上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

Architecture Support for System Security

Building security from solid base

Yubin Xia

Security: Why Hardware?

- Security is a negative goal
  - How to make a program not do something?
  - E.g., not execute any code from user, not leak some secret from memory, etc.
- Hardware features based security
  - Fixed and robust (hopefully)
  - More efficient

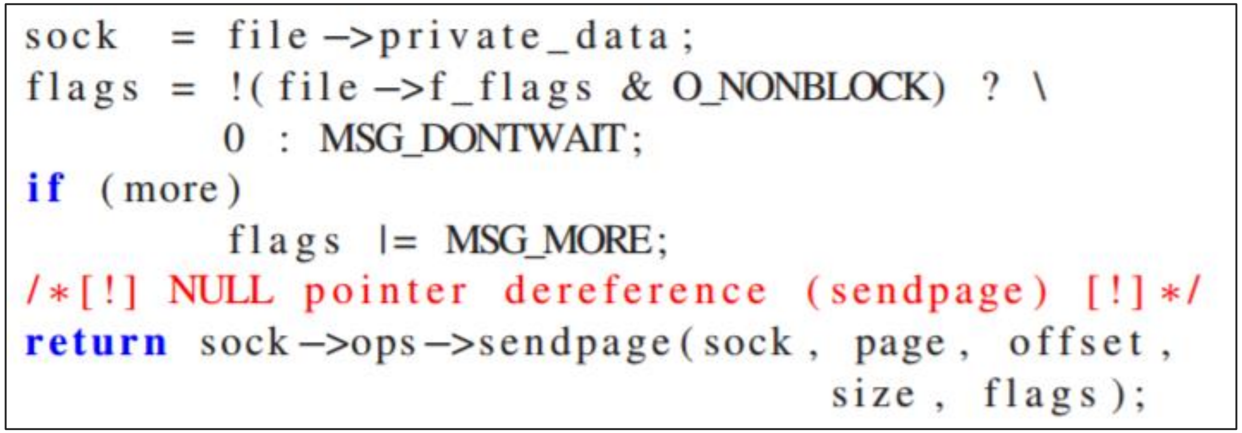
SMEP: Supervisor Mode Execution Prevention

SMAP: Supervisor Mode Access Prevention

SMEP/SMAP

Return-to-user Attack

NULL function pointer in Linux (net/socket.c)



```
sock  = file->private_data;
flags = !(file->f_flags & O_NONBLOCK) ? \
         0 : MSG_DONTWAIT;
if (more)
         flags |= MSG_MORE;
/*[!] NULL pointer dereference (sendpage) [!]*/
return sock->ops->sendpage(sock, page, offset,
                            size, flags);
```

- The value of **sendpage** is set to **NULL**
  - What will happen then?

Return-to-user Attack

NULL data pointer in Linux (fs/splice.c)

```
/*[!] NULL pointer dereference (ops) [!]*/
ibuf->ops->get(ipipe, ibuf);
obuf = opipe->bufs + nbuf;
*obuf = *ibuf;
```

- The *ops* field becomes **NULL** after the invocation of the **tee** system call
  - Then *get()* will be at 0x0000001C, which is controlled by the app

SMAP

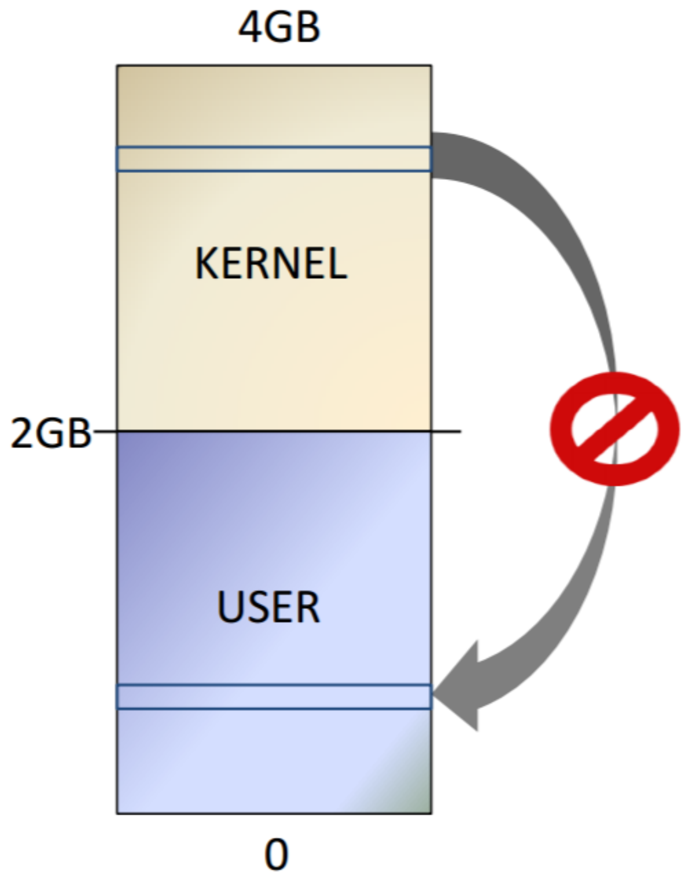
- SMAP: Supervisor Mode Access Prevention
  - Allows pages to be protected from supervisor-mode data accesses
  - If SMAP = 1, OS cannot access data at linear addresses of application

SMEP

- SMEP: Supervisor Mode Execution Prevention
  - Allows pages to be protected from supervisor-mode instruction fetches
  - If SMEP = 1, OS cannot fetch instructions from application
- Introduced at Intel processors from Ivy Bridge
  - Security feature launched in 2011
  - Enabled by default since Windows 8.0 (32/64 bits)

Prevent Return-to-user Attack

- The CPU will prevent the OS from executing user-level instructions

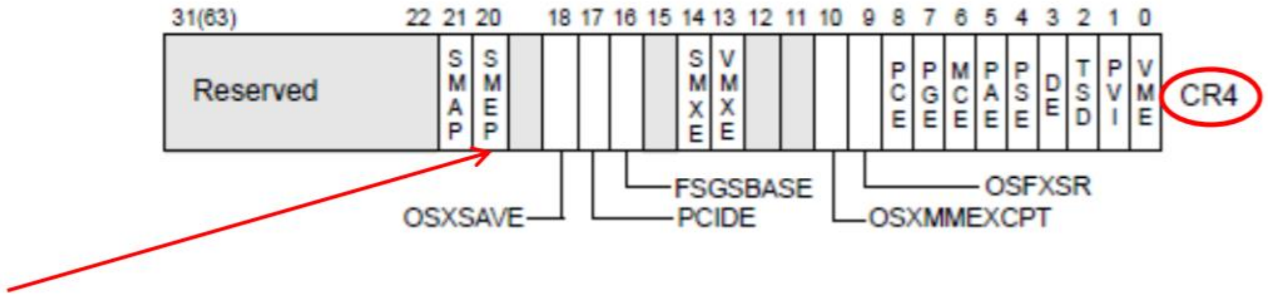


SMEP CPU Support

- Desktop processors
  - Intel Core: Latest models of i3, i5, i7
  - Intel Pentium: G20X0(T) and G21X0(T)
  - Intel Celeron: G1610(T), G1620(T) and G1630
- Server processors
  - Intel Xeon: Latest models of E3, E5, E7
  - Intel Pentium: 1403v3 and 1405v2

How to Bypass SMAP/SMEP?

- 1. Turn off the bit 20th/21st of the CR4 register
  - E.g., "mov rax,0xFFFEFFFFF" / "mov cr4,rax" / "ret "
  - Jump to USER SPACE



Memory Protection eXtension

MPX

Bounds Error of Software

- C/C++ programs are prone to bounds errors
  - Not type-safe language
  - E.g., buffer overflow bugs

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define noinline __attribute__((noinline))

char dog[] = "dog";
char password[] = "secr3t";

noinline
char dog_letter(int nr)
{
        return dog[nr];
}

int main(int argc, char **argv)
{
        int max = sizeof(dog);
        int i;

        if (argc >= 2)
                max = atoi(argv[1]);

        for (i = 0; i < max; i++)
                printf("dog[%d]: '%c'\n", i, dog_letter(i));

        return 0;
}
```

Bounds Error

Running that program with a bad input ("10" is longer than "dog") can yield unexpected results

```
dog[0]: 'd'
dog[1]: 'o'
dog[2]: 'g'
dog[3]: ''
dog[4]: 's'
dog[5]: 'e'
dog[6]: 'c'
dog[7]: 'r'
dog[8]: '3'
dog[9]: 't'
```

Bounds Error

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define noinline __attribute__((noinline))

char dog[] = "dog";
char password[] = "secr3t";

noinline
char dog_letter(int nr)
{
        return dog[nr];
}

int main(int argc, char **argv)
{
        int max = sizeof(dog);
        int i;

        if (argc >= 2)
                max = atoi(argv[1]);

        for (i = 0; i < max; i++)
                printf("dog[%d]: '%c'\n", i, dog_letter(i));

        return 0;
}
```

```
gcc -Wall -o mpx-out-of-bounds -mmpx -fcheck-pointer-bounds mpx-out-of-bounds.c
```

```
dog[0]: 'd'
dog[1]: 'o'
dog[2]: 'g'
dog[3]: ''
Saw a #BR! status 1 at 0x317004
dog[4]: 's'
Saw a #BR! status 1 at 0x317004
dog[5]: 'e'
Saw a #BR! status 1 at 0x317004
dog[6]: 'c'
Saw a #BR! status 1 at 0x317004
dog[7]: 'r'
Saw a #BR! status 1 at 0x317004
dog[8]: '3'
Saw a #BR! status 1 at 0x317004
dog[9]: 't'
```

MPX: Memory Protection eXtension

- Intel introduces **MPX** from Skylake
- Programmer can create and enforce bounds
  - Specified by <u>two 64-bit addresses</u> specifying the beginning and the end of a range
  - New instructions are introduced to efficiently compare a given value against the bounds, raising an exception when the value does not fall within the permitted range

Using Intel MPX

Without MPX

```
00000000004006e0 <dog_letter>:
  4006e0:       48 63 ff                movslq %edi,%rdi
  4006e3:       0f b6 87 43 10 60 00    movzbl 0x601043(%rdi),%eax
  4006ea:       c3                      retq
```

With MPX

```
0000000000400750 <dog_letter>:
  400750:       66 0f 1a 05 f8 08 20    bndmov 0x2008f8(%rip),%bnd0   # >__chkp_bounds_of_dog>
  400757:       00
  400758:       48 63 ff                movslq %edi,%rdi
  40075b:       48 8d 87 67 10 60 00    lea    0x601067(%rdi),%rax
  400762:       f3 0f 1a 00             bndcl  (%rax),%bnd0
  400766:       66 0f 1a 0d e2 08 20    bndmov 0x2008e2(%rip),%bnd1   # >__chkp_bounds_of_dog>
  40076d:       00
  40076e:       f2 0f 1a 08             bndcu  (%rax),%bnd1
  400772:       0f b6 87 67 10 60 00    movzbl 0x601067(%rdi),%eax
  400779:       f2 c3                   bnd retq
```

Intel MPX's New Instructions

- bndmov: Fetch the bounds information (upper and lower) out of memory and put it in a bounds register.
- bndcl: Check the lower bounds against an argument (%rax)
- bndcu: Check the upper bounds against an argument (%rax)
- bnd retq: Not a "true" Intel MPX instruction
  - The bnd here is a prefix to a normal retq instruction
  - It just lets the processor know that this is Intel MPX-instrumented code

Bounds Tables

- For efficiency, four bounds can be stored into dedicated registers
  - Registers: bnd0 to bnd3
  - When more bounds are required, they are stored in memory, and the bound registers serve as a caching mechanism
  - Bounds tables are a two-level radix tree, indexed by the virtual address of the pointer for which you want to load/store the bounds
  - The BNDLDX/BNDSTX instructions essentially take a pointer value and move the bounds information between a bounds register & bounds tables

Memory Consumption

- Bounds tables can consume and reference a lot of memory
  - A one-page (4 KB) data structure entirely filled with pointers will consume four pages (16 KB) of bounds tables because each bounds table entry contains four pointers' worth of data
  - In the worst case, bounds tables can cause an application to consume **500%** more memory compared to if the application was not using MPX

Performance of MPX

- While the bounds-checking itself is very efficient, the usage of many bounds is not
    - For instance, GCC's implementation of MPX buffer checking frequently spills bounds registers to memory

How to Use MPX?

```
make CFLAGS="-mmpx -fcheck-pointer-bounds -lmpx" LDFLAGS="-lmpxwrappers -lmpx"
```

```
# ldd ./mpx-out-of-bounds
    linux-vdso.so.1 (0x00007ffc4f5aa000)
    libmpx.so.0 => /lib64/libmpx.so.0 (0x00007ff42d223000)
    libmpxwrappers.so.0 => /lib64/libmpxwrappers.so.0 (0x00007ff42d021000)
    libc.so.6 => /lib64/libc.so.6 (0x00007ff42cc5f000)
    libpthread.so.0 => /lib64/libpthread.so.0 (0x00007ff42ca42000)
    /lib64/ld-linux-x86-64.so.2 (0x0000558cb43be000)
```

MPK: Memory Protection Keys

MPK

MPK: Memory Protection Keys

- With MPK, every page belongs to one of 16 domains
    - A domain is determined by 4 bits in every page-table entry (referred to as the protection key)
- For every domain, there are two bits in a special register (**pkru**)
    - Denote whether pages associated with that key can be read or written
- Kernel and application
    - Only the kernel can change the key of a page
    - Application can read and write the **pkru** register using the *rdpkru* and *wrpkru* instructions respectively

MPK: Memory Protection Keys

- Isolation can be enabled using MPK by placing the sensitive data in pages that have a particular protection key, forming the sensitive domain
- An appropriate instrumentation enables reads and/or writes to the data by setting the access disable and write-disable bits, respectively, using *wrpkru*
    - As long as these bits are unset, the sensitive domain is accessible
    - By setting the bits back, the sensitive domain is disabled, making only the non-sensitive domain available

MPK is not "New"

- Similar software implementation: using mprotect()
  - Application can already change the permission of pages
  - Why is MPK needed?
- Hardware for efficiency
  - Page table entries are expensive to manipulate
    - Require system call
    - A change requires invalidating translation lookaside buffer (TLB)
  - Using MPK, application can frequently change memory permission

Use Cases of MPK?

- Use case 1: **protect critical data**
  - Handling of sensitive cryptographic data
  - Only enable access to private key during encryption
- Use case 2: **prevent data corruption**
  - In-memory database prevents writes most of the time
  - Only enable changing data when needs to change
  - Changing protection on gigabytes using *mprotect()* is too slow

PA: Pointer Authentication

ARM PA

ARM Pointer Authentication

- ARM64 only use 40 bits out of 64 bits
  - On an ARM64 Linux system using three-level page tables, only the bottom 40 bits are used, while the remaining 24 are equal to the highest significant bit
  - The 40-bit address is sign-extended to 64 bits
  - Those uppermost bits (or a subset thereof) could be put to other uses, including holding an authentication code
- Use the 24 bits for security!

Key Management

- PA defines five keys
    - Four keys for PAC* and AUT* instructions (combination of instruction/data and A/B keys),
    - One key for use with the general purpose PACGA instruction
- Key storage
    - Stored in internal registers and are not accessible by EL0 (user mode)
    - The software (EL1, EL2 and EL3) is required to switch keys between exception levels
    - Higher privilege levels control the keys for the lower privilege levels

New Instructions

- PAC value creation
  - Write the value to the uppermost bits in a destination register alongside an address pointer value
- Authentication
  - Validate a PAC and update the destination register with a correct or corrupt address pointer
  - If the authentication fails, an indirect branch or load that uses the authenticated, and corrupt, address will cause an exception
- Removing a PAC value from the specified register

Use Case: Software Stack Protection

	No stack protection	Software Stack protection

Function Prologue	`SUB sp, sp, #0x40`
	`STP x29, x30, [sp,#0x30]`
	`ADD x29, sp, #0x30`
	…
		`SUB sp, sp, #0x50`
		`STP x29, x30, [sp, #0x40]`
		`ADD x29, sp, #0x40`
		`ADRP  x3, {pc}`
		`LDR   x4, [x3, #SSP]`
		`STR   x4, [sp, #0x38]`
		…

Function Epilogue	…
	`LDP x29,x30,[sp,#0x30]`
	`ADD sp,sp,#0x40`
	`RET`
		…
		`LDR      x1, [x3, #SSP]`
		`LDR      x2, [sp, #0x38]`
		`CMP      x1, x2`
		`B.NE     __stack_chk_fail`
		`LDP x29, x30, [sp, #0x40]`
		`ADD sp, sp, #0x50`
		`RET`

	No stack protection	With Pointer Authentication

Function Prologue	`SUB sp, sp, #0x40`
	`STP x29, x30, [sp,#0x30]`
	`ADD x29, sp, #0x30`
	…
		`PACIASP`
		`SUB sp, sp, #0x40`
		`STP x29, x30, [sp,#0x30]`
		`ADD x29, sp, #0x30`
		…

Function Epilogue	…
	`LDP x29,x30,[sp,#0x30]`
	`ADD sp,sp,#0x40`
	`RET`
		…
		`LDP x29,x30,[sp,#0x30]`
		`ADD sp,sp,#0x40`
		`AUTIASP`
		`RET`

Control-flow Enforcement Technology

CET

Intel CET

- Control-flow Enforcement Technology
- Two major techs
    - Shadow stack
    - Indirect branch tracking

Shadow Stack

- A shadow stack is a second stack for the program
  - Used exclusively for control transfer operations
  - Is separate from the data stack
  - Can be enabled for operation individually in user mode or supervisor mode

Shadow Stack Mode

- CALL instruction
    - Pushes the return address on both the data and shadow stack
- RET instruction
    - Pops the return address from both stacks and compares them
    - If the return addresses from the two stacks do not match, the processor signals a control protection exception (#CP)

- Note that the shadow stack only holds the return addresses and not parameters passed to the call instruction

Protecting the Shadow Stack

- The shadow stack is protected by page table
  - Page table support a new attribute: mark page as "Shadow Stack" pages
- Control transfers are allowed to store return addresses to the shadow stack
  - Like near call, far call, call to interrupt/exception handlers, etc.
  - However stores from instructions like MOV, XSAVE, etc. will not be allowed
- When control transfer instructions attempt to read from the shadow stack
  - Access will fault if the underlying page is not marked as a "Shadow Stack" page
- Detects and prevents conditions that cause an overflow or underflow of the shadow stack or any malicious attempts to redirect the processor to consume data from addresses that are not shadow stack addresses

Indirect Branch Tracking

- New instruction: **ENDBRANCH**
  - Mark valid indirect call/jmp targets in the program
  - Becomes a NOP on legacy processor
  - On processors that support CET the ENDBRANCH is still a NOP and is primarily used as a marker instruction by the in-order part of the processor pipeline to detect control flow violations

WAIT_FOR_ENDBRANCH State

- The CPU implements a state machine that tracks indirect *jmp* and *call*
    - When one of these instructions is seen, the state machine moves from **IDLE** to **WAIT_FOR_ENDBRANCH** state
    - In **WAIT_FOR_ENDBRANCH** state the next instruction in the program stream must be an **ENDBRANCH**
    - If an **ENDBRANCH** is not seen the processor causes a control protection fault else the state machine moves back to **IDLE** state

Part-2

Hardware Features Not Designed for Security

TSX: Transactional Synchronization eXtensions

INTEL TSX

Transactional Memory 101

- **Hardware TM to mass market**
  - Intel's restricted transactional memory (RTM) ✓
  - IBM's IBM Blue Gene/Q
  - AMD advanced synchronization family (ASF proposal)
- **Generally provides:**
  - Opportunistic concurrency
  - Strong atomicity: read set & write set
  - Semantic of both <u>all-or-nothing</u> and <u>before-or-after</u>
- **Real-world best-effort TM**
  - Limited read/write set
  - System events may abort an TX

Programming with RTM

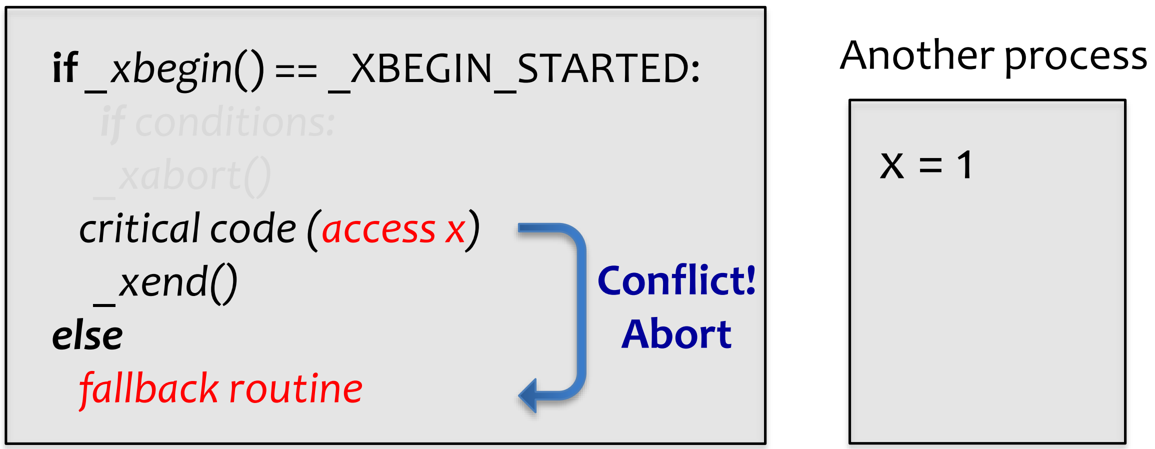
- **If transaction starts successfully**
  - Do work protected by RTM, and then try to commit
- Fallback routine to handle abort event
  - If abort, system rollback to _xbegin, return an abort code
- Manually abort inside a transaction

```
if _xbegin() == _XBEGIN_STARTED:
    if conditions:
      _xabort()
  critical code
    _xend()
else
  fallback routine
```

No conflict

Programming with RTM

- **If transaction start successfully**
  - Do work protected by RTM, and then try to commit
- **Fallback routine to handle abort event**
  - If abort, system rollback to _xbegin, return an abort code
- Manually abort inside a transaction

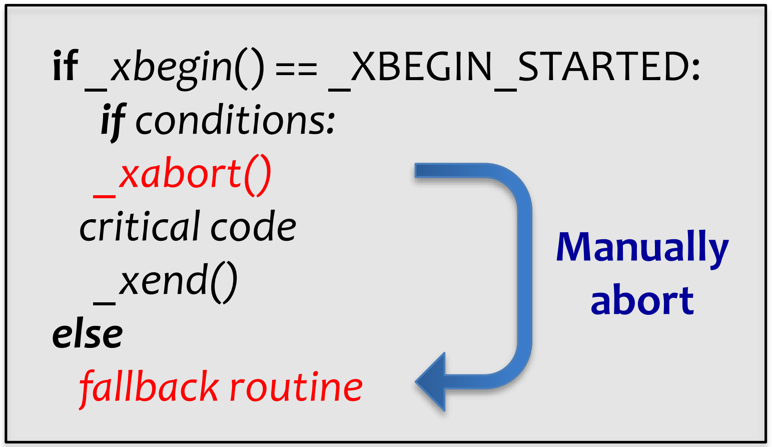


```
if _xbegin() == _XBEGIN_STARTED:
    if conditions:
      _xabort()
  critical code (access x)
    _xend()
else
  fallback routine
```

```
x = 1
```

Programming with RTM

- **If transaction start successfully**
  - Do work protected by RTM, and then try to commit
- **Fallback routine to handle abort event**
  - If abort, system rollback to _xbegin, return an abort code
- **Manually abort inside a transaction**



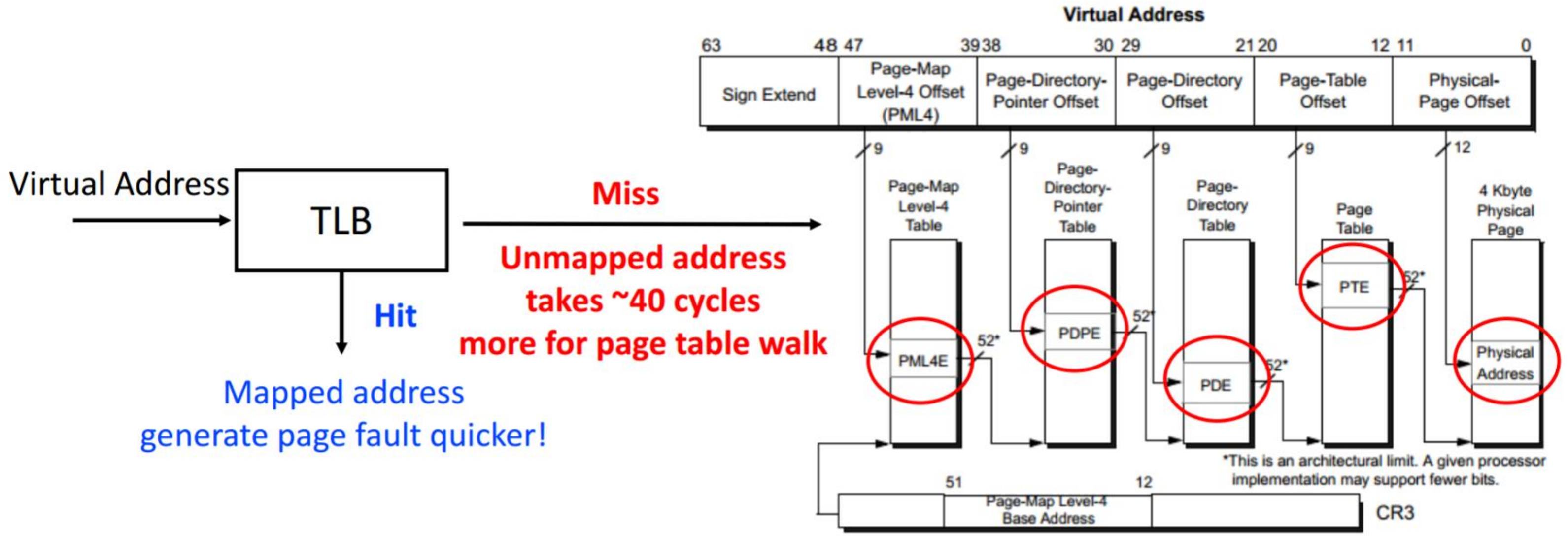
```
if _xbegin() == _XBEGIN_STARTED:
    if conditions:
   _xabort()
  critical code
   _xend()
else
  fallback routine
```

Using HTM for Data Protection

- Idea: leverage the strong atomicity guarantee provided by HTM to defeat illegal concurrent accesses to the memory space that contains sensitive data
  - Each private-key computation is performed as an atomic transaction
- During the transaction
  - Private key is first decrypted into plaintext
  - Use to decrypt or sign messages
  - If the transaction is interrupted, the abort handler clears all updated but uncommitted data in the transaction
  - Before committing the computation result, all sensitive data are carefully cleared

Using TSX for Attack KASLR

- TLB Timing Side Channel



Unmapped address takes ~40 cycles more for page table walk

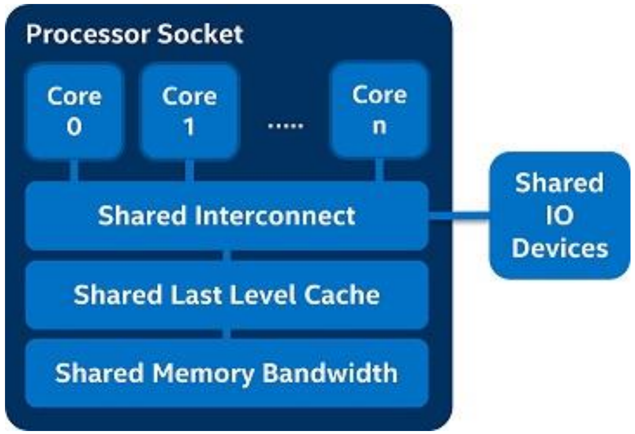
Mapped address generate page fault quicker!

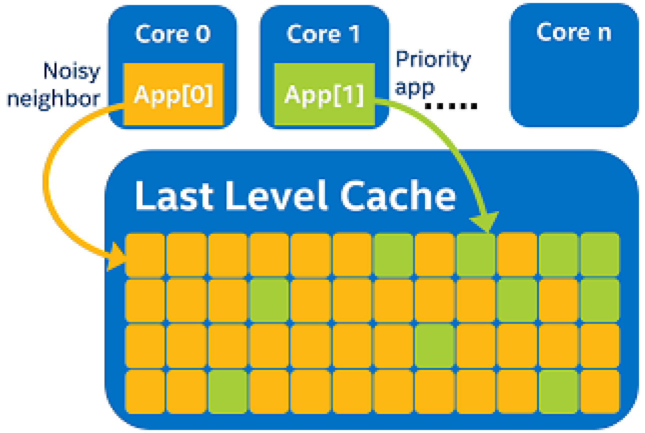
CAT: Cache Allocation Technology

INTEL CAT

The "Noisy Neighbor" Problem

- "noisy neighbor" on core zero over-utilizes shared resources in the platform, causing performance inversion
- Though the priority app on core one is higher priority, it runs slower than expected



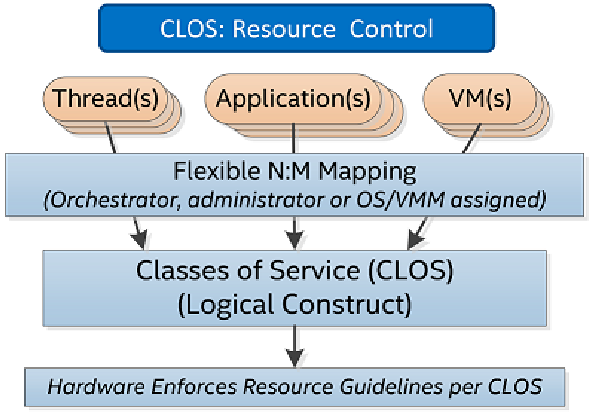


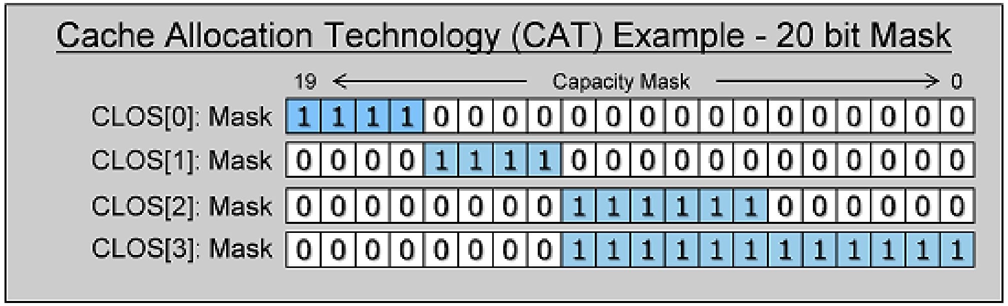
Software Controlled Cache Allocation

- The basic mechanisms of CAT include:
  - The ability to enumerate the CAT capability and the associated LLC allocation support via CPUID
  - Interfaces for the OS/hypervisor to group applications into classes of service (CLOS) and indicate the amount of last-level cache available to each CLOS
  - These interfaces are based on MSRs
    - Model-Specific Registers

Class of Service (CLOS)

- CLOS acts as a resource control tag into which a thread / app / VM / container can be grouped
- CLOS in turn has associated resource capacity bitmasks (CBMs) indicating how much of the cache can be used by a given CLOS


CLOS: Resource Control
Thread(s)
Application(s)
VM(s)
Flexible N:M Mapping
(Orchestrator, administrator or OS/VMM assigned)
Classes of Service (CLOS)
(Logical Construct)
Hardware Enforces Resource Guidelines per CLOS

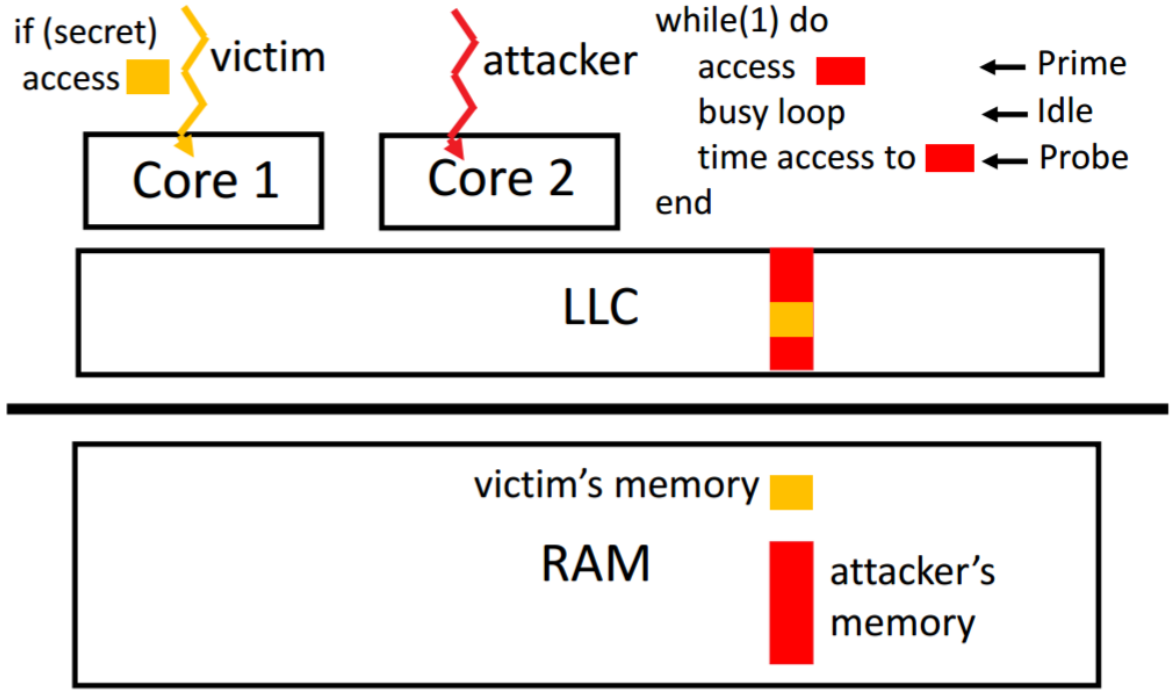


Cache Allocation Technology (CAT) Example - 20 bit Mask

19 ← Capacity Mask → 0

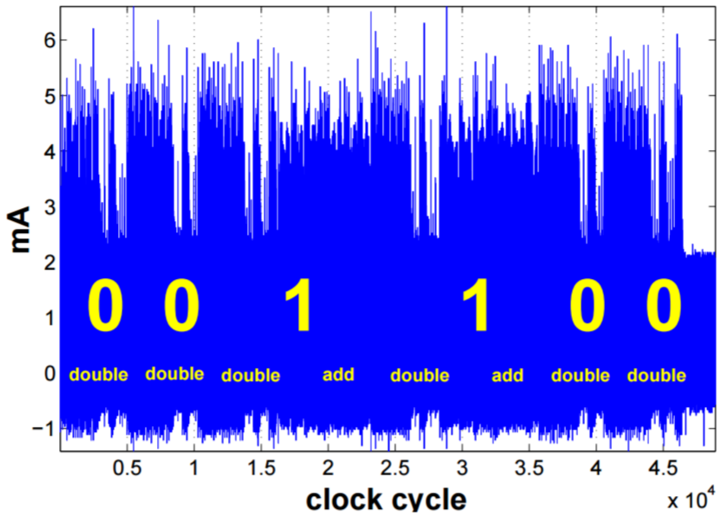
CLOS[0]: Mask	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
CLOS[1]: Mask	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
CLOS[2]: Mask	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0
CLOS[3]: Mask	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1

The PRIME+PROBE Attack



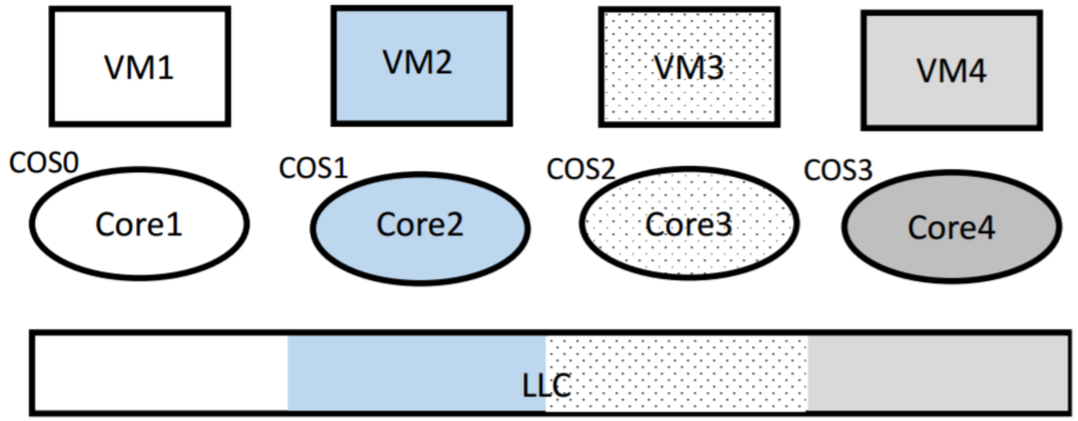
Cache Side-channel Attack

- An attacker uses the side-channel information from one measurement directly to determine (parts of) the secret key
- A simple analysis attack exploits the relationship between the executed operations and the side-channel information



Use CAT to Mitigate Cache Side-channel Attack

- Partitioning of the LLC between cores using CAT



PMU: Performance Monitor Unit

PMU

Monitor Control Flow by Existing PMU

- PEBS: Precise Performance Counter
  - Save samples in memory region for batching
  - Atomic-freeze: record exact IP address precisely
- BTS: Branch Trace Store
  - Capture all control transfer events
  - Also save exact IP in memory region
- LBR: Last Branch Record
  - Save samples in register stack, only 16 pairs
- Event Filtering
  - E.g. "do not capture near return branches"
  - Only available in LBR, not BTS
- Conditional Counting
  - E.g. "only counting when at user mode"

Completeness
Accuracy
Efficiency

Trace Samples

```
0xff01cb -> 0xff01bb
0xff01c0 -> 0xff01fb
          …
0xff01cb -> 0xff01bb
0xff01c0 -> 0xff01fb
```

Motivation: Code Injection Attack

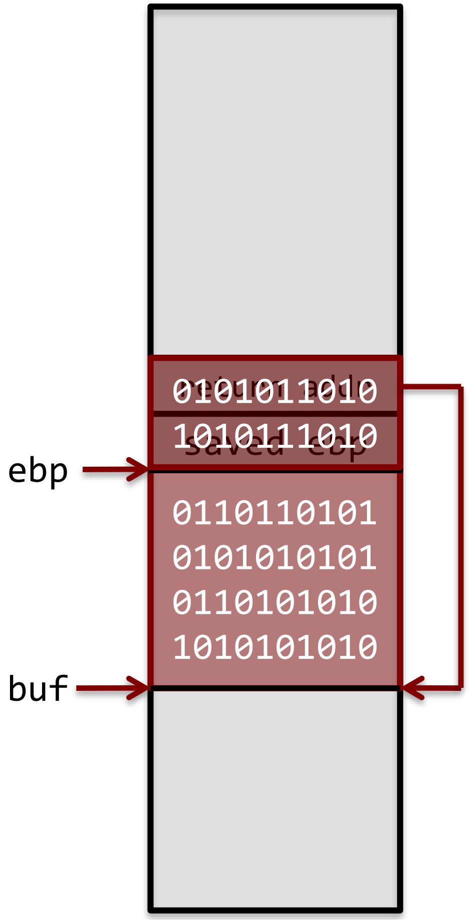
- A Typical Buffer Overflow Attack

```
void function(char *str) {
    char buf[16];
    strcpy(buf,str);
}
```

  - Inject malicious code in buffer
  - Overwrite return address to buffer
  - Once return, the malicious code runs

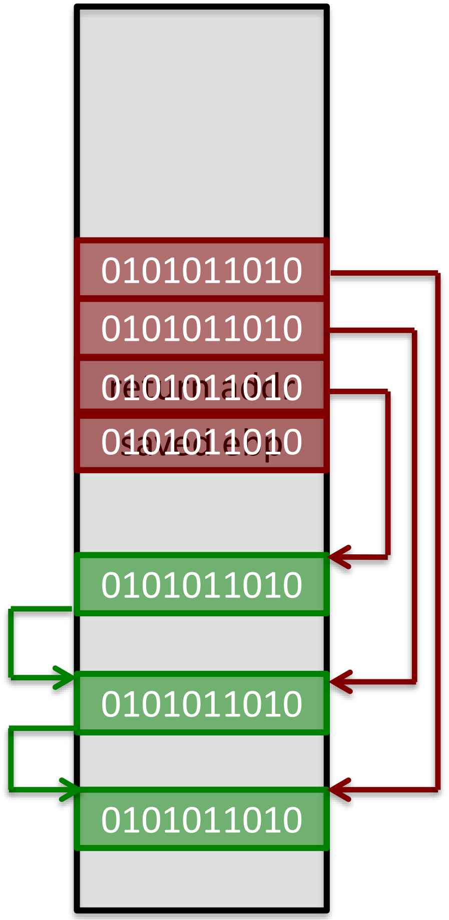
- Solutions
  - StackGuard[Cowan'98], FormatGuard[Cowan'98]
  - Make data section non-executable

- New Attacks: Code-reuse Attack
  - Return-to-libc & return-oriented programming



Motivation: Code Reuse Attack

- Return-oriented Programming
  - Find code gadgets in existed code base
    - Usually 1-3 instructions, ends with 'ret'
    - In libc and application, intended and unintended
  - Push address of gadgets on the stack
  - Leverage 'ret' at the end of gadget to connect each code gadgets
  - **No code injection**
- Solutions
  - Return-less kernels [Li'10]
  - Heuristic means
- New: Jump-oriented attacks [Bletsch'11]
  - Use gadget as dispatcher



Motivation: CFI

- General Solutions to Enforce CFI
  - Some need binary re-writing or source re-compiling
  - Some need application/OS/Hardware re-designing
  - Some have large overhead (3.6X for LIFT and 37X for TaintCheck)
- Challenges
  - Non-intrusive general attack detection
  - Apply to existing applications on commodity hardware
- Observations
  - **Performance counters can be leveraged to monitor CFI**

The Main Idea

- Leverage PMU for CFI Monitoring
  - Using already existing hardware
  - No need to modify software
- Two Phases
  - Offline phase: Get all the legal targets for each branch source
  - Online phase: Monitor all branches and detect malicious ones

Branch Types

- Direct Branches
  - Direct call
  - Direct jump
- Indirect Branches
  - Return
  - Indirect call
  - Indirect jump

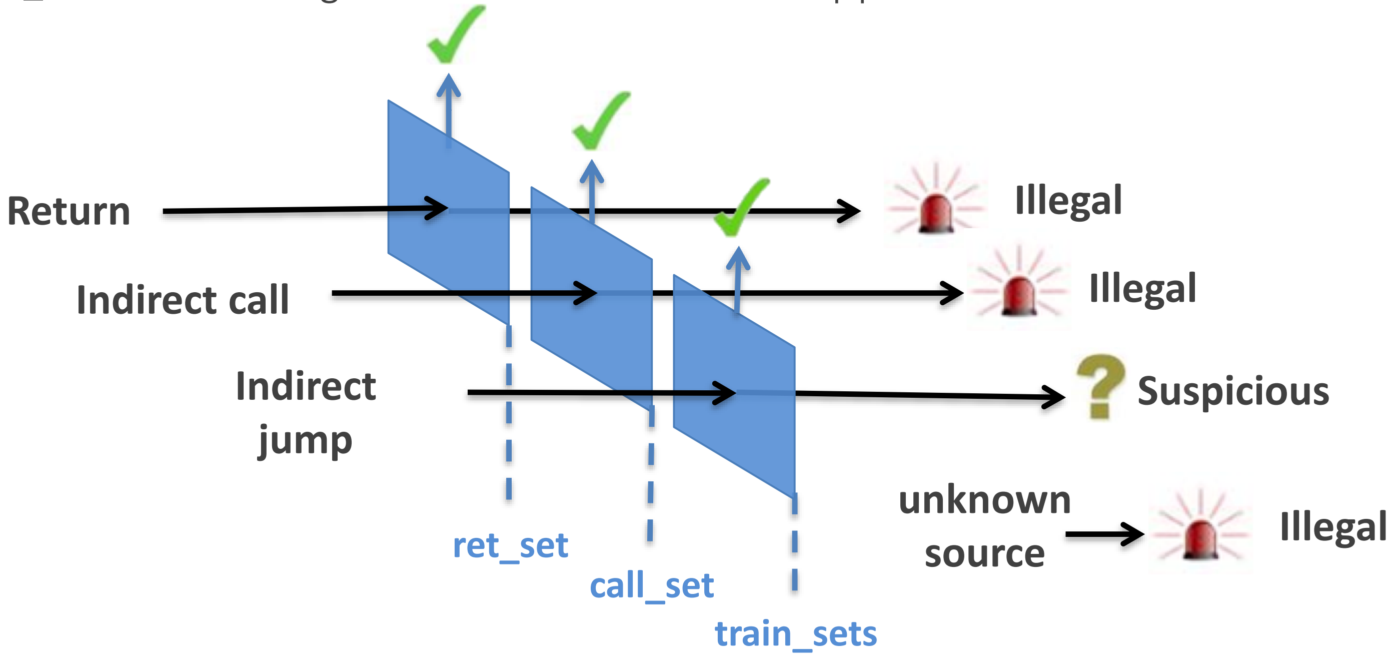
In Apache and its libraries

Types	In Binary	Run-time
Direct call	16.8%	14.5%
Direct jump	74.3%	0.8%
Return	6.3%	16.3%
Indirect call	2.1%	0.2%
Indirect jump	0.5%	68.3%

- has 1 target: 94.7%
- <= 2 targets: 99.3%
- >10 targets: 0.1%

Target Address Sets

- Target Sets for Indirect Branches
  - ret_set: all the addresses next to a call
  - call_set: all the first addresses of a function
  - train_sets: all the target addresses that once happened



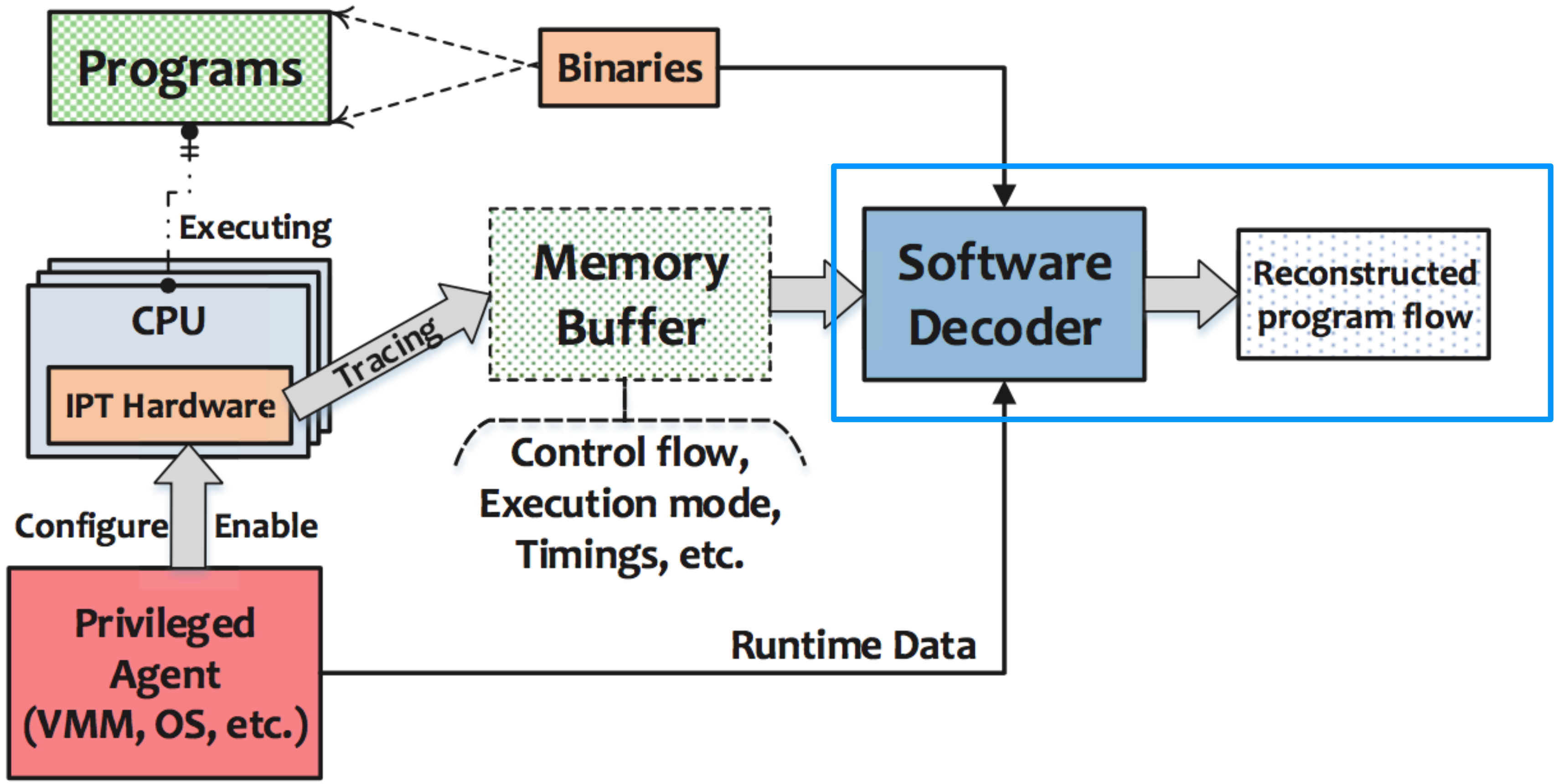
Intel PT: Intel Processor Tracing

INTEL PT

Intel Processor Tracing (IPT)

- Privileged agent configures IPT per core
  - Define memory location and size for tracing
  - 3 filtering mechanisms: CPL, CR3, IP range
- Efficiently captures various information
  - Control flow, timing, mode change, etc.

Background: IPT Overview



Challenges: Fast Trace vs. Slow Decode

- IPT uses aggressive compression
  - Unconditional direct branches are not logged at all
  - Conditional branches are compressed to a single bit
  - Each indirect branch is traced as one target address
  - Result in average <1 bit per retired instruction

Challenges: Fast Trace vs. Slow Decode

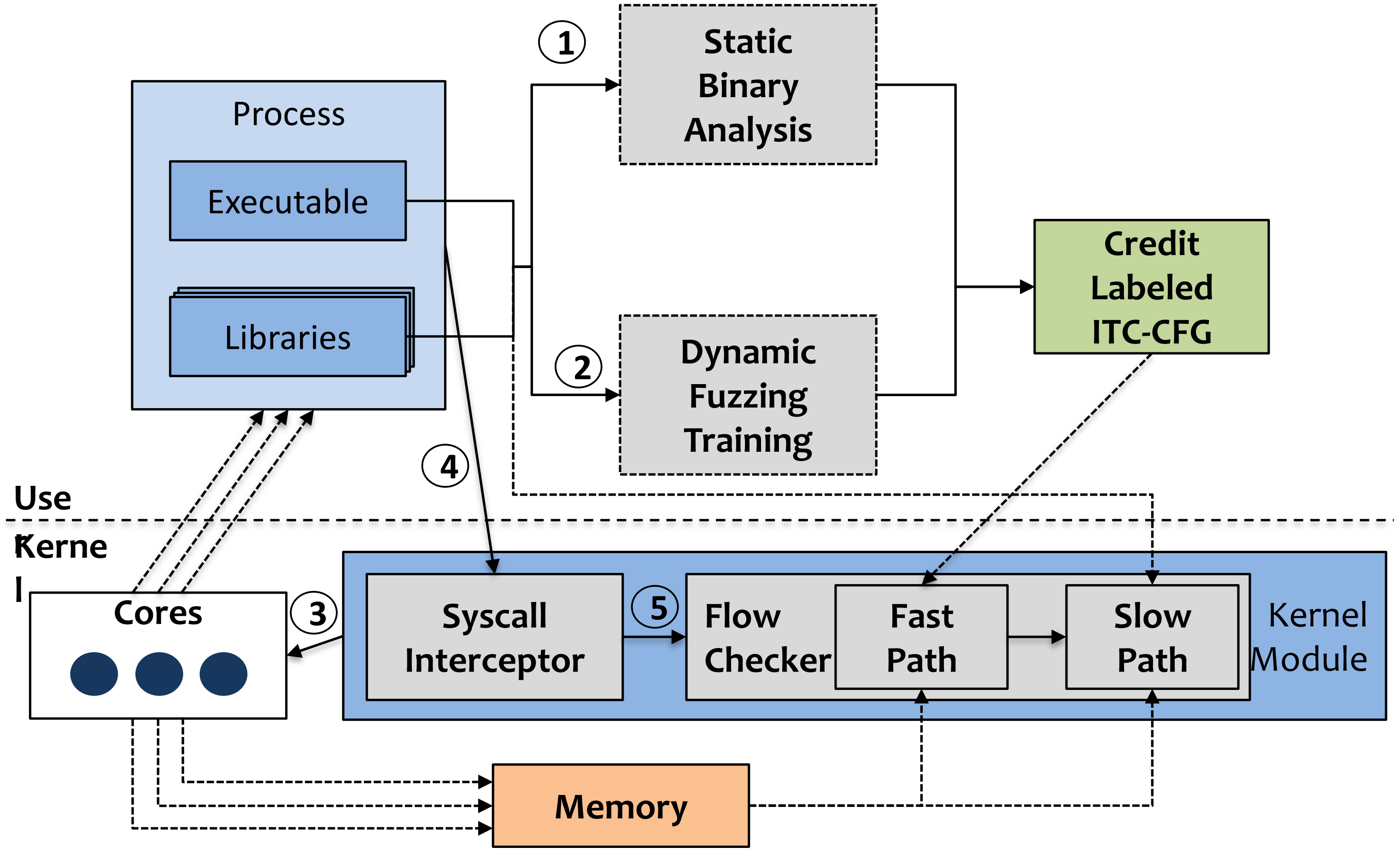
- Performance overhead is shifted from tracing to decoding
  - Decoding is **several orders of magnitude slower** than tracing

	Precise	Tracing	Decoding	Filtering
BTS	Full	Slow (50X)	Fast	None
LBR	Low	Very Fast (< 1%)	Fast	CPL, CoFI
IPT	Full	**Fast (3%)**	**Slow (200X)**	CPL, CR3, IP

FlowGuard

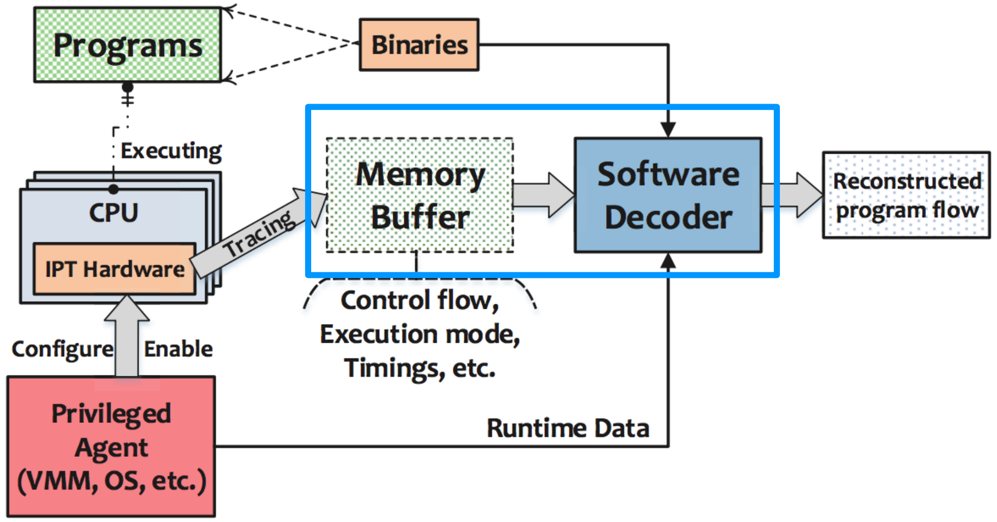
- FlowGuard: transparent, efficient and precise CFI
  - **Transparent**: no source code needed, no hardware change
  - **Precise**: enforce fine-grained CFI with dynamic information
  - **Efficient**: reconstruct CFG and separate fast and slow paths
- Evaluation results
  - Apply FlowGuard to real machine with server workloads
  - Prevent a various of real code reuse attacks
  - Less than 8% performance overhead for normal use cases

FlowGuard Architecture



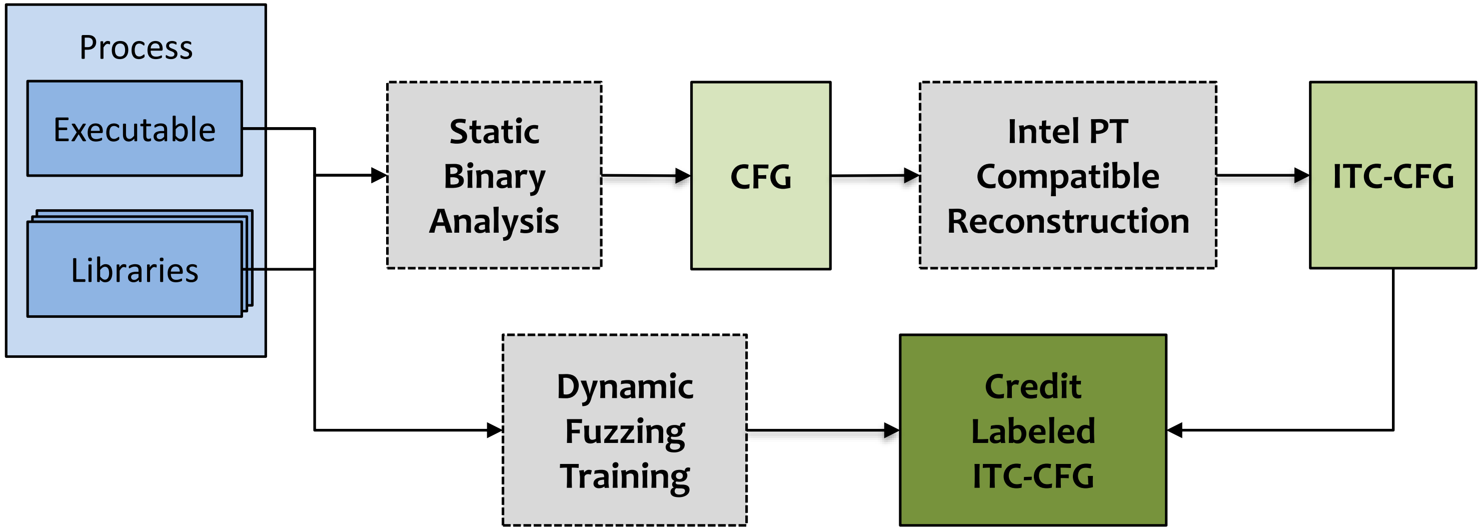
Using IPT for CFI

- Privileged agent configures IPT per core
  - Define memory location and size for tracing
- Efficiently captures various information
  - Control flow, timing, mode change, etc.



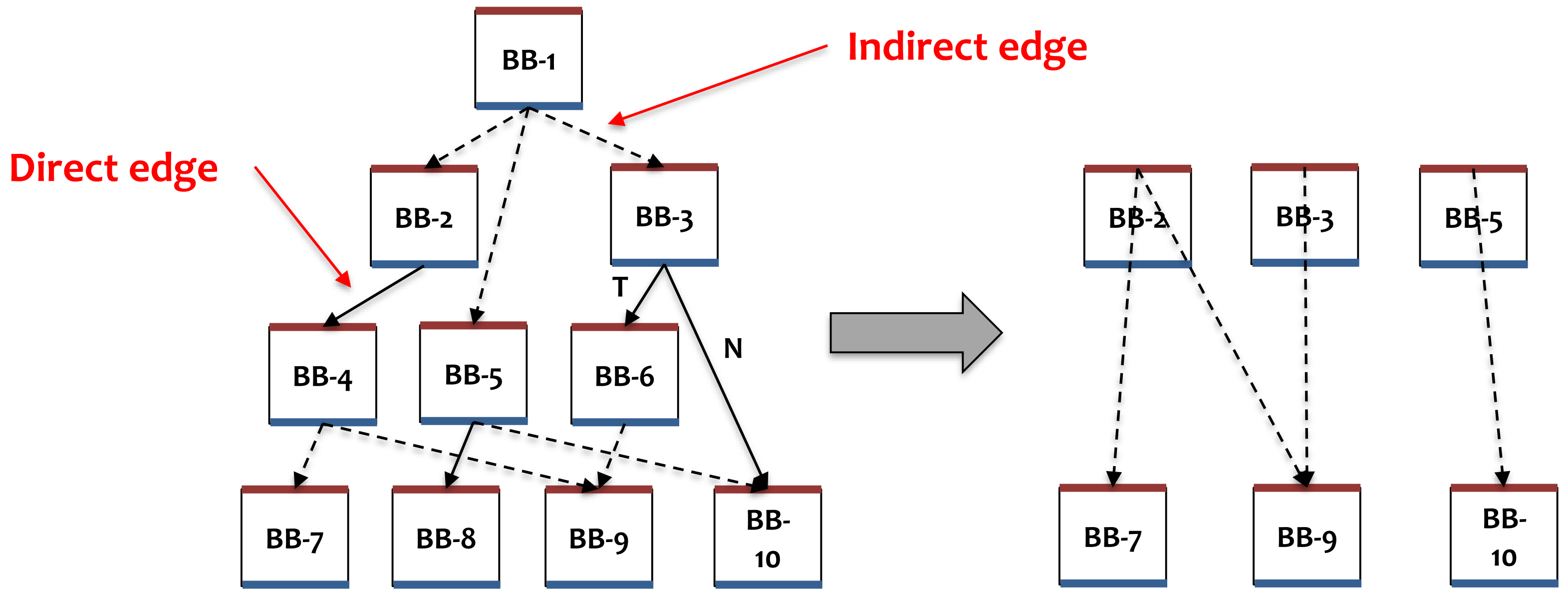
CFG Generation Overview

- Conservatively generate CFG with static binary analysis
- Reconstruct the CFG to be IPT compatible (ITC-CFG)
- Label the edges of ITC-CFG with credits using dynamic fuzzing training



ITC-CFG Construction Example

- IPT traced data can be directly matched on the ITC-CFG



(a). Traditional CFG

(b). ITC-CFG