."*

  [LWN: https://lwn.net/Articles/643797/]

- Memory protection keys is described in Intel® 64 and IA-32 Architectures Software Developer's Manual [Volume 3A]

- TL;DR ?

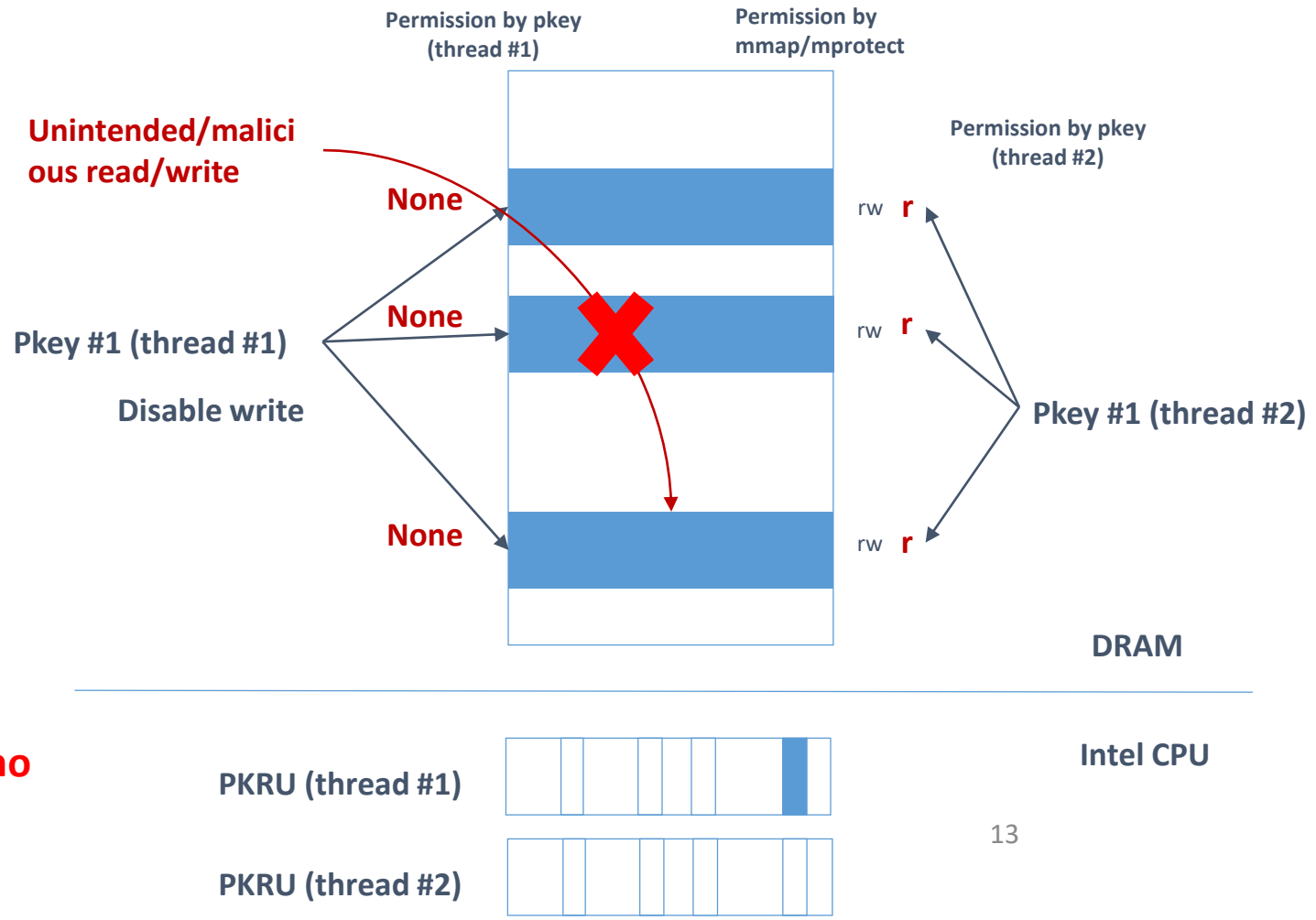
- What is Intel's MPK?
  - **Protection Keys**. [section 4.6.2 in Intel SDM]
  - Tagging memory pages with extra permission bits.
- Properties?
  - Fast permission switches of user level pages.
  - Allows pages to be "**read-only**" or "**inaccessible**".
  - Support 16 memory domains per process.



- Access permission is decided jointly by page permission **&** Protection Keys permission
- *pkey* bits are used as an "index" to PKRU and each has two bits.
- *pkey* applies to only user level pages (U/S=1)
- Supervisor accesses subject to the same checks as user accesses
  - If pkey denies access, direct memory accesses from kernel are also rejected.

- Primitives: 3 new syscalls and 2 new instructions added:
  - **New Syscall: int pkey_alloc(unsigned long flags, unsigned long initial_rights);**
  - **New Syscall: int pkey_mprotect(void *start, size_t len, int prot, int pkey);**
  - **New Syscall: int pkey_free(int key);**
  - **New Insn: wrpkru /* change memory permission of pages that bind to a pkey. */**
  - **New Insn: rdpkru /* get the memory permission of a pkey */**

- Turned on for each process.
- 16 keys per process
- Each key could bind to a large number of non-contiguous memory regions.
- Permission change by one instruction “wrpkru”
- Permission is per-thread based.



Performance: 60 ~ 120 cycles for wrpkru. Almost no relevance to memory range size. In compare, one “mprotect” could cause 20,000 cycles.

```
void main()
{
        int real_prot = PROT_READ | PROT_WRITE;
        int pkey = pkey_alloc();
        char * ptr = mmap(NULL, PAGE_SIZE, PROT_READ|PROT_WRIE,
                                MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
        ret = pkey_mprotect(ptr, PAGE_SIZE, real_prot, pkey);
        pkey_set(pkey, PKEY_DISABLE_WRITE, 0);
        *ptr = 0x30;
}
```

Why does Protection Keys have anything to do with eXecutable Only Memory ?

```
static inline void
wrpkru(unsigned int pkru)
{
   unsigned int eax = pkru;
   unsigned int ecx = 0;
   unsigned int edx = 0;

   asm volatile(".byte 0x0f,0x01,0xef\n\t"
            : : "a" (eax), "c" (ecx), "d" (edx));
}

int pkey_set(int pkey, unsigned long rights, unsigned long flags)
{
   unsigned int pkru = (rights << (2 * pkey));
   wrpkru(pkru);
   return 0;
}
```

pkey_set() disables the memory write access using wrpkru instruction (opcode: 0x0f 0x01)

- Protection Keys can be used for eXecutable Only Memory
  - **Marking a code page “inaccessible” does not prevent execution**
- eXecutable Only Memory Supported in Linux 4.9+
  - **Enhanced mprotect(addr, PROT_EXEC) = 0**
  - Makes a code page executable only.
- Update: glibc adopts Protection Keys support in 12/2017. However,

Support of XOM is missing in both GLIBC and compiler!

Recompiling/rewriting code is needed

Applying Protection Keys based eXecutable Only Memory to ELF Binaries.

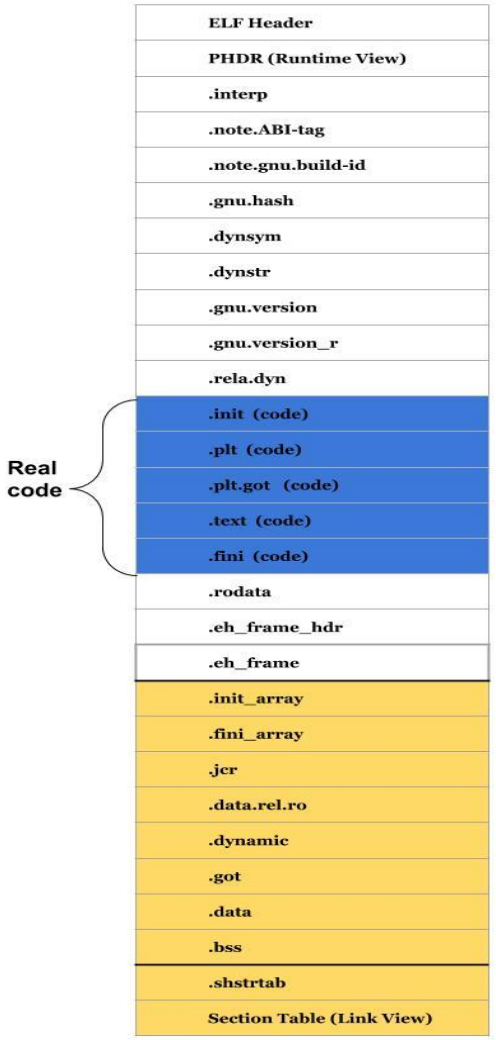
- We use a static binary rewriting to enable Protection Keys on ELF executables and libraries with the following key features:
  - **No source code or significant binary rewriting/translation needed.**
  - **Almost no runtime overhead and works on large applications.**
  - **Open source (GPLv2 and later) for community.**

- **Assumptions**
  - You have Intel CPU with Protection Keys feature turned on AND
  - You have Linux kernel 4.9+ that supports Protection Keys
  - **OR** you have Amazon AWS account and launch an **C5 instance** ☺ [1]

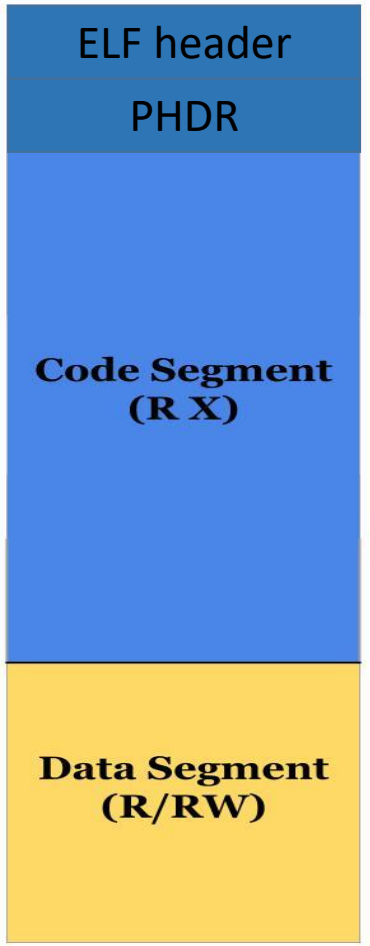
- **Idea**
  - **Identifying code pages of a program at load time and marking them as executable.**
- **Challenges**
  - Applying Protection Keys accurately on just code pages.
  - Applying Protection Keys on all ELF binaries including dependent libraries
  - Attackers may subvert XOM by abusing wrpkru/xsave

- Challenge #1: Code and data mixed in Binary.
  - ELF has two “views”

Information is lost from link time to runtime.



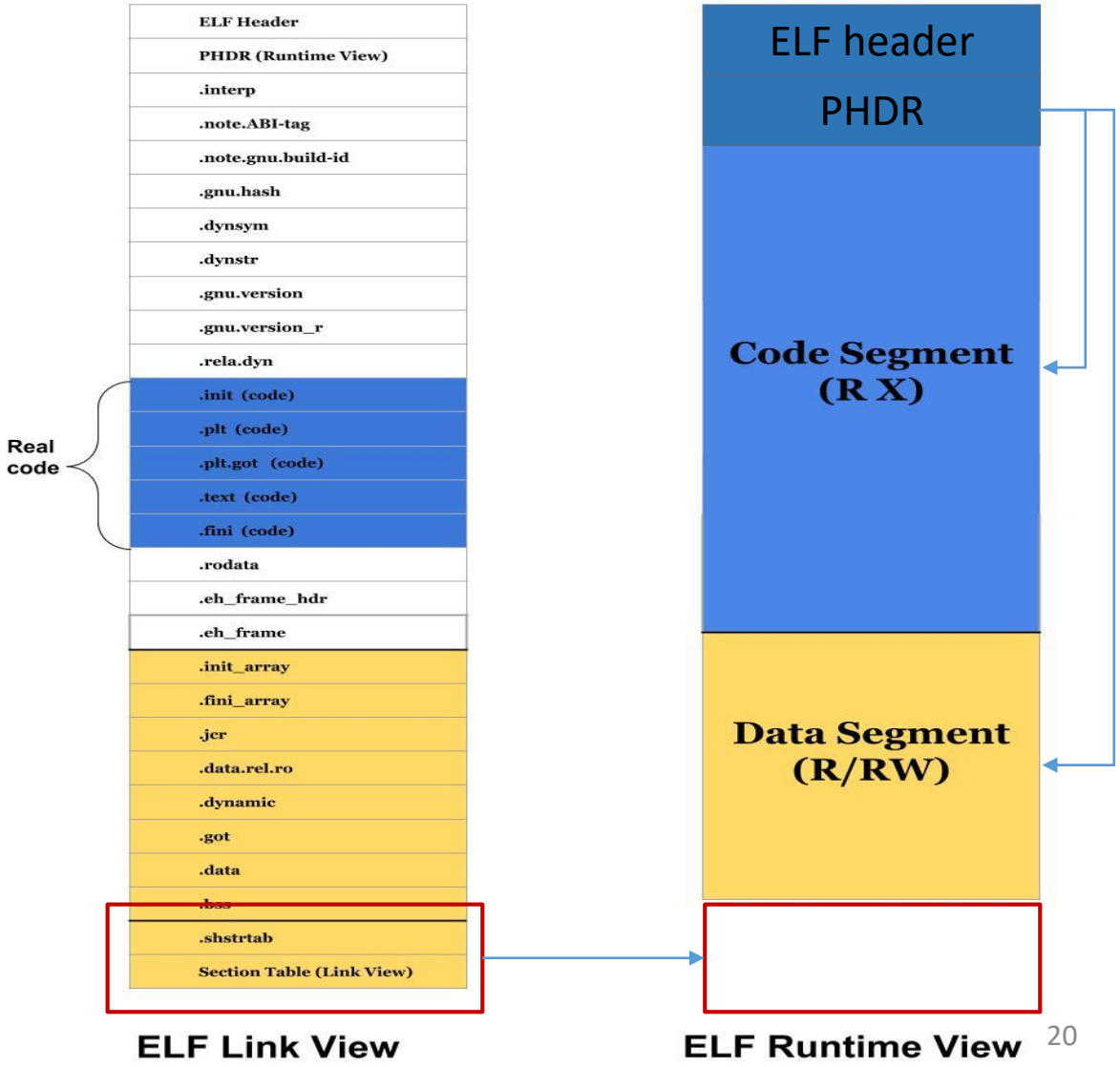
ELF Link View



ELF Runtime View

- Challenge #1: Code and data mixed in Binary.
  - ELF has two “views”
  - Runtime code segment mixes:
    - ELF metadata
    - Read-only data
    - Data in the middle of code (jump tables, lookup tables and/or compiler issues).

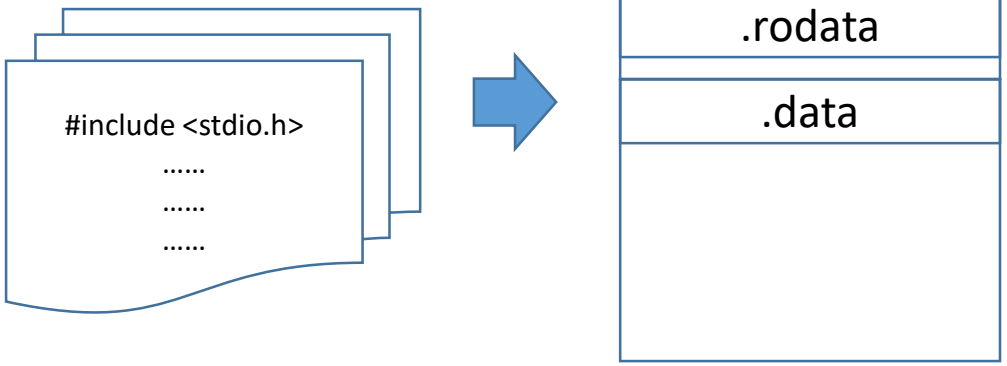
We use section table information to discover code pages.



- **Challenge #2: Applying Protection Keys on all ELF binaries.**
  - Need to do it at runtime…
    - executable path is known, but libraries may not be known until runtime.
  - Need to find the right method/place to hook.
    - LD_PRELOAD/LD_LIBRARY_PATH: **too late**
    - Protection Keys enabling in kernel: **cumbersome**
    - Recompiling program loader (ld.so): **cumbersome/unstable**
      - recompile the whole glibc libraries.
      - ld.so incompatible with libc.so.6 in different compilation.
      - **Recompiled glibc may have compatibility issues with other libs (e.g, libstdc++.so.6)**

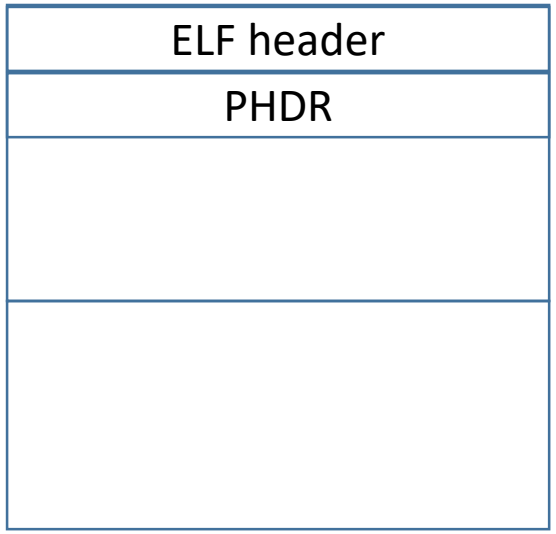
XOM-switch does not require recompilation of glibc or heavyweight binary rewriting

- **Steps of XOM-Switch:**
  - **Develop a binary that inspects ELF structure using C.**
  - **Extract out the code/rodata/data of the binary .**
  - **Patch the program loader with the extracted code/data.**
  - **Inject code at runtime and mark code pages exe only.**



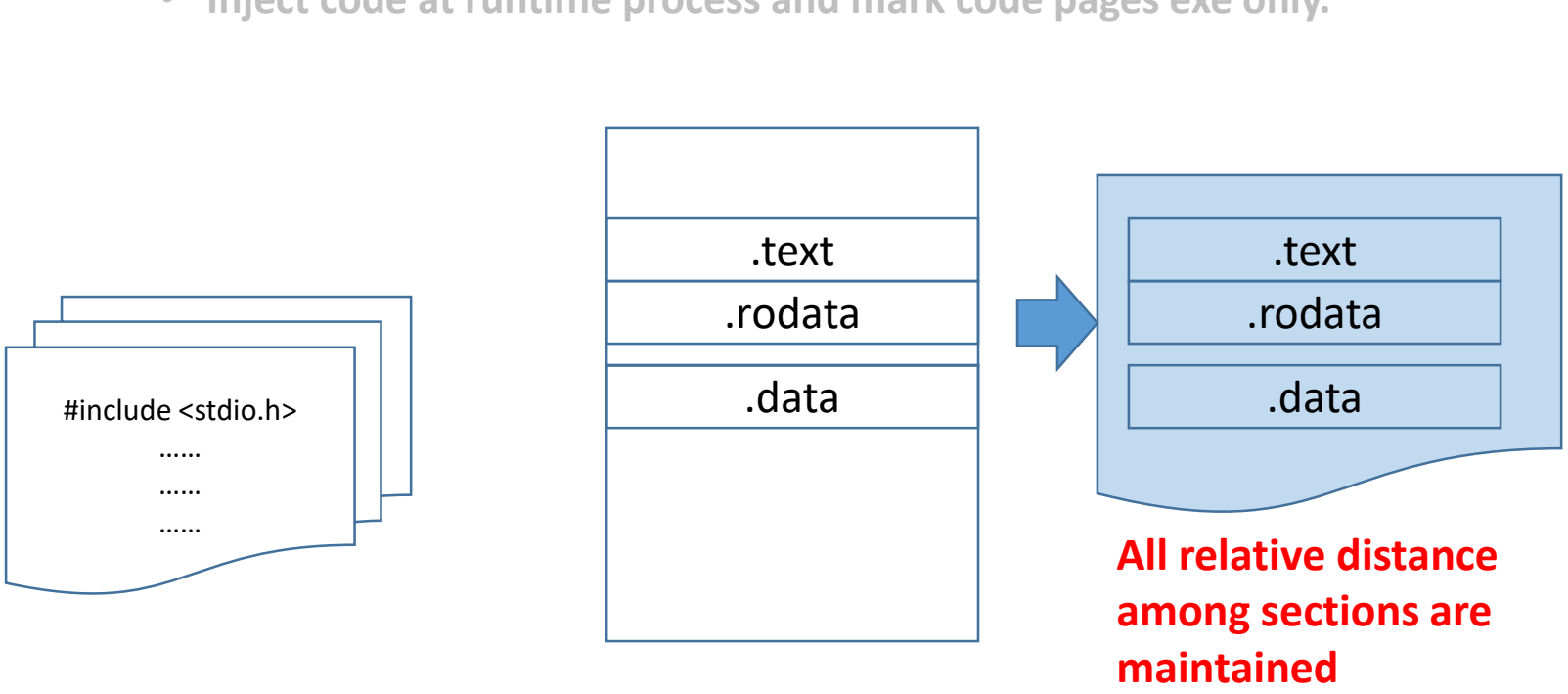
XOM Source Code

XOM Enabling Binary

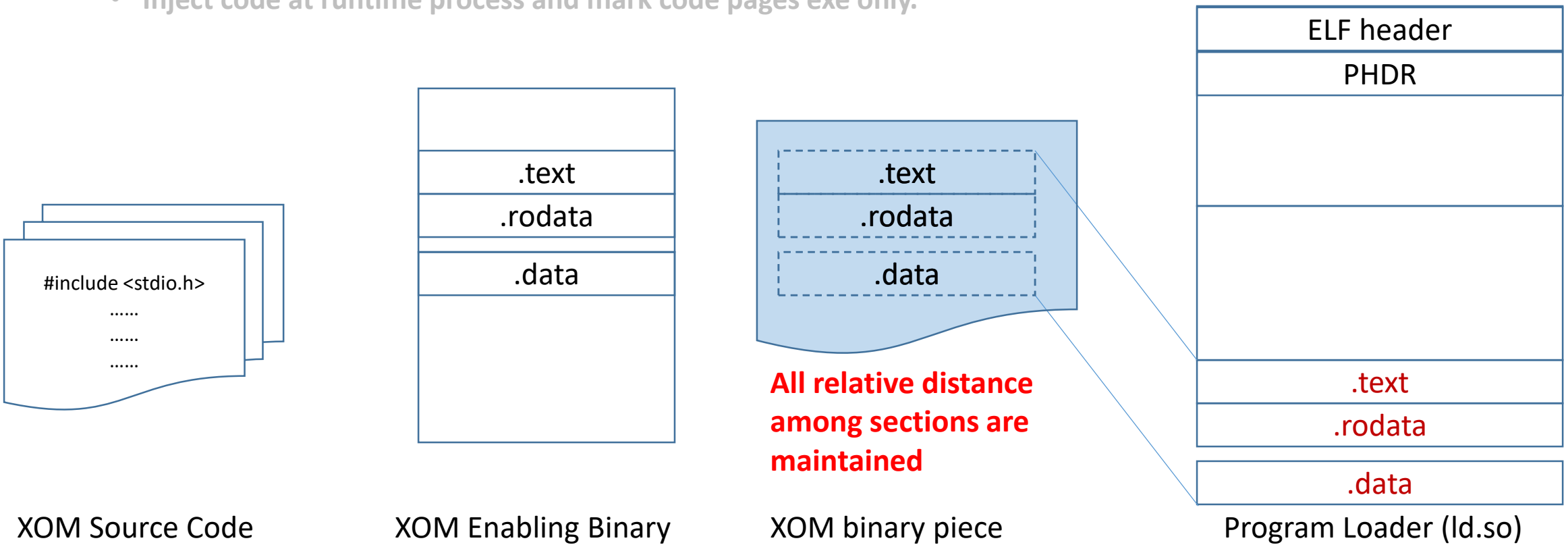


Program Loader (ld.so)

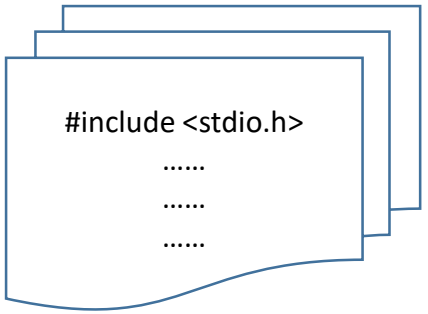
- **Steps of XOM-Switch:**
  - Develop a binary that inspects ELF structure using C.
  - **Extract out the code/rodata/data of the binary .**
  - Patch the program loader with the extracted code/data.
  - Inject code at runtime process and mark code pages exe only.



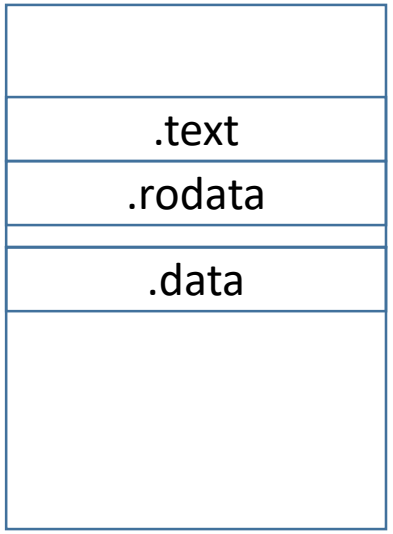
- **Steps of XOM-Switch:**
  - Develop a binary that inspects ELF structure using C.
  - Extract out the code/rodata/data of the binary.
  - **Patch the program loader with the extracted code/data.**
  - Inject code at runtime process and mark code pages exe only.



- **Steps of XOM-Switch:**
  - Develop a binary that inspects ELF structure using C.
  - Extract out the code/rodata/data of the binary.
  - **Patch the program loader with the extracted code/data.**
  - Inject code at runtime and mark code pages exe only.

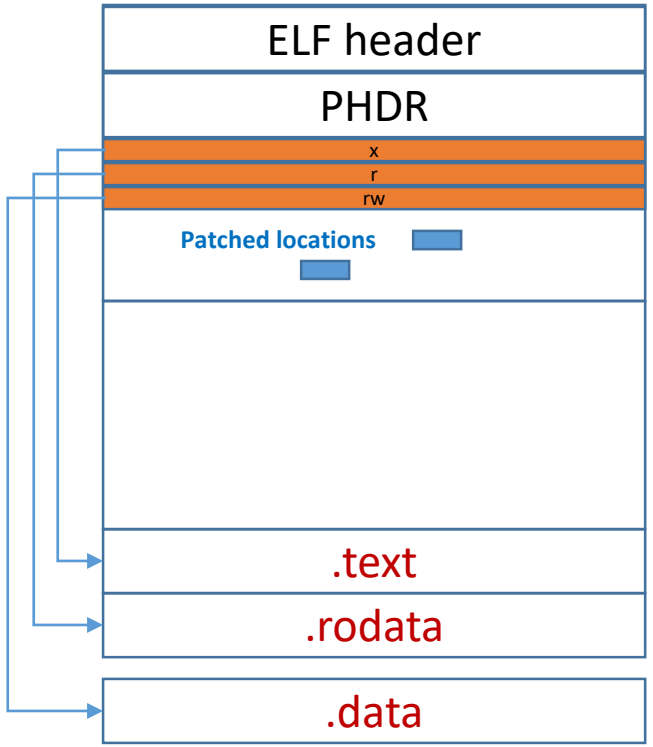


XOM Source Code



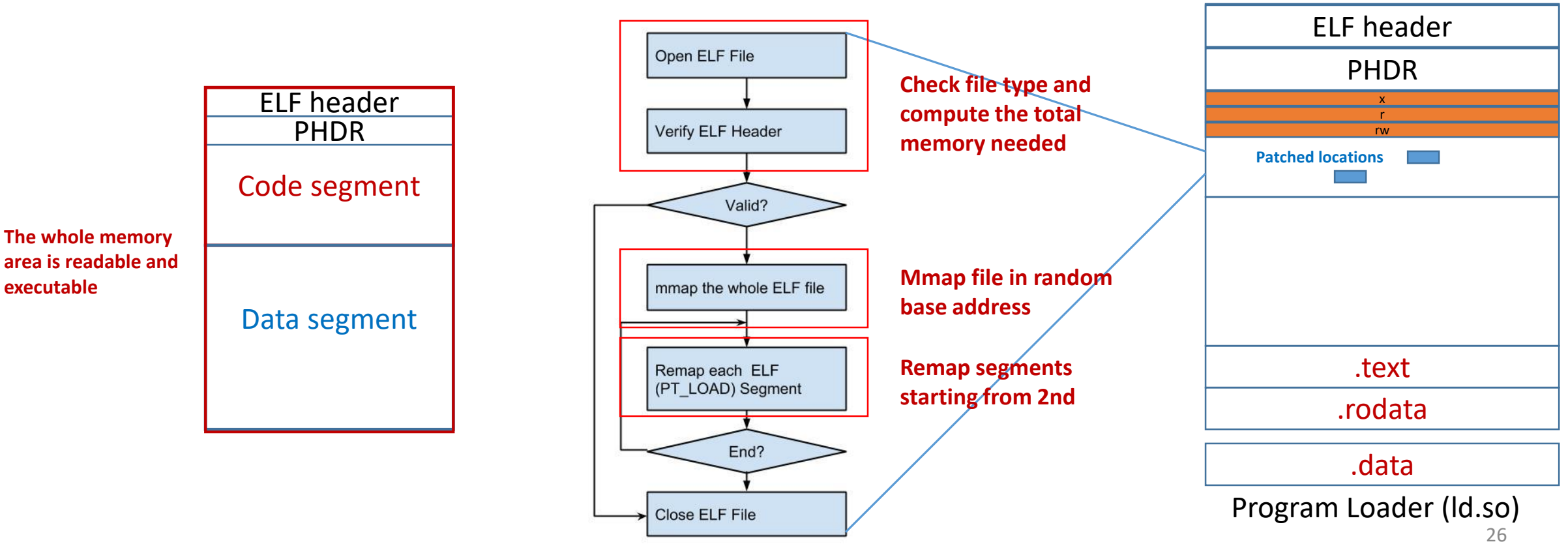
XOM Enabling Binary

Extending PHDR of ld.so with three PT_LOAD segments.



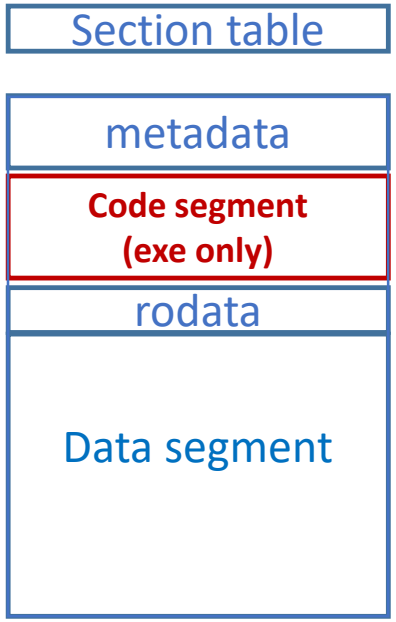
Program Loader (ld.so)

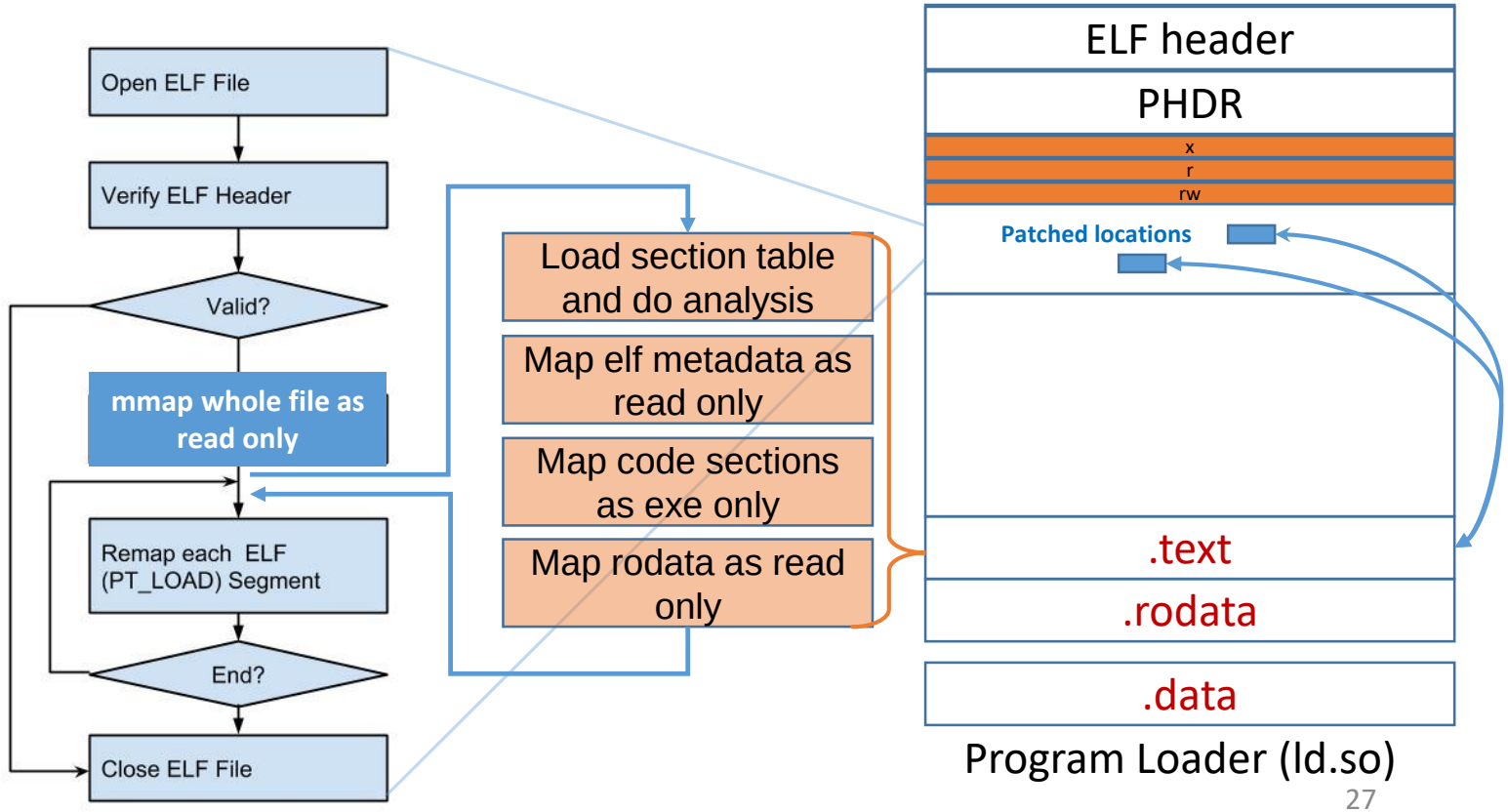
- **Steps of XOM-Switch:**
  - **Inject code at runtime process and mark code pages exe only.**
  - **Let's see the normal ELF binary loading process in glibc:**



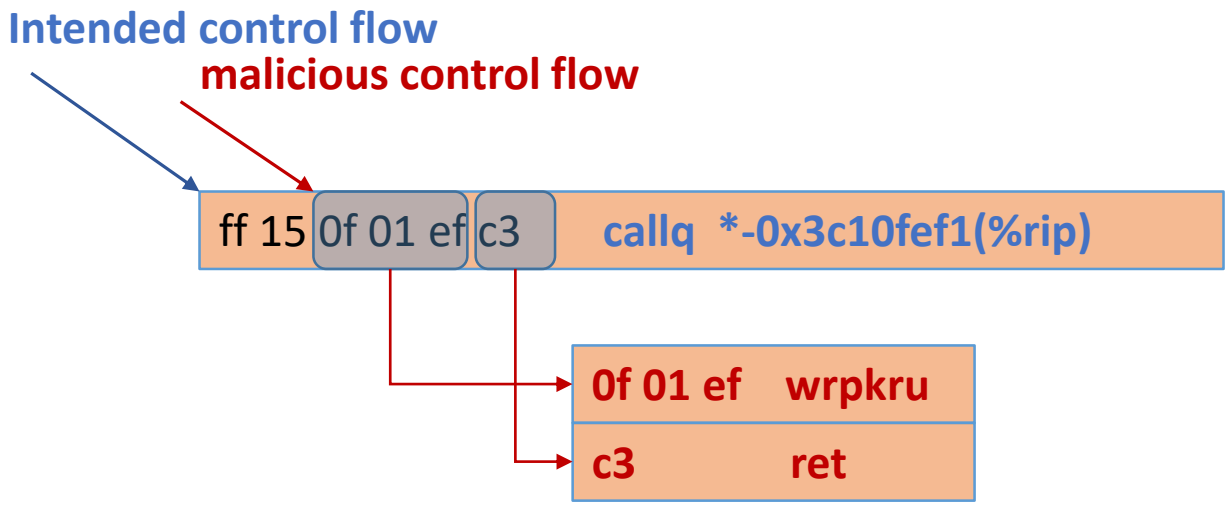
- **Steps of XOM-Switch:**
  - **Injected code at runtime process all ELF loaded and mark code pages exe only.**
  - **Now let's see the modified (patched) ELF binary loading process**

The whole memory area initially is read only



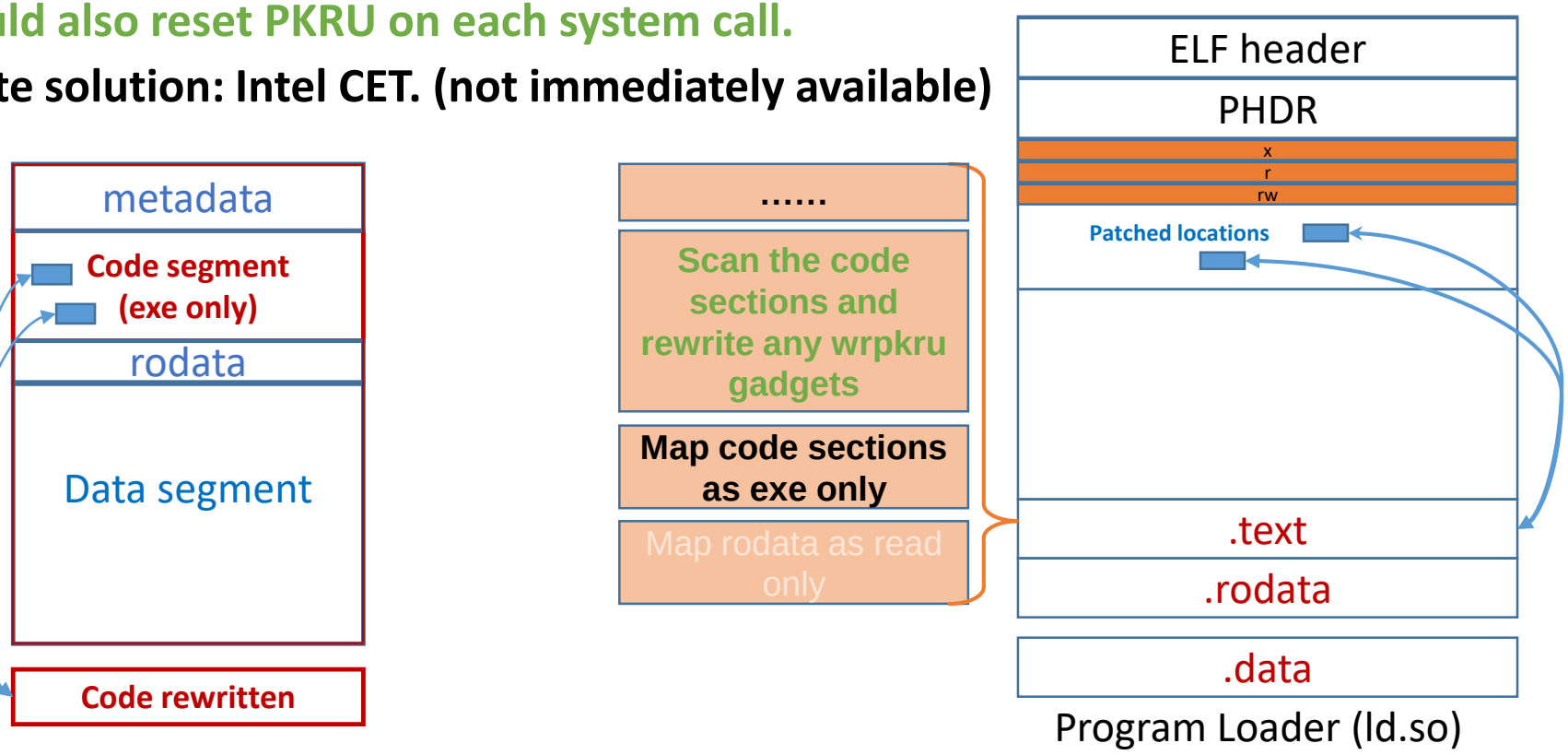


- **Challenge #3: Abusing Protection Keys to disable XOM**
  - **Attackers may use gadgets that contain wrpkru/xsave to disable XOM!**



- **Challenge #3: Abusing Protection Keys to disable XOM**
  - We could potentially scan the code sections and rewrite dangerous instructions
  - We could also reset PKRU on each system call.
  - **Ultimate solution: Intel CET. (not immediately available)**

The defense is limited in power, but could be helped by CFI/code randomizations.



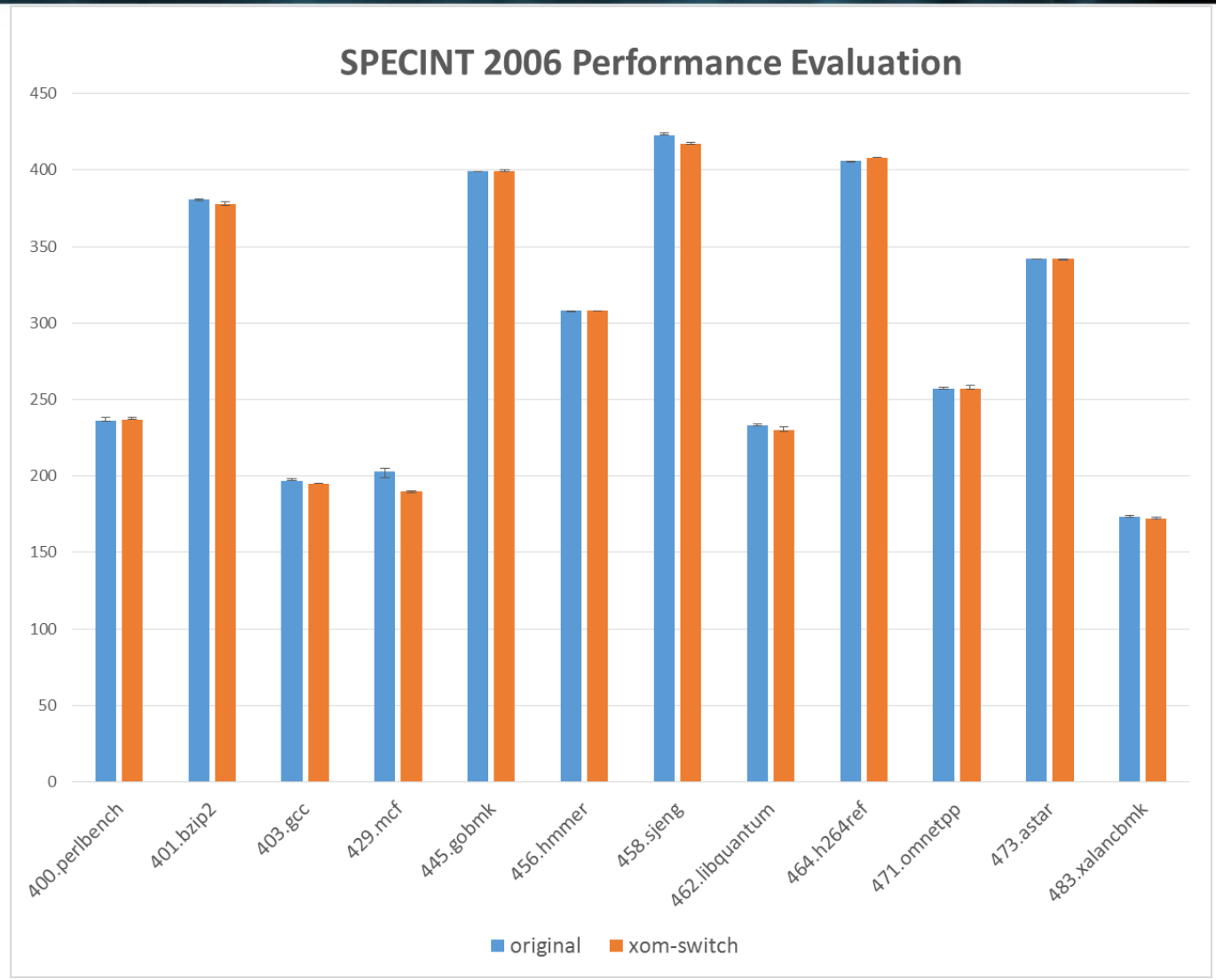
Defense #2: Linux kernel: reset PKRU at each system call

Evaluation

- **Cost**
  - Code Size Increase:
    - ld.so: **7%**
    - libc.so: **0.9% (optional)**
    - Other binaries: **0% (no change)**
  - Code Loading Overhead:
    - A few extra system calls:
      - 1 mmap and 1 munmap for section table loading
      - 3~5 mprotect for permission changes in code segment

- **Cost**
  - The average overhead is:

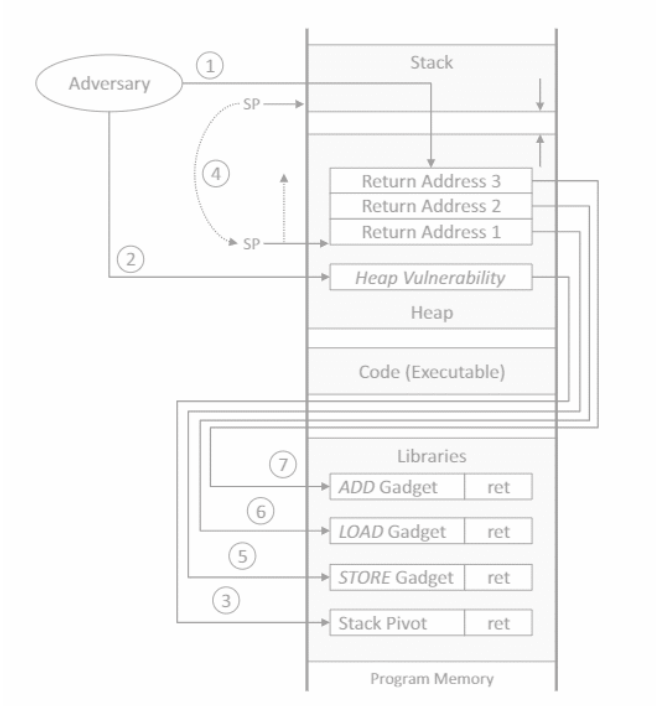
    0% (±1%)

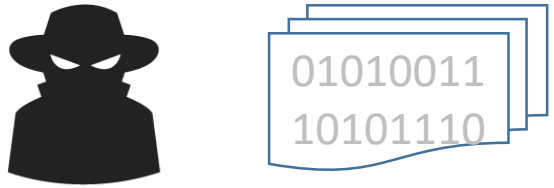


- Effectiveness
  - CPU: Intel CPU with Protection Keys enabled
  - OS: Ubuntu 17.04 with Linux kernel 4.10.0-21-generic
  - Glibc: glibc-2.24

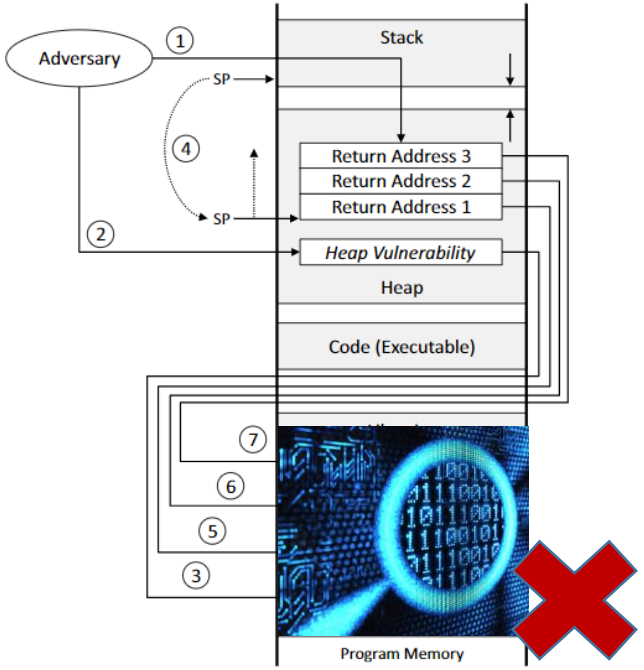
Application Name	Libraries Loaded	Original Code Size (KB)	XOM enabled Code/Data (KB)			
			Readable Code	XOM Code	Read-Only "Code"	Total Reduction
Firefox v54	130	104,032	1480 (1.42%)	60324 (57.98%)	42228 (40.59%)	98.57%
soffice.bin (LibreOffice)	118	144,336	1360 (.94%)	66876 (46.33%)	76100 (52.72%)	99.05%
Kile	132	101,804	1060 (1.04%)	44624 (43.83%)	56120 (55.12%)	98.95%

- Effectiveness





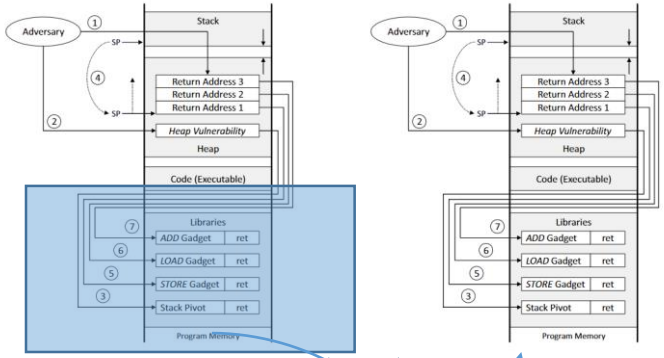
Traditional Code Reuse Attacks



Just-In-Time Code Reuse Attacks

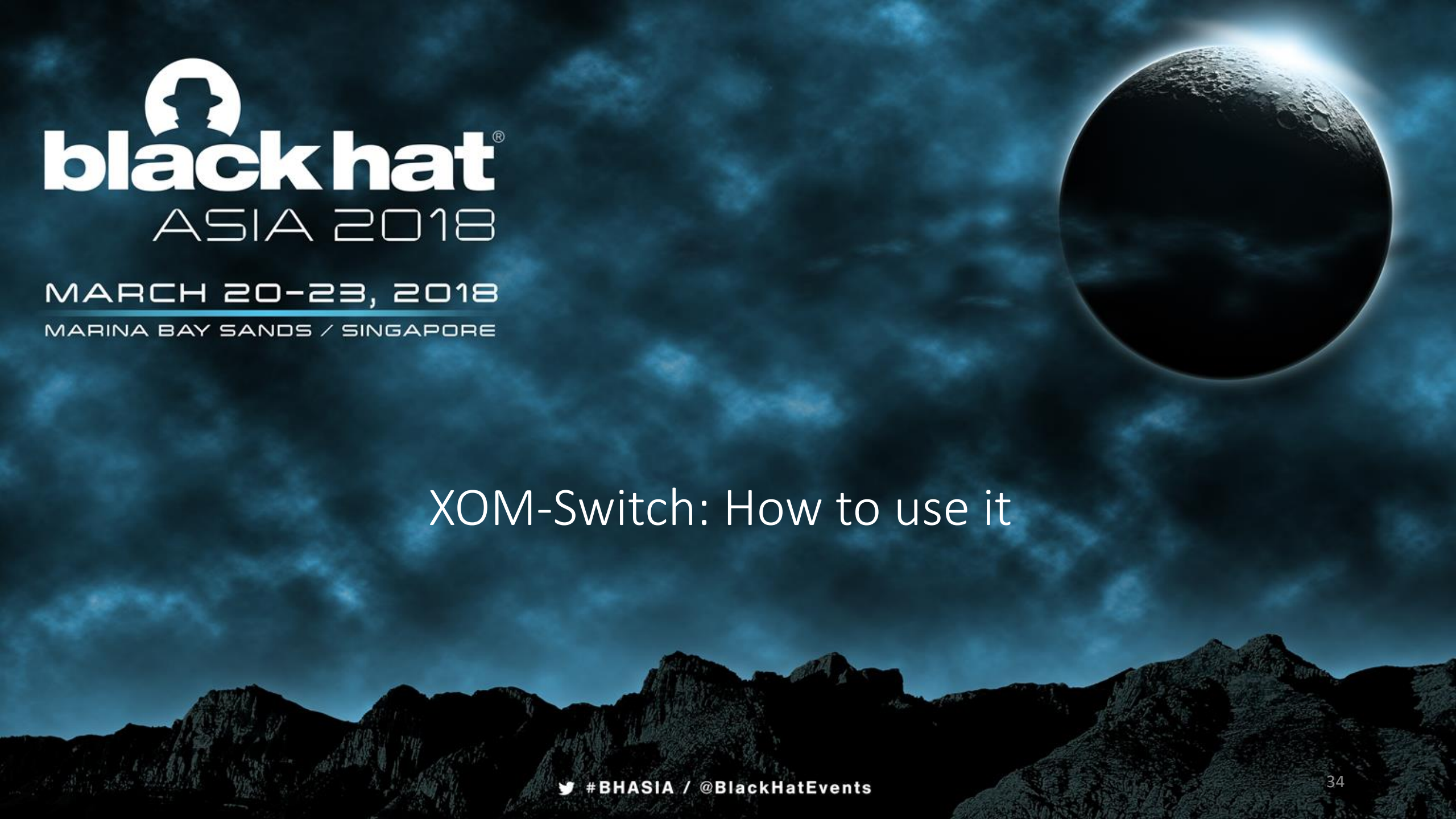
1st ROP harvest code pages

2st ROP launch real attack



XOM cannot be read from kernel

Blind Code Reuse Attacks



XOM-Switch: How to use it

- Download the source code
  - https://github.com/intel/xom-switch.git
- Dependency
  - Make sure you have python 2.7;
  - Download radare2: https://github.com/radare/radare2.git
  - Setup radare2 path properly.
- Binary patching
  - Transform your program loader:

    src/analysis/patch_loader.sh /lib64/ld-linux-x86-64.so.2 ./your_new_ld.so

    sudo cp ./your_new_ld.so /lib64/ld-xom.so
  - Transform your libc.so (optional):

    src/analysis/patch_libc.sh /path/to/your/libc.so.6 ./your_new_libc.so

- To Run:

  /lib64/ld-xom.so /usr/lib/firefox/firefox

  LD_PRELOAD=/path/to/your/libc.so /lib64/ld-xom.so /usr/lib/firefox/firefox

- Verify:
  - Check /proc/your_pid/maps and see memory permission maps

- Acknowledgements
  - Thanks to Ravi Sahita, Deepak Gupta, Michael LeMay, David Durham, Andy Anderson, David Koufaty, Dave Hansen for all the support on this work.

- References:
  - https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf
  - https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf
  - https://www.kernel.org/doc/Documentation/x86/protection-keys.txt

XOM-SWITCH: Hiding Your Code From Advanced Code Reuse Attacks in One Shot

Source code: https://github.com/intel/xom-switch.git

Contact: mingwei.zhang@intel.com

Question ?

- System Internals and Overcome work…
- Transparency Issues
  - System calls
  - Modified program loader
- Data embedded in the middle of code
  - Making exceptions: libavcodec.so; libcrypto.so; … (all 4)
  - Making fine grained policies