Institute of Parallel And Distributed Systems (IPADS, SJTU)

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Hardware-based System Security

Hardware Features, Usage and Scenarios

Yubin Xia
http://ipads.se.sjtu.edu.cn

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

Outline

1	2	3
Features designed for Security	**Isolated Execution Environment**	**Features NOT for Security**
• SMEP/SMAP	• Virtualization	• Intel TSX
• Intel MPX/MPK	• Intel SGX	• Intel CAT
• ARM AP	• AMD SME/SEV	• PMU
• Intel CET	• ARM TrustZone	• Intel PT

Part-1

Hardware Features Designed for Security

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

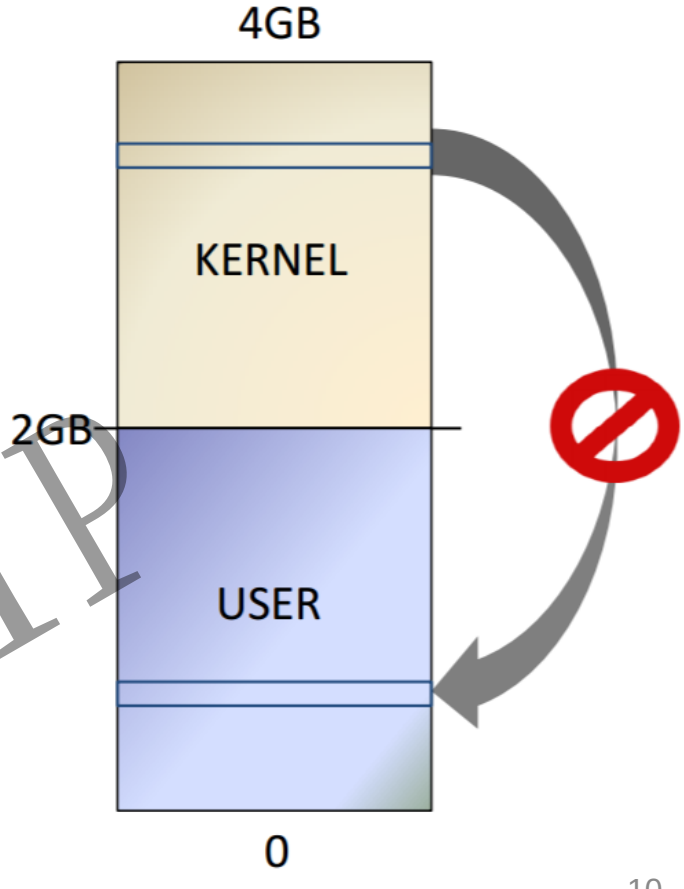
- SMAP: Supervisor Mode Access Prevention
  - Allows pages to be protected from supervisor-mode data accesses
  - If SMAP = 1, OS cannot access data at linear addresses of application

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

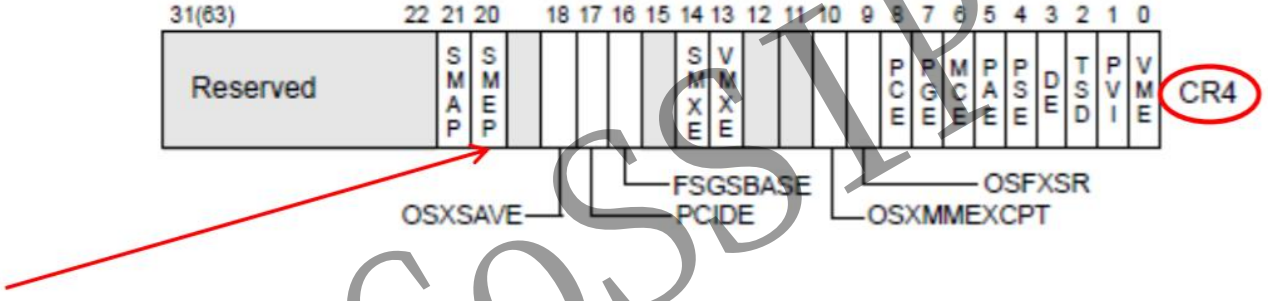


SMEP CPU Support

- 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

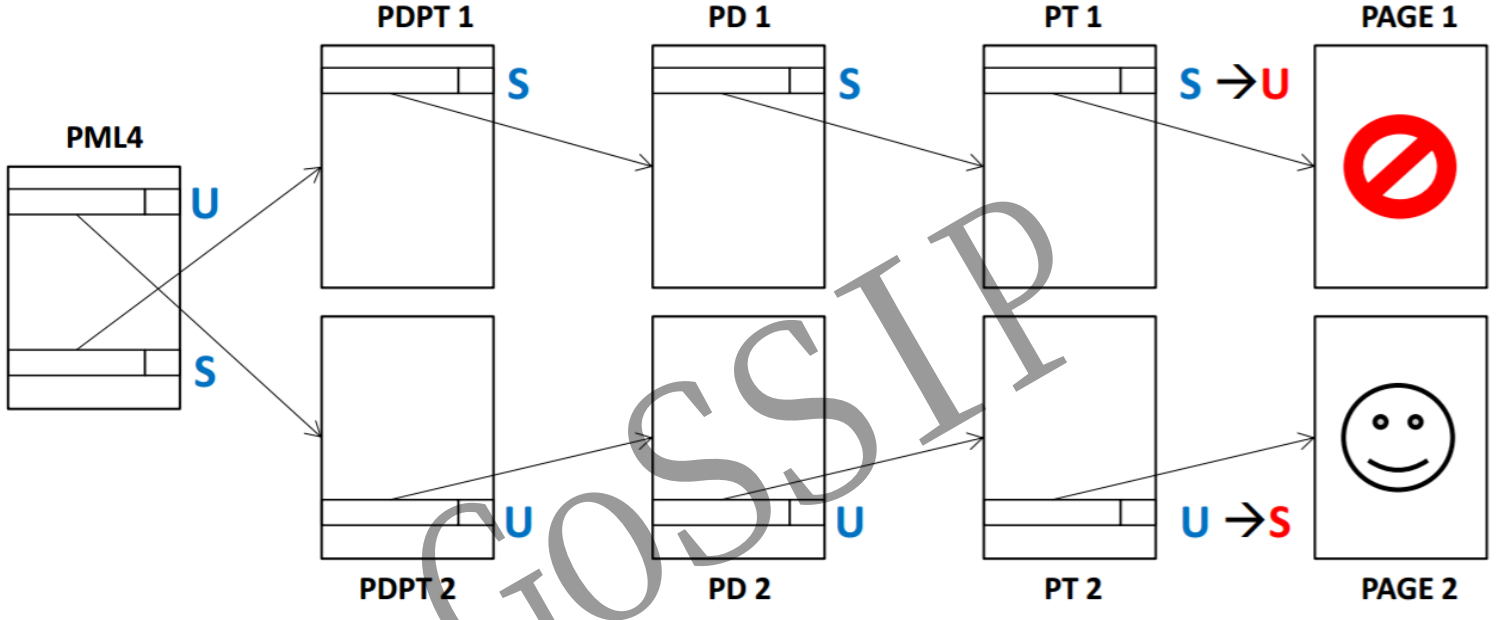
How to Bypass SMAP/SMEP?

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



How to Bypass SMAP/SMEP?

- 2. Flipping the **U/S** bit



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

Bounds Error

```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;
}
```

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

```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;
}
```

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 two 64-bit addresses 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

Isolated Execution Environment

HARDWARE-ASSISTED VIRTUALIZATION

VT-x Modes

VMX root operation:

Full privileged, intended for Virtual Machine Monitor

VMX non-root operation:

Not fully privileged, intended for guest software

Both forms of operation support all four privilege levels from 0 to 3

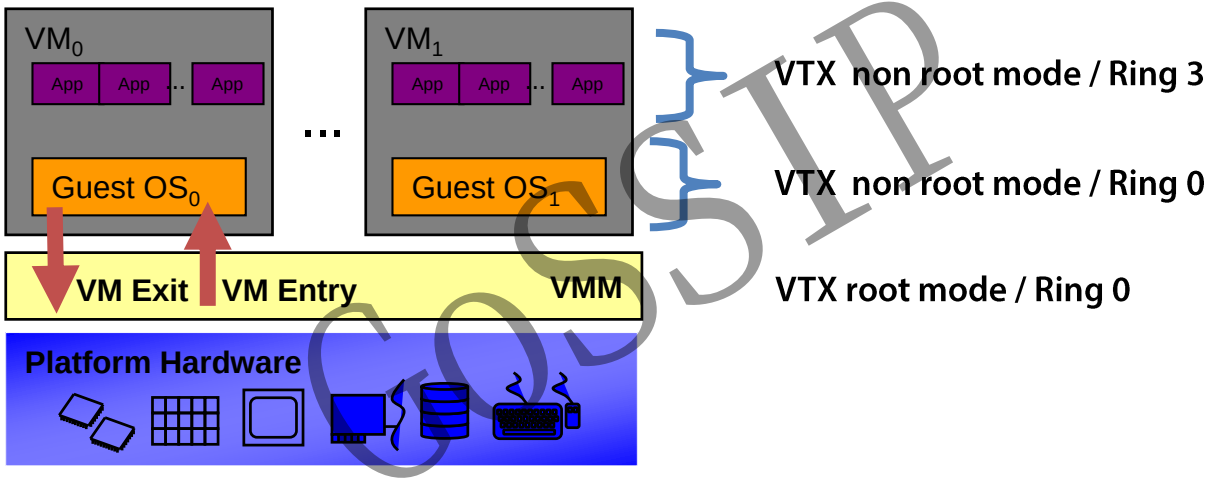
VT-x transition mechanisms

VM exit

From VMX non-root operation mode to VMX root operation mode

VM entry

from VMX root operation mode to VMX non-root operation mode



Virtual Machine Control Structure (VMCS)

Data structure that manages VM entries and VM exits

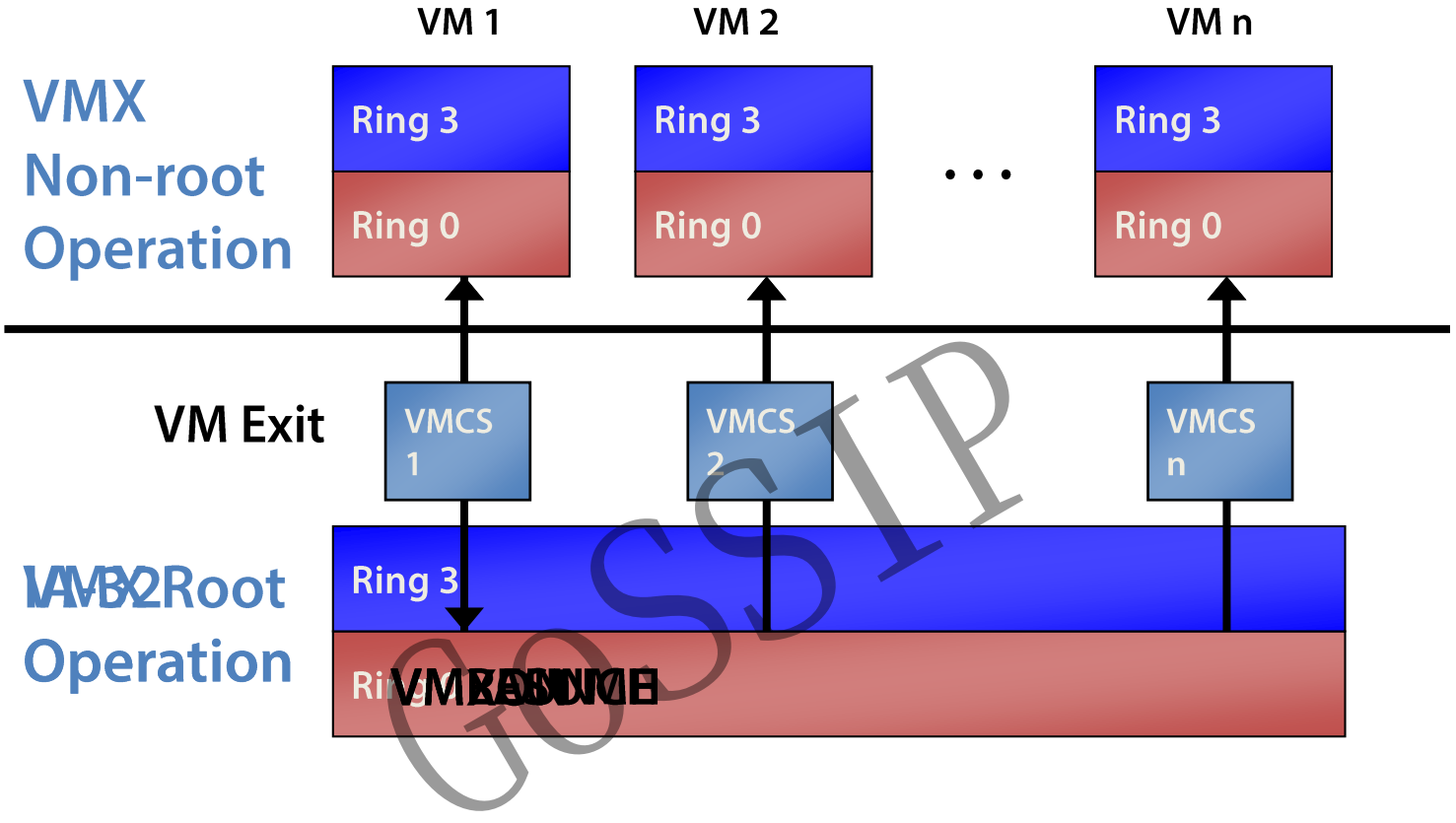
VMCS is logically divided into:

- Guest-state area
- Host-state area.
- VM-execution control fields
- VM-exit control fields
- VM-entry control fields
- VM-exit information fields

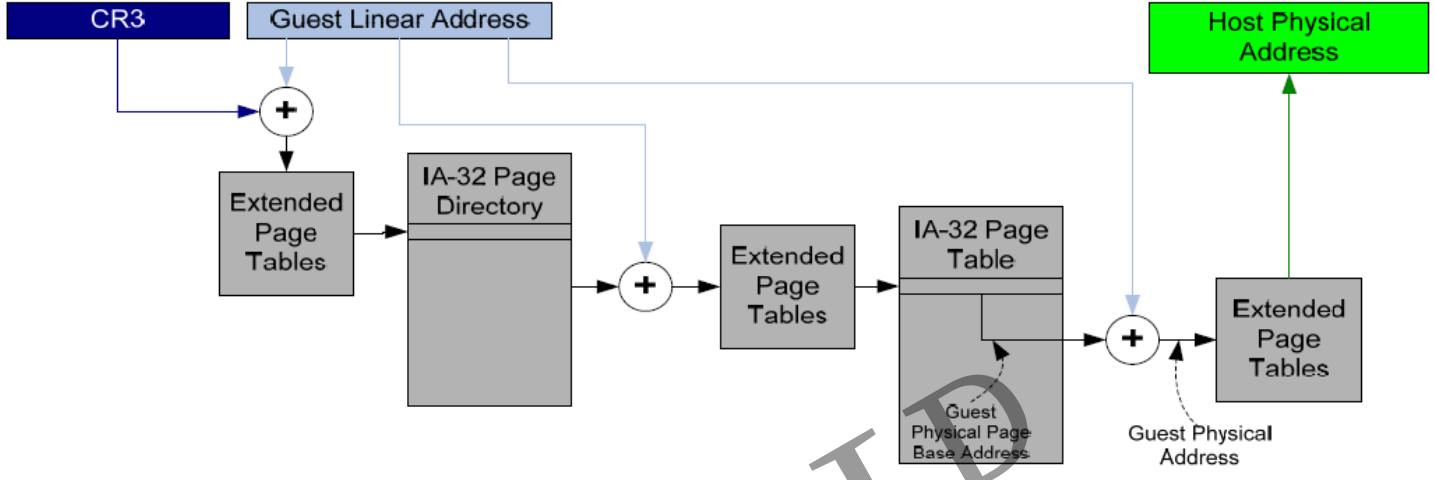
VM entries load processor state from the guest-state area

VM exits save processor state to the guest-state area and the exit reason, and then load processor state from the host-state area

VT-x Operations



VT-x extension: Extended Page Table (EPT)

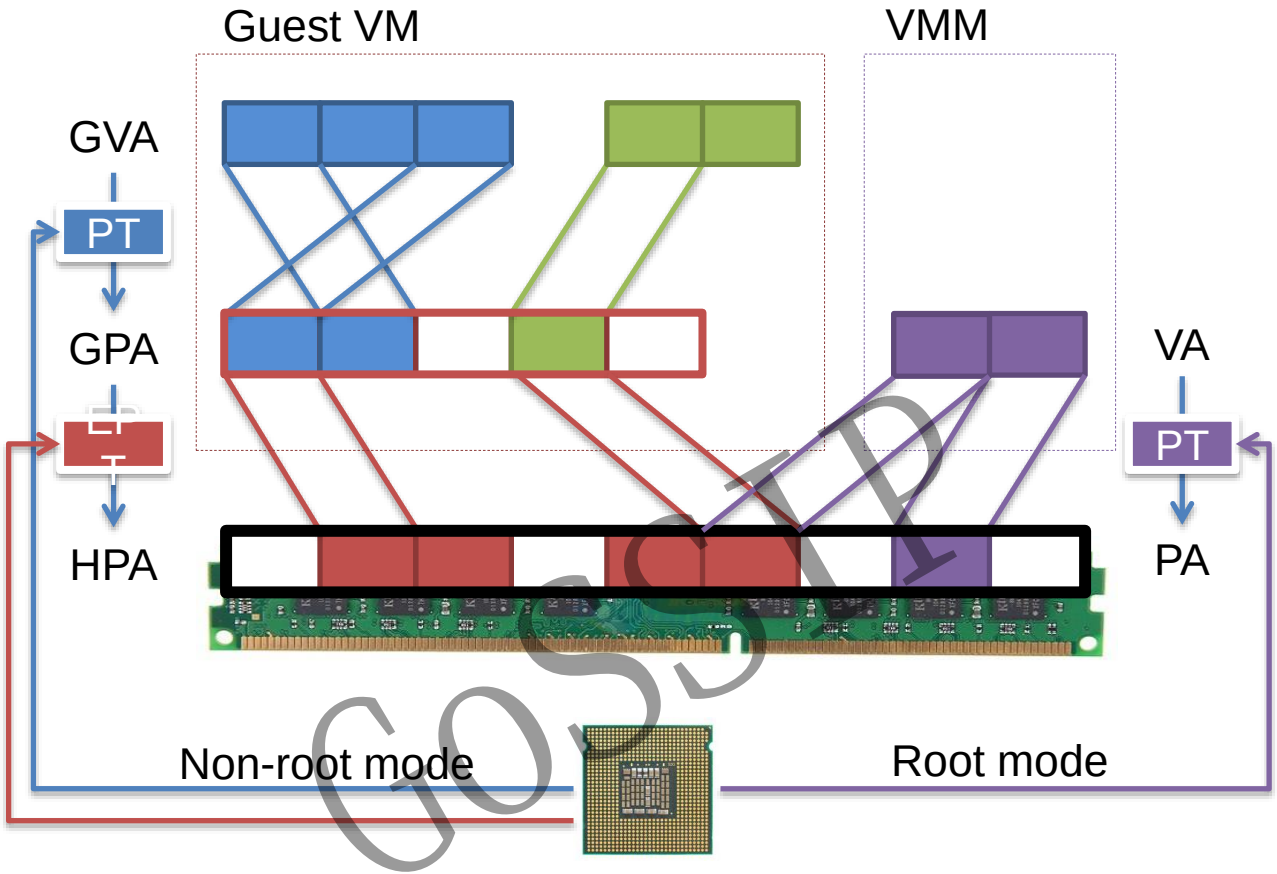


All guest-physical addresses go through extended page tables

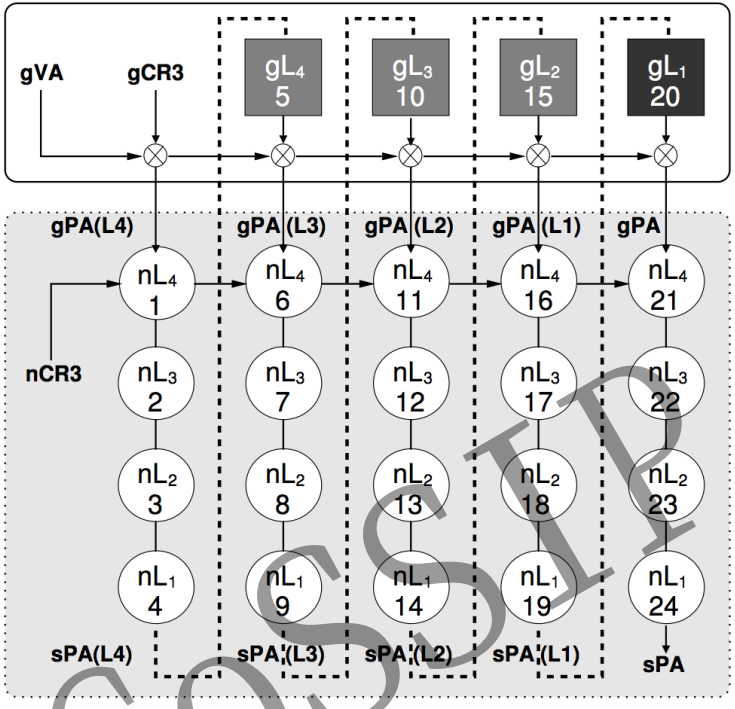
Reduces the frequency of VM exits to VMM

The net effect of both implementations (EPT or NPT) is to allow the guest OS to own and manage its own page table, and not force the host to get involved

Extended Page Table (EPT)



EPT Increases Memory Access



sPA == HPA
nCR3 == hCR3
nL_1 == hL_1

One memory access from the guest VM may lead up to ***20 memory accesses***!

VT-x New instructions

VMXON and VMXOFF
- To enter and exit VMX-root mode.

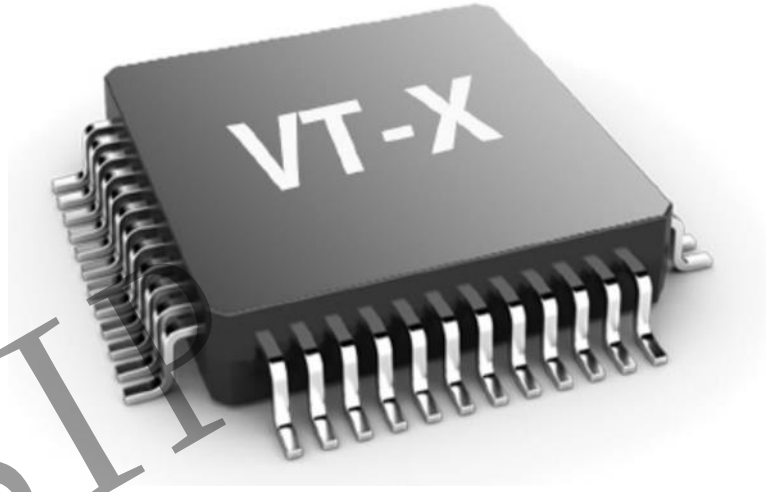
VMLAUNCH: Used on initial transition from VMM to Guest
- Enters VMX non-root operation mode

VMRESUME: Used on subsequent entries
- Enters VMX non-root operation mode
- Loads Guest state and Exit criteria from VMCS

VMEXIT
- Used on transition from Guest to VMM
- Enters VMX root operation mode
- Saves Guest state in VMCS
- Loads VMM state from VMCS

VMPTRST and VMPTRLD
- To Read and Write the VMCS pointer.

VMREAD, VMWRITE, VMCLEAR
- Read from, Write to and clear a VMCS.



Hardware assisted I/O Virtualization

- Emulated devices
  - Slow performance
  - High implementation complexity
- Para-virtualized I/O virtualization
  - Needs modification to guest OS, e.g., Linux, Windows
  - May still have suboptimal performance
- **Hardware-assisted I/O virtualization**
  - Provide efficient hardware support for I/O virtualization
  - E.g., Intel VT-d

Issues to Address

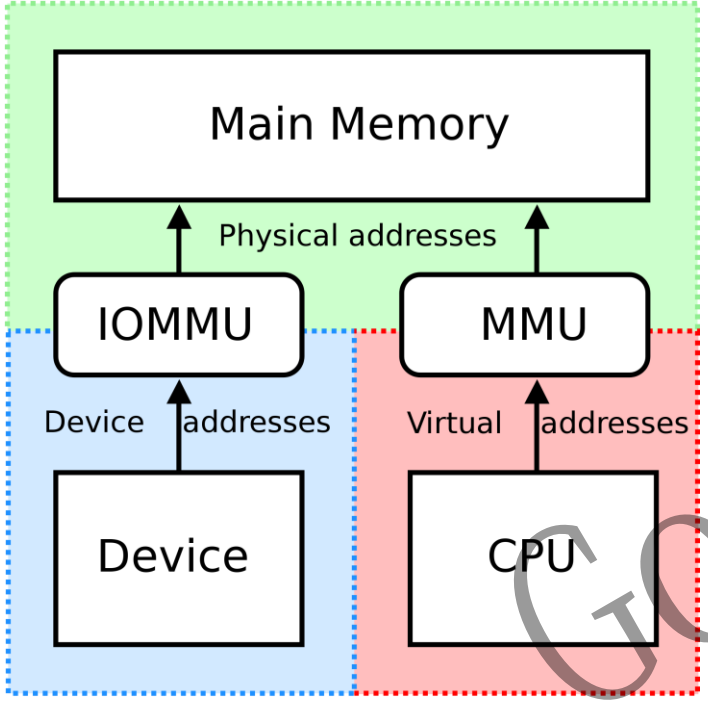
- **I/O address translation**
  - How to translate I/O address to host physical address
- **Interrupt mapping**
  - How to route an interrupt correctly to a guest VM
- **Device multiplexing**
  - How to multiplex a single hardware device among multiple VMs
- **Mostly importantly**
  - Provide strong isolation, while reduce hypervisor involvement

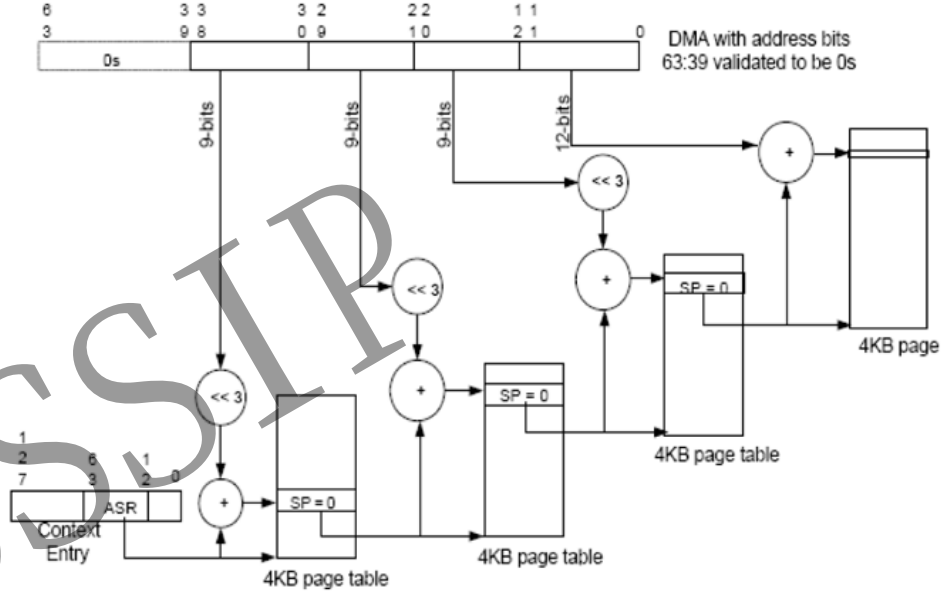
VT-d: Intel® Virtualization Technology for Directed I/O

- **Provides the capability to ensure improved isolation of I/O resources for greater reliability, security, and availability.**
- **Supports the remapping of I/O DMA transfers and device-generated interrupts.**
- **Provides flexibility to support multiple usage models that may run un-modified, special-purpose, or "virtualization aware" guest OSs.**

Address Translation Services

- VT-d architecture defines a multi-level page-table structure for DMA address translation
- The multi-level page tables are similar to IA-32 processor page-tables, enabling software to manage memory at 4 KB or larger page granularity





VT-d Feature: Interrupt Remapping

- The interrupt requests generated by I/O devices must be controlled by the VMM
- When the interrupt occurs, the VMM must present the interrupt to the guest. This is not accomplished through hardware
- The VT-d interrupt-remapping architecture addresses this problem by redefining the interrupt-message format
- Interrupt requests specify a requester-ID and interrupt-ID, and remap hardware transforming these requests to a physical interrupt

Device Multiplexiing: Single Root I/O Virtualization

- **With SR-IOV:**
  - SI's will get direct access to PCIe device functions
  - No more need for hypervisor (VI) to manage all system resources
- **PCIe devices will have multiple virtual functions (VF's)**
  - utilizable by multiple SI's
  - a single SI may also use multiple virtual functions
- **Security of I/O Streams ensured by**
  - Independency of control structures between VF's within one PCIe device
  - I/O address translation services
  - Interrupt remapping mechanisms

Single Root I/O Virtualization

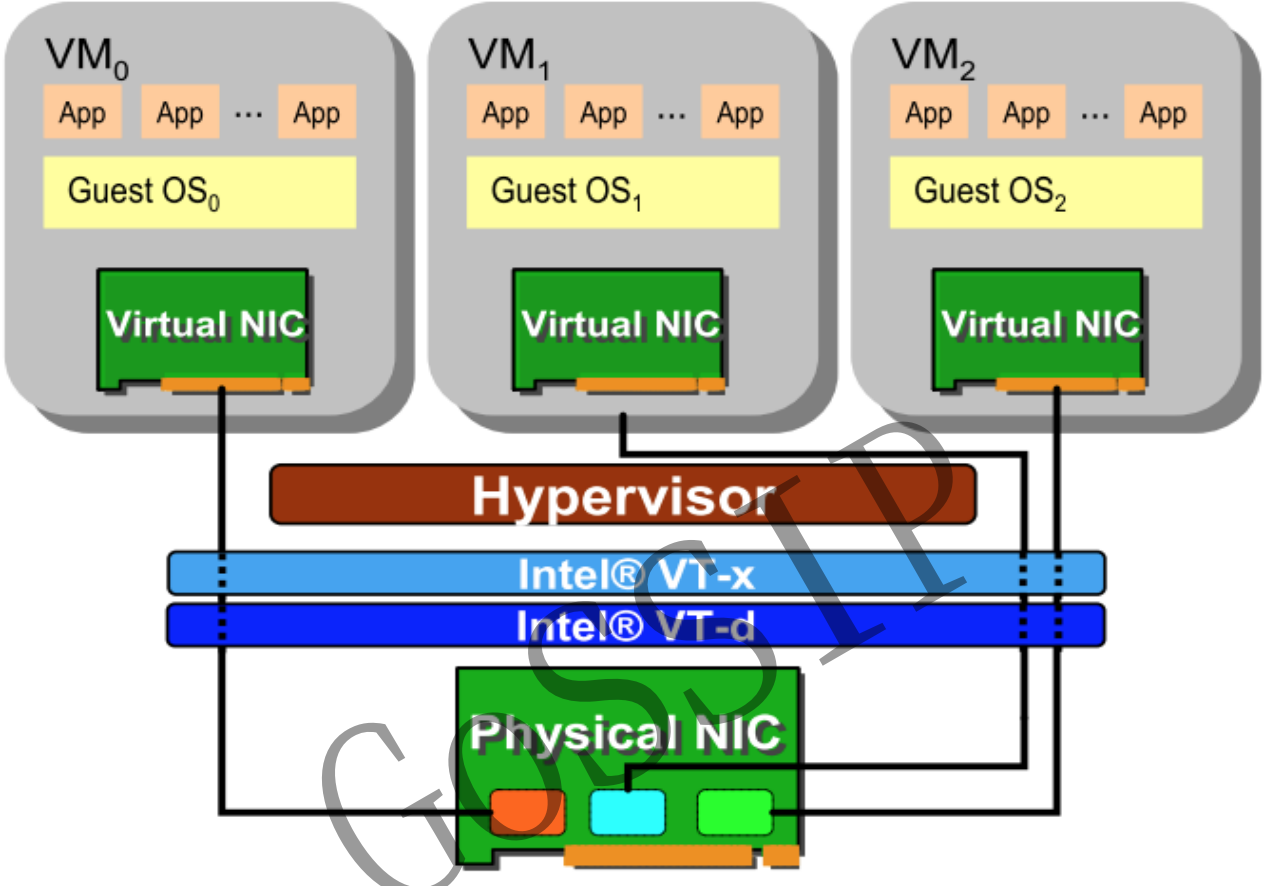
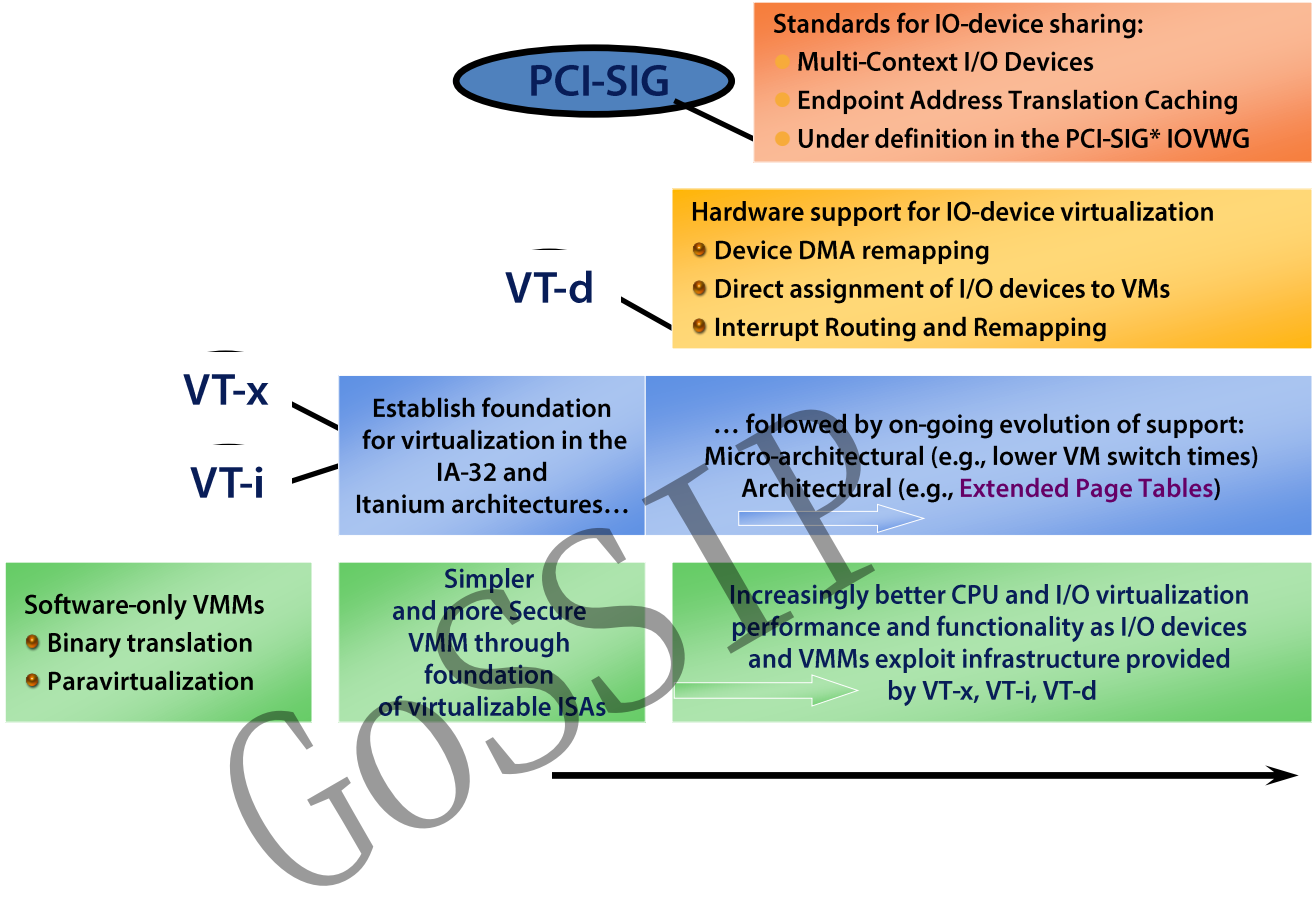
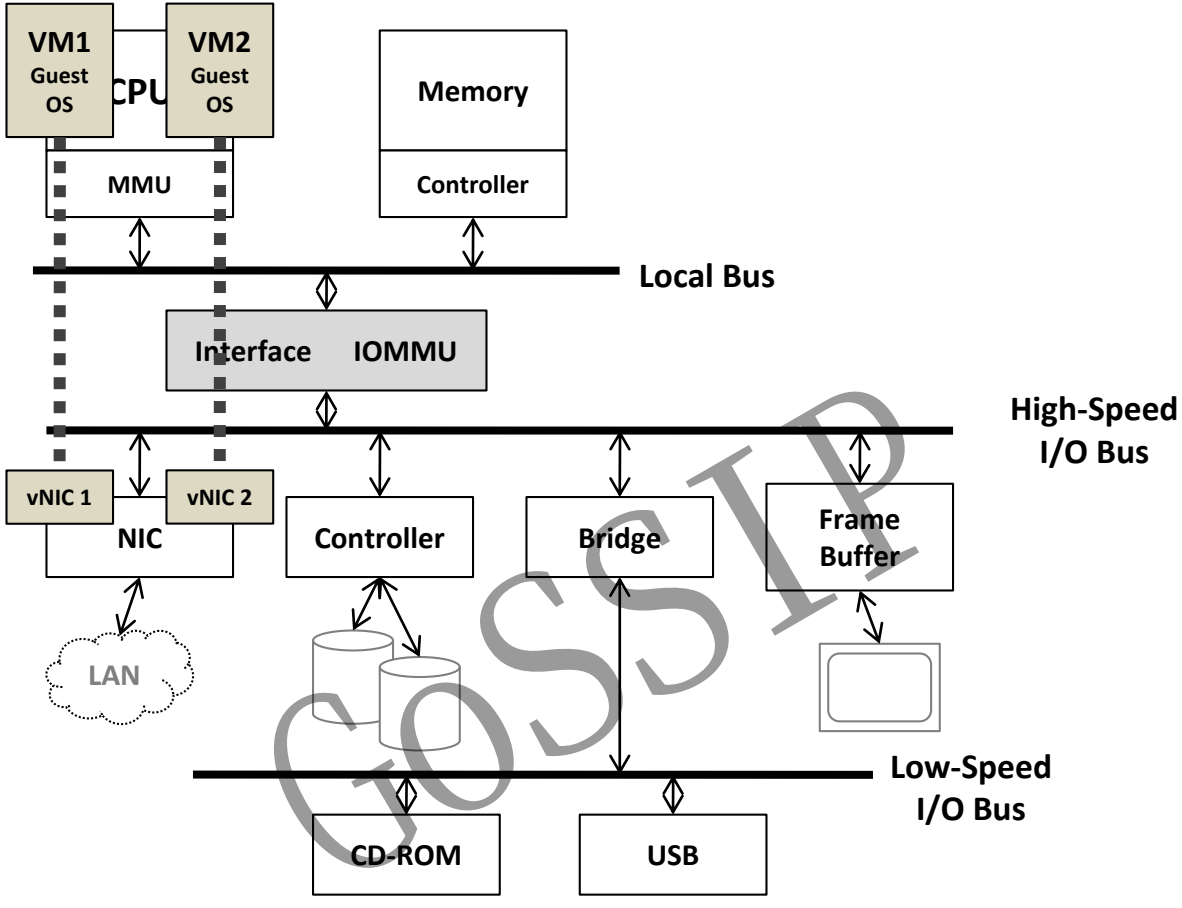


Figure 1. Natively Shared

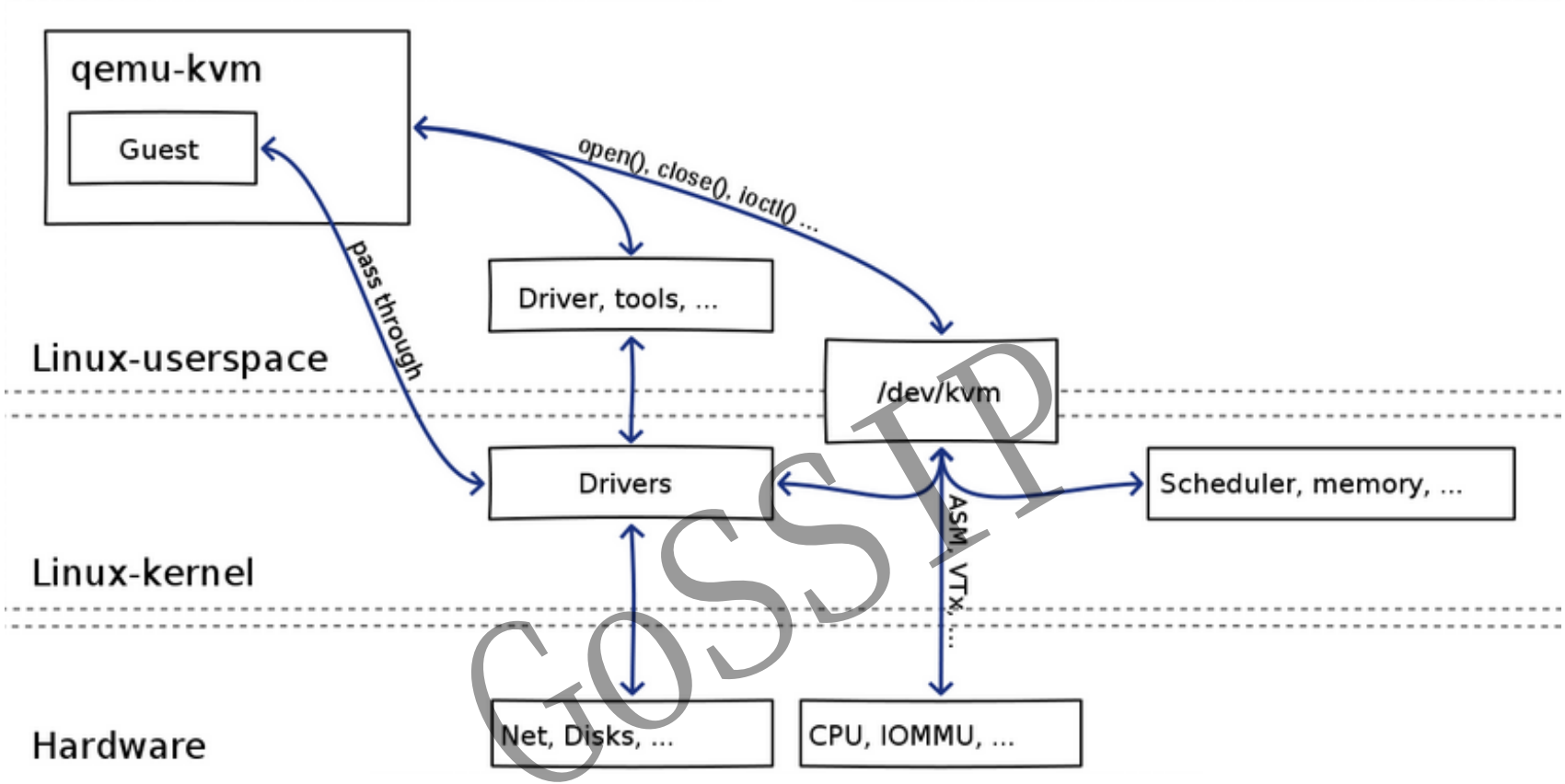
Intel Virtualization Technology Evolution



Virtualization Enabled Device



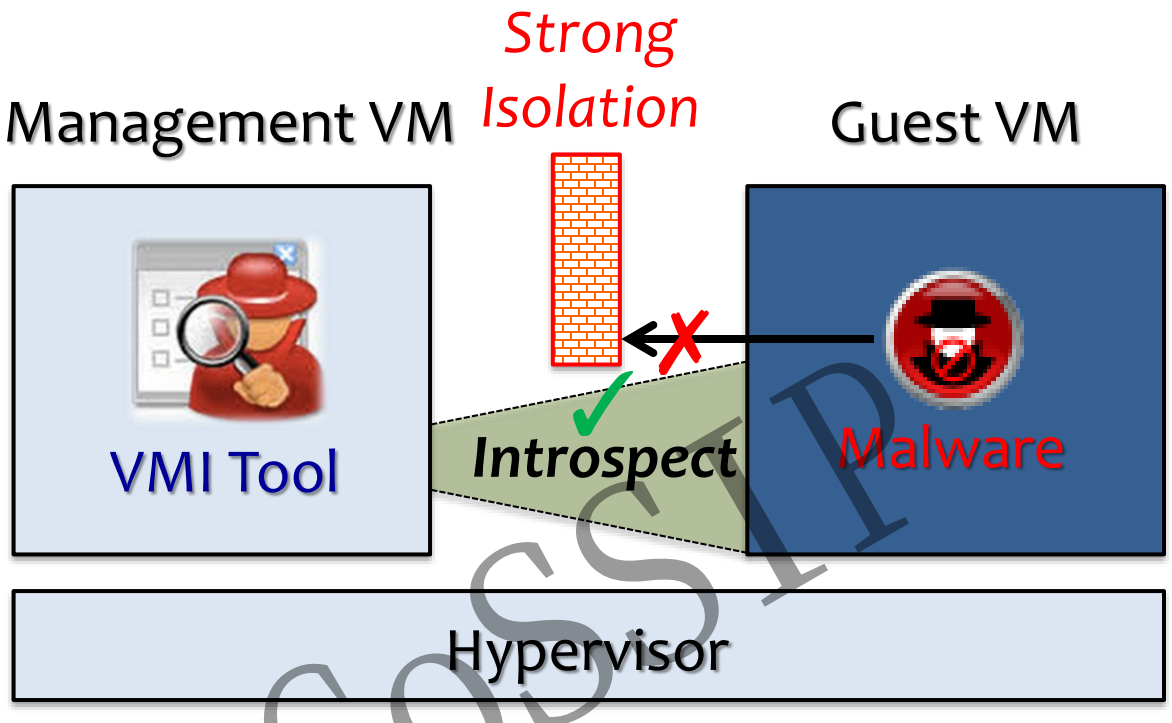
KVM in 5 Seconds



VMI: Virtual Machine Introspection

VMI

Virtual Machine Introspection(VMI)



* T. Garfinkel, M. Rosenblum *et al.*, "A virtual machine introspection based architecture for intrusion detection." in *Proc. NDSS*, 2003.

Virtual Machine Introspectiont

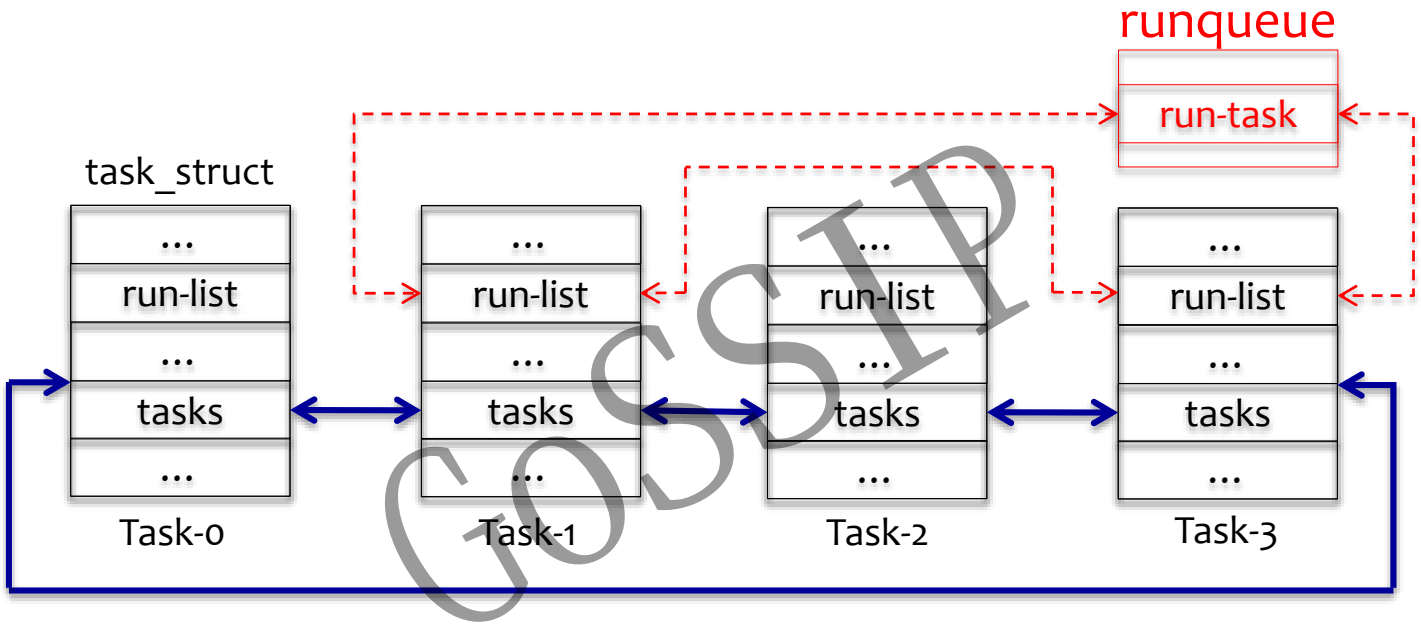
- **Benefits of VMI**
  - Non-intrusive to guest VMs
  - Better security due to strong isolation from vulnerable VMs

- **Widely used for numerous purposes**
  - Intrusion detection, malware analysis
  - Long line of research in architecture, systems and security community
  - Already integrated by VMware and other vendors

Example: Hidden Malware Detection

Task list used by `*ps aux*` to show all the tasks

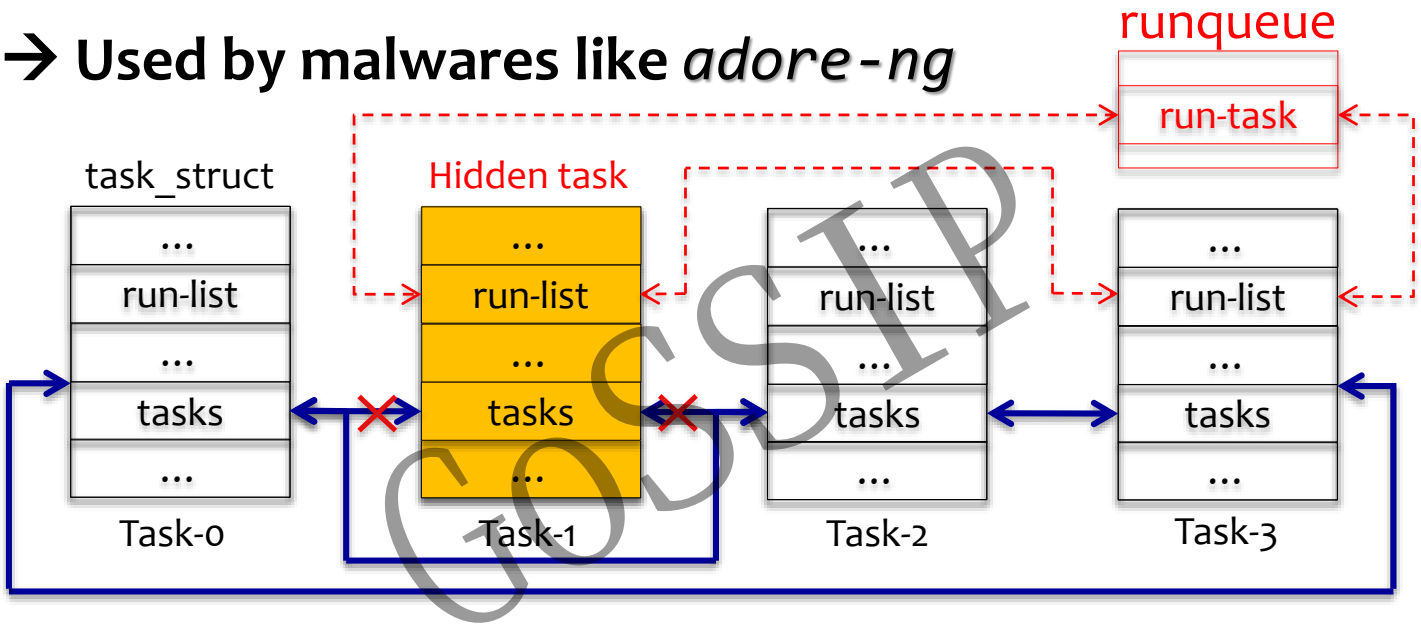
Runqueue used by scheduler to run a task



Example: Hidden Process

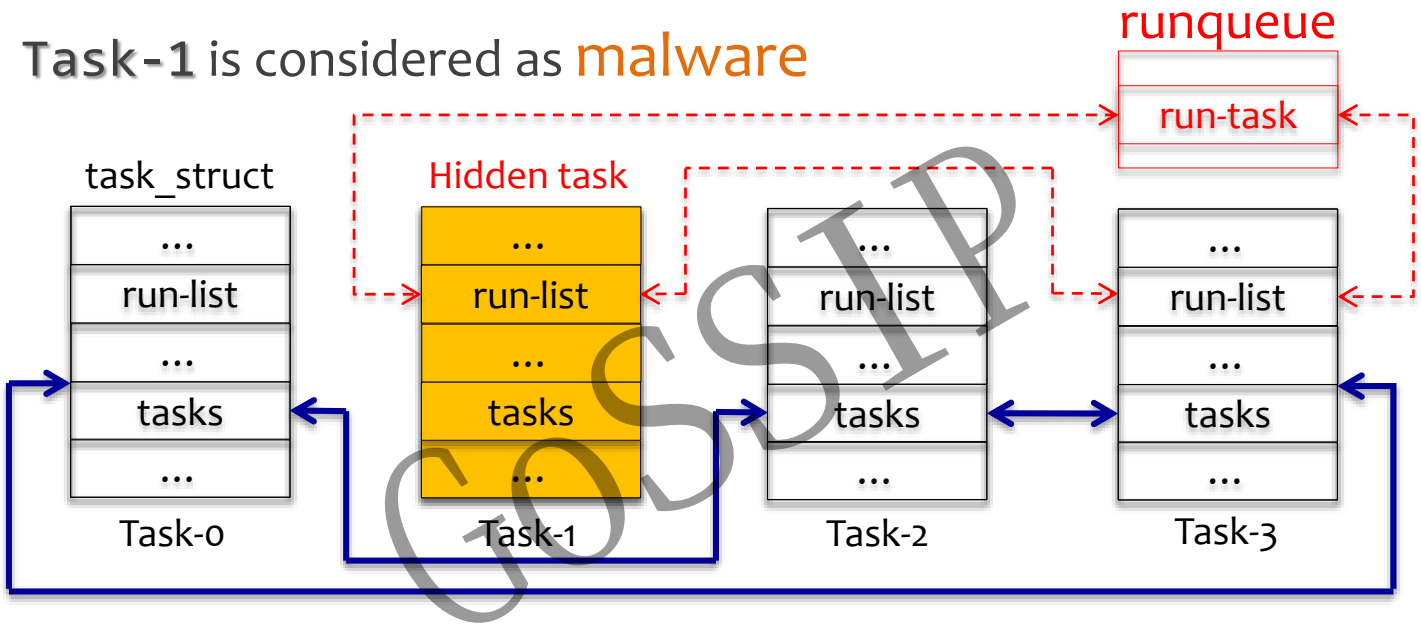
- **Malware hides itself from `*ps aux*`**
  - Remove its own task from the *task list*
  - Invariants: still need to keep the task in *runqueue* to be scheduled

→ Used by malwares like *adore-ng*



VMI for Hidden Process Detection

- **Find tasks in *runqueue* but not in *task list***
  - Run queue: `1,3`
  - Task list: `0,2,3`
  - **`Task-1`** is considered as malware



Example VMI Code

```
vmi_pause_vm()
/* traverse the tasks list */
foreach task in tasks_list:
    all_procs.add(task)
/* traverse the runqueue */
foreach run_task in runqueue:
    run_procs.add(run_task)
vmi_resume_vm()
```

Copy VM data

Pause VM to retrieve states

Do VMI check

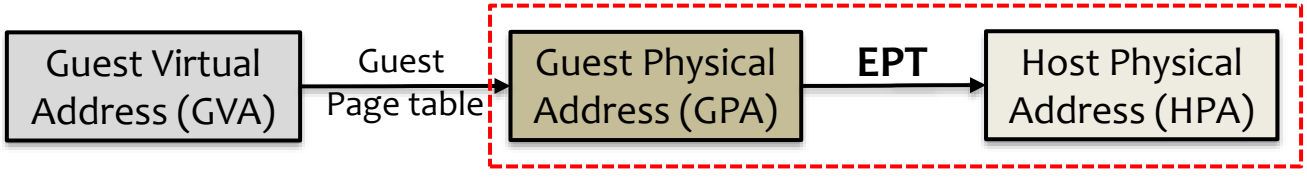
```
/* check if there is any hidden running process */
foreach pid in run_procs:
    check_if_pid_in_all_procs(pid, all_procs)
alarm_if_needed()
```

VMFUNC

GoSSIP

Review: Extended Page Table (EPT)

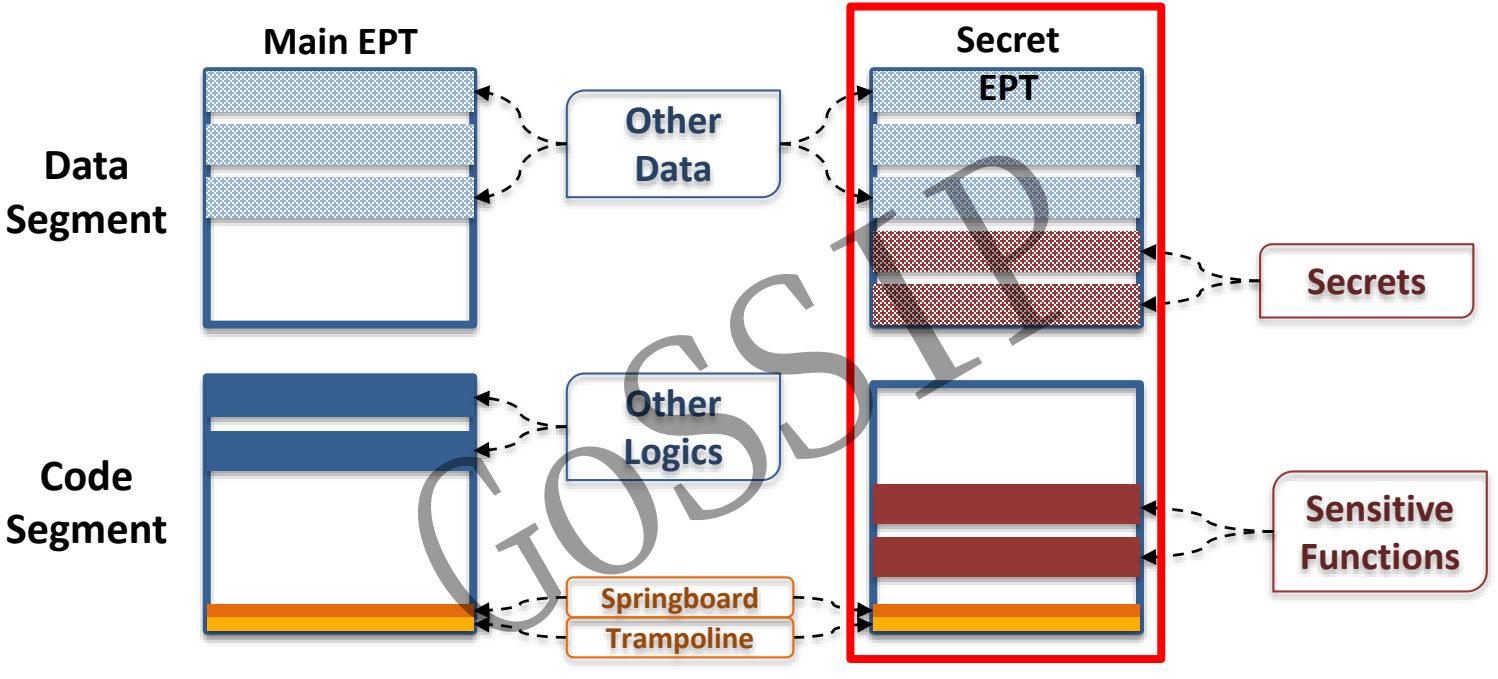
- **Translate guest physical addr to host physical addr**
  - The two-level translation are all done by hardware



- **EPT is manipulated and maintained by hypervisor**
  - Hypervisor controls how guest accesses physical address
  - Any EPT violation triggers VMExit to hypervisor

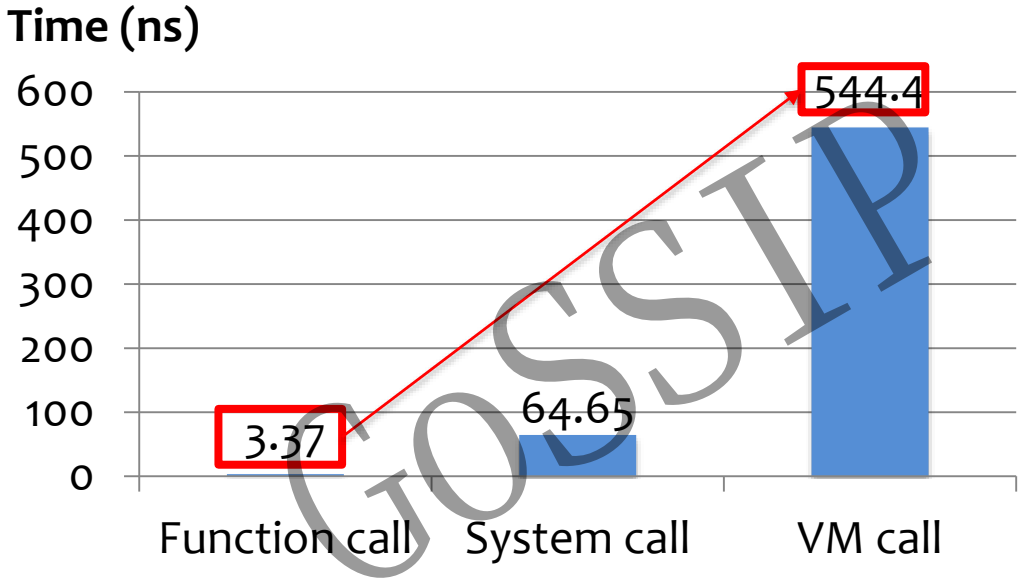
Memory Isolation using EPT Mechanism

- **Leverage EPT mechanism to shadow secret memory**
  - Data segment: secret memory is removed from main EPT
  - Code segment: sensitive functions only exist in secret EPT



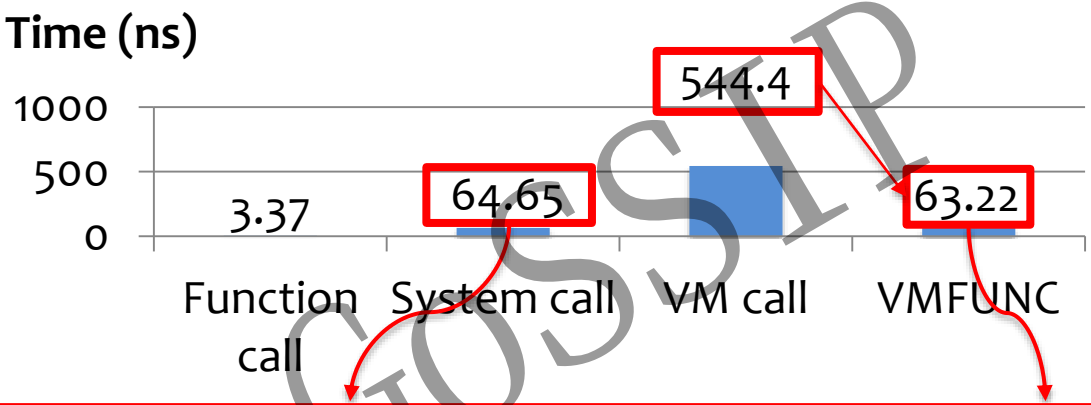
Problem of EPT protection

- **Context switch introduces large overhead**
  - Every EPT switch is intervened by hypervisor
  - VMExit takes much more time than function call



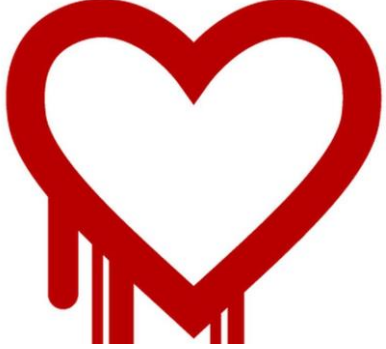
VM Function (VMFUNC) 101

- **VM Functions: Intel virtualization extension**
  - Non-root guest VMs can directly invoke some functions without VMExit
- **VM Function 0: EPTP Switching**
  - Software in guest VM can directly load a new EPT pointer

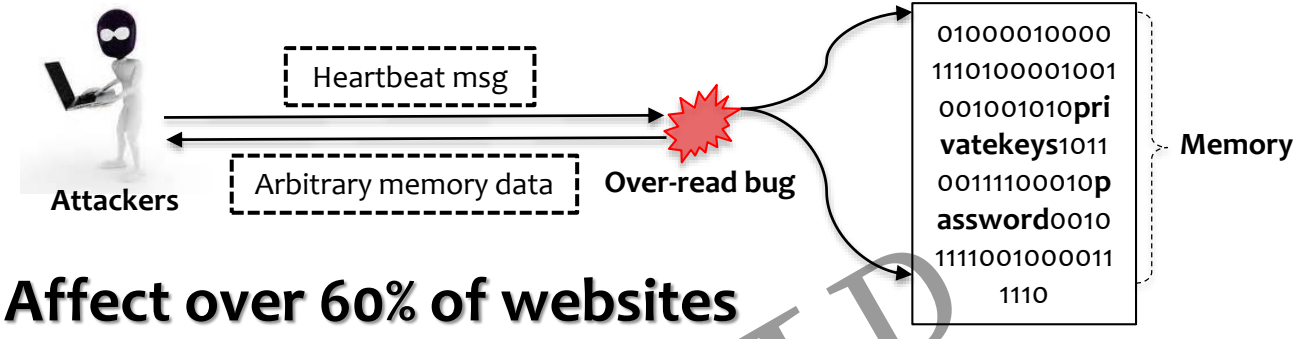


VMFUNC can provide the hypervisor-level function at the cost of system calls

HeartBleed Attack



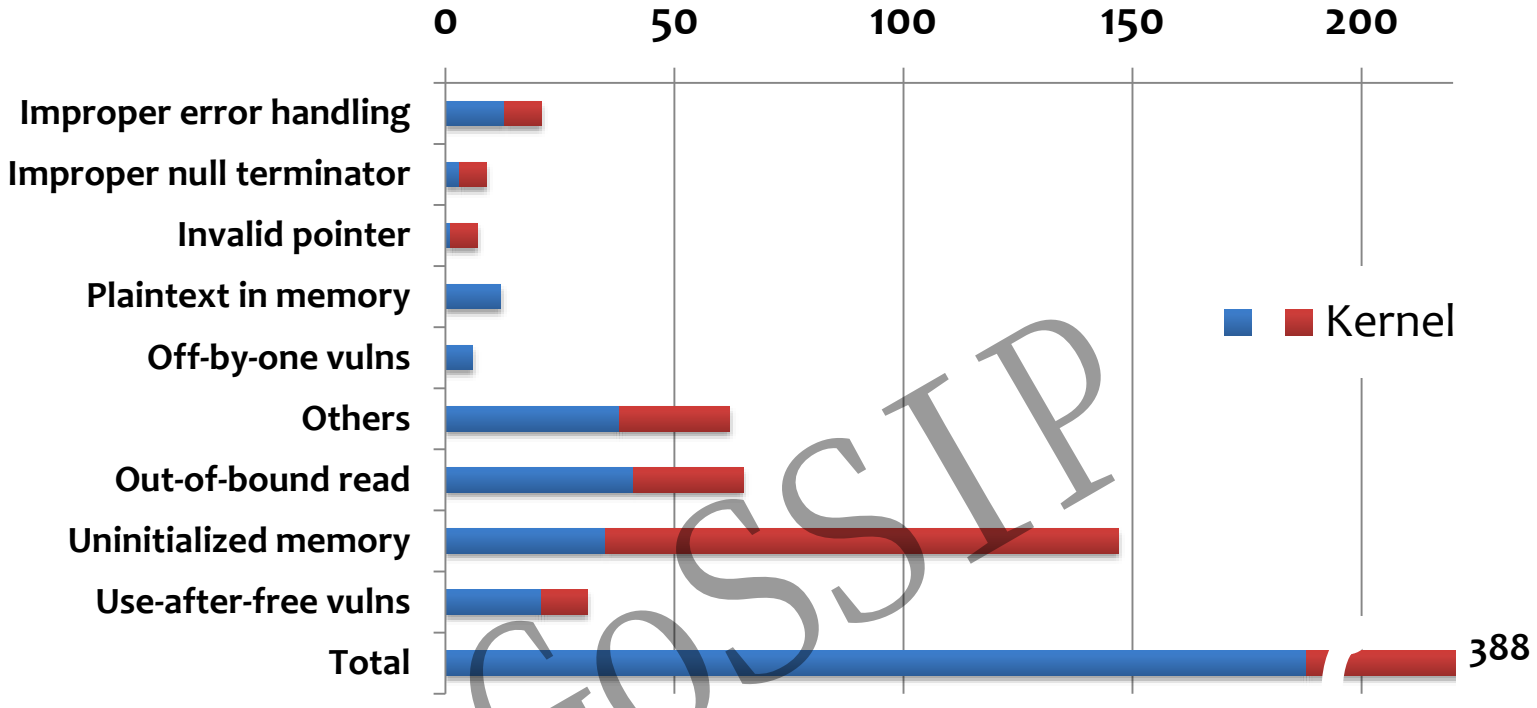
- **In-application memory disclosure attack**
  - One over-read bug discloses the whole memory data



- **Affect over 60% of websites**
  - Attackers can steal private keys and user accounts

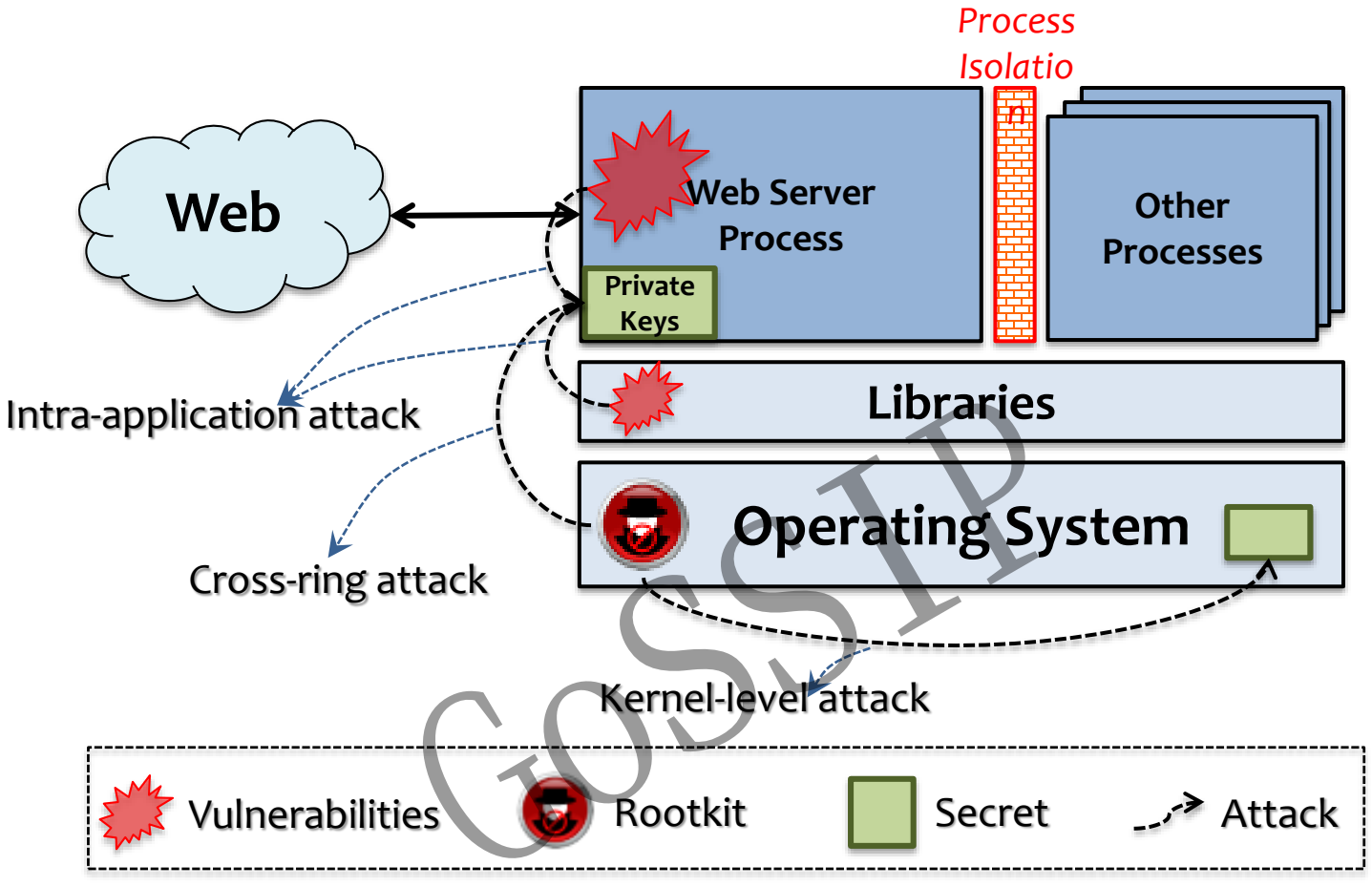
Memory Disclosure Vuls are Prevalent

- **CVEs summary (2000 ~ 2015)**



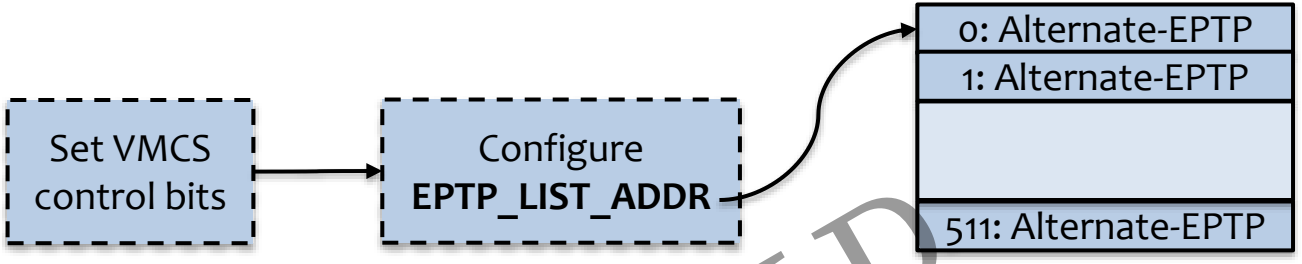
- **It is hard to prevent all of these bugs**

Large Attack Surface for Secret Disclosure



Using VMFUNC for Efficiency

- **Separate control plane from data plane**
  - **Control plane (policy):** hypervisor pre-configure the EPT used by different compartments

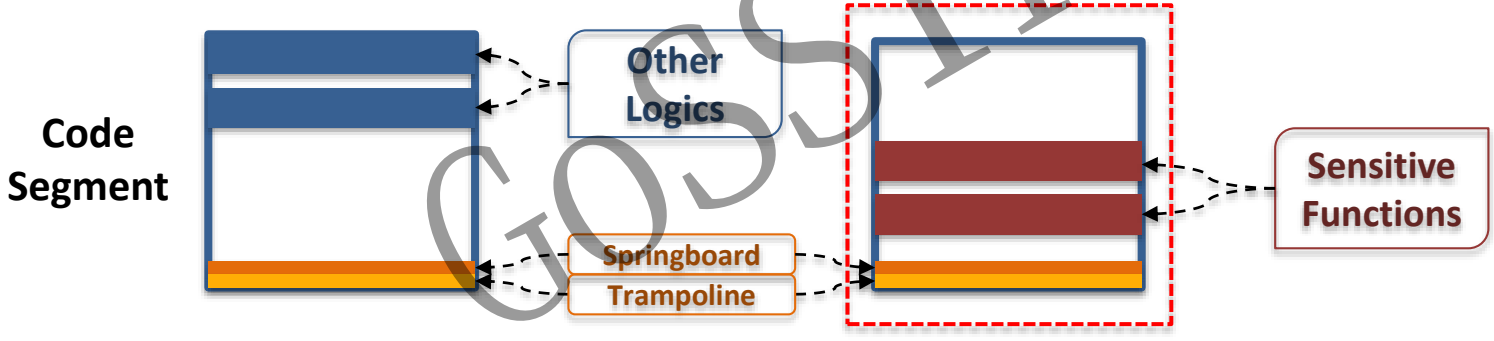


  - **Data plane (invocation):** application can directly switch EPT without hypervisor intervention

EPTP Switching invocation: VMFUNC opcode (EAX = 0, ECX = EPTP_index)

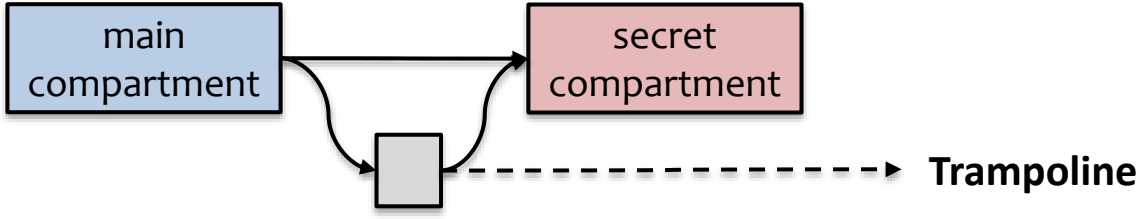
Security Problem of VMFUNC

- **What if attackers directly switch EPT?**
  - Since EPT switching is not checked by hypervisor
- **Recall: the code segment of the secret compartment**
  - It only contains trusted sensitive functions
  - The legal entrances to the secret compartment are fixed
  - Invalid VMFUNC invocation causes EPT violation

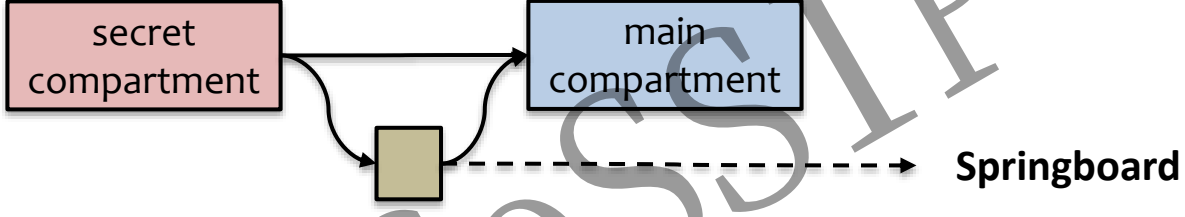


Secret Compartment is Not Self-contained

- **Main compartment may invoke sensitive functions**



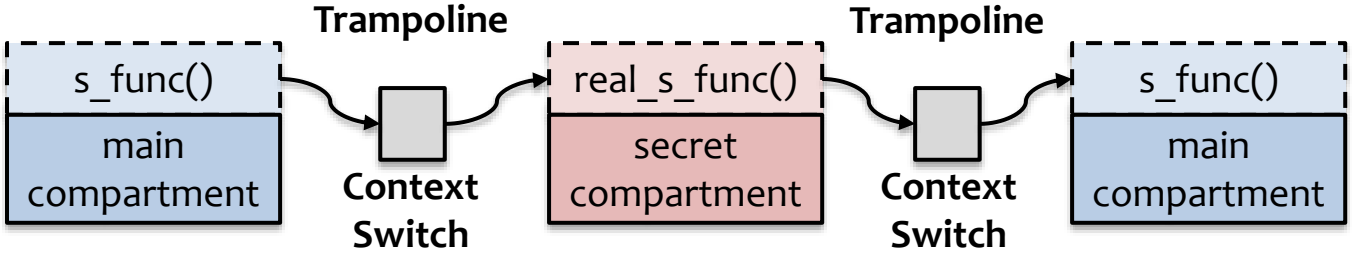
- **Secret compartment may invoke normal functions**
  - E.g., system call,library call, etc.



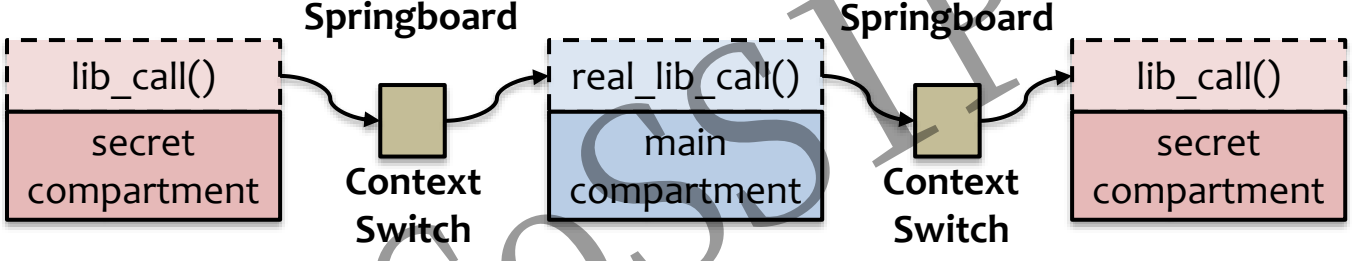
- **Different compartments have different context**
  - We use trampoline and springboard to switch the context

Use VMFUNC to Secure Execution Flow

- **Trampoline control flow**



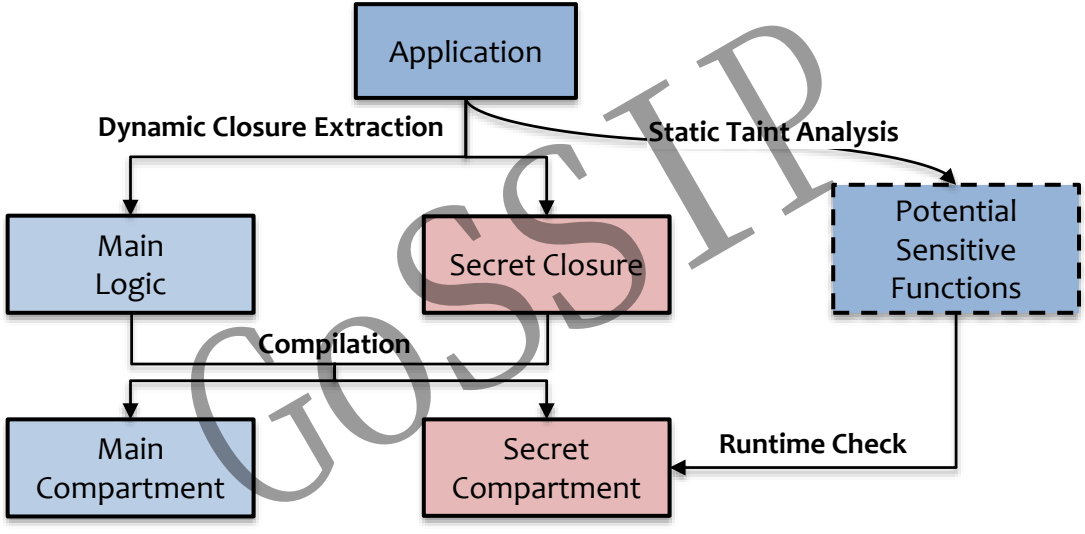
- **Springboard control flow**



Context switch is done using VMFUNC

Application Decomposition in SeCage

- **A hybrid approach to decomposing application**
  - Dynamic approach to extracting the **secret closure**
  - Automatic decomposition during compilation time
  - Static approach to getting the complete potential sensitive functions, used to avoid corner case during runtime



INTEL SGX

GoSSIP

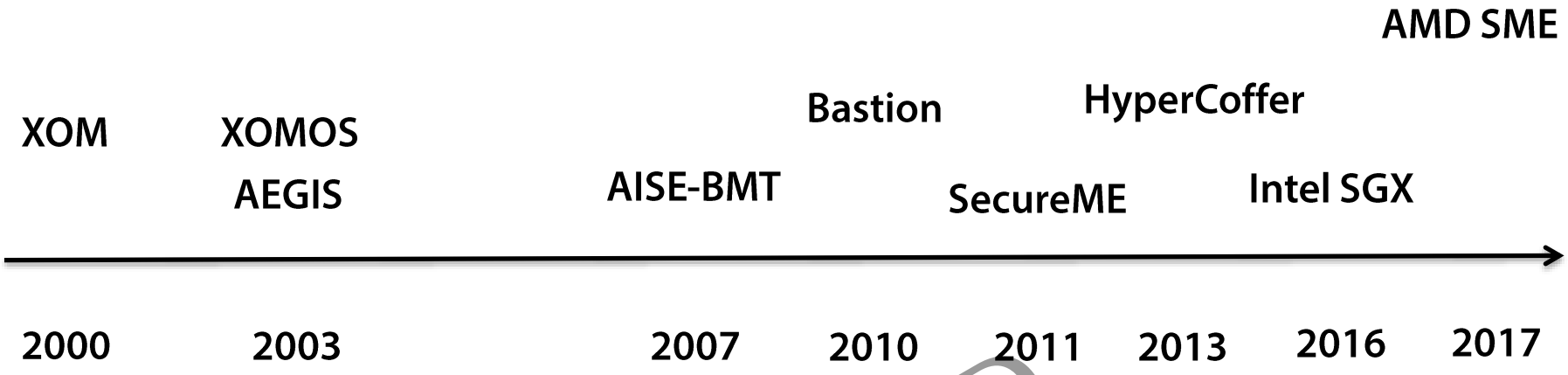
Why Intel SGX?

- Motivation: untrusted privileged software
  - E.g., protect application from **untrusted OS**
- What if the OS direct accesses application's memory?
  - Data are **encrypted** in memory
  - Data can only be accessed by the app within CPU boundary
  - The TCB contains only the CPU and app, no OS

How can Memory Always be Encrypted?

- **Question**: data will **eventually be decrypted** when using
  - Then, what if an attacker steal data when it is being used?
- **Solution**: only decrypt data **inside CPU (in cache)**
  - The attacker now has to steal data directly from CPU

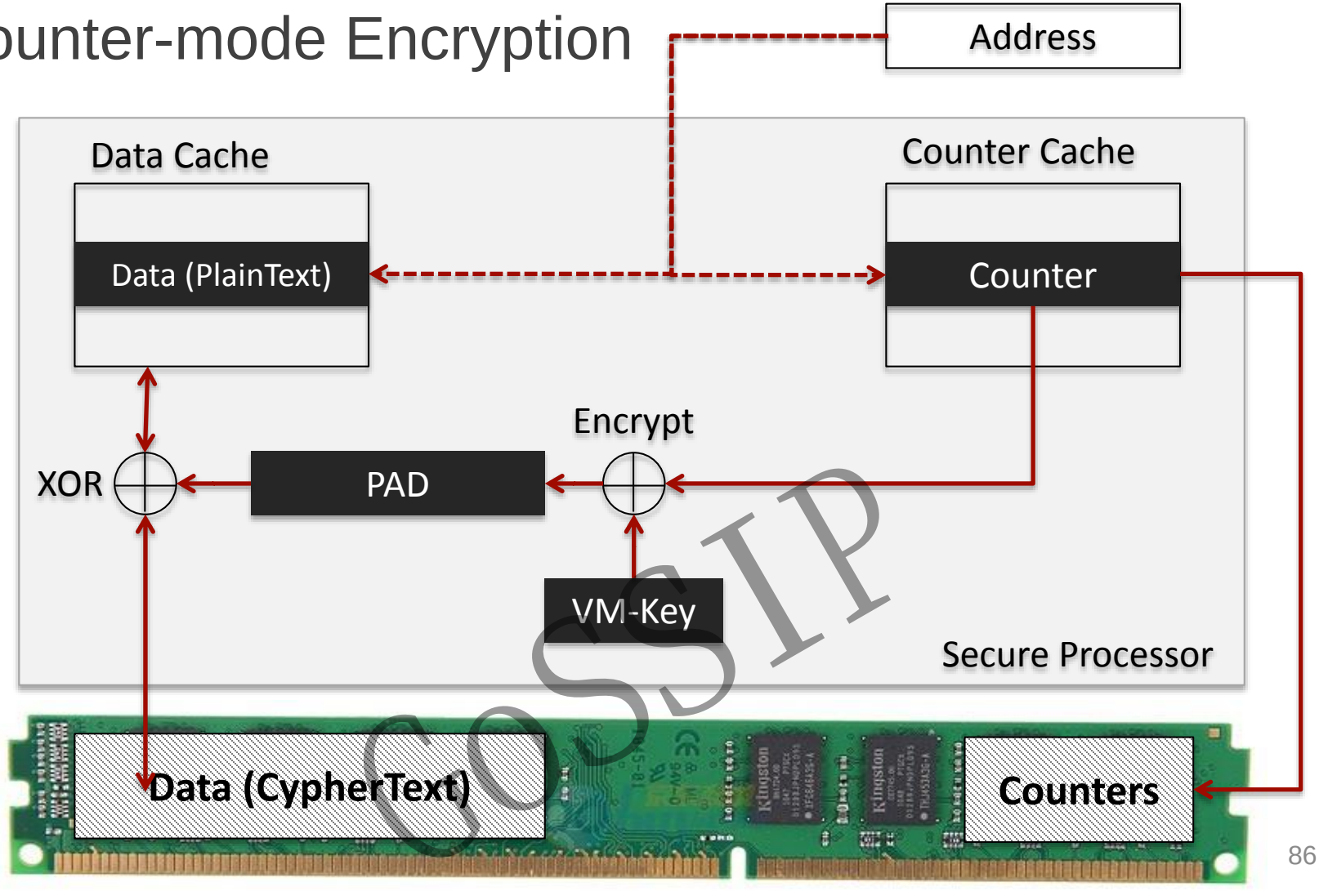
History of Memory Encryption



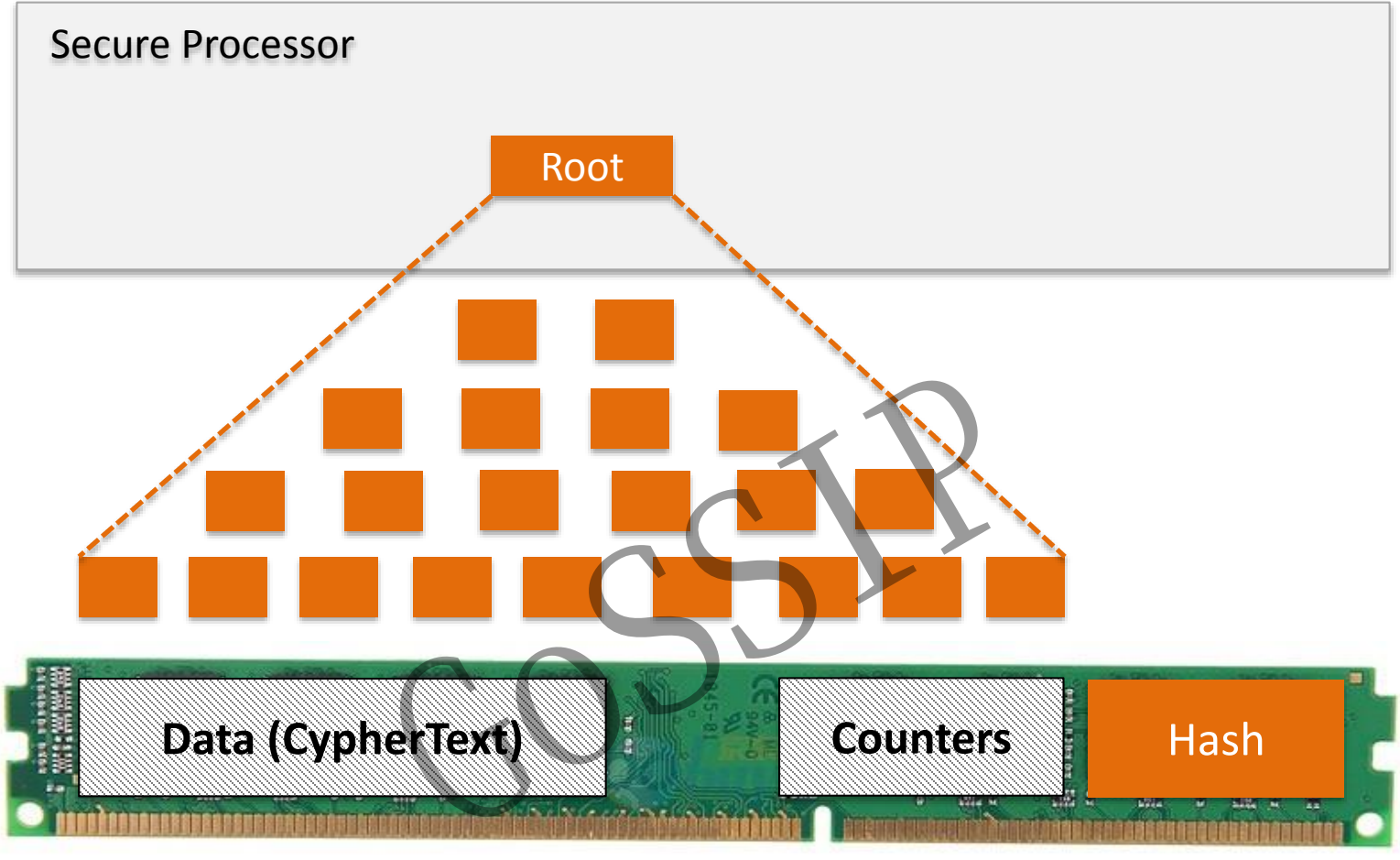
Counter-mode encryption

Merkle-tree based integrity check

Counter-mode Encryption

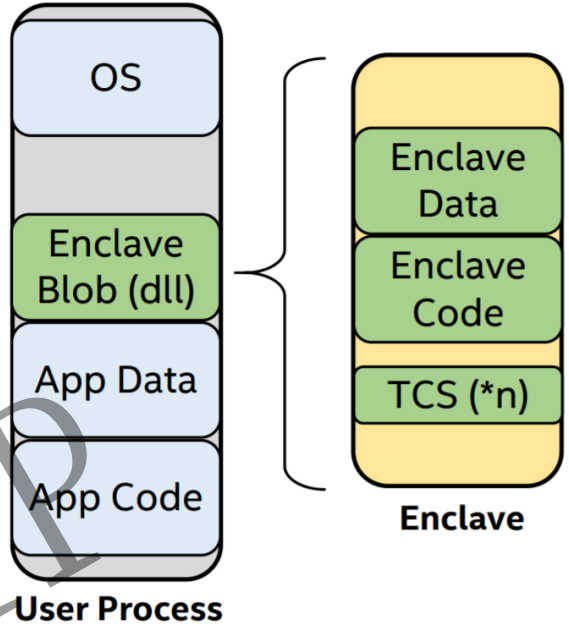


Merkel Tree for Data Integrity



Process View

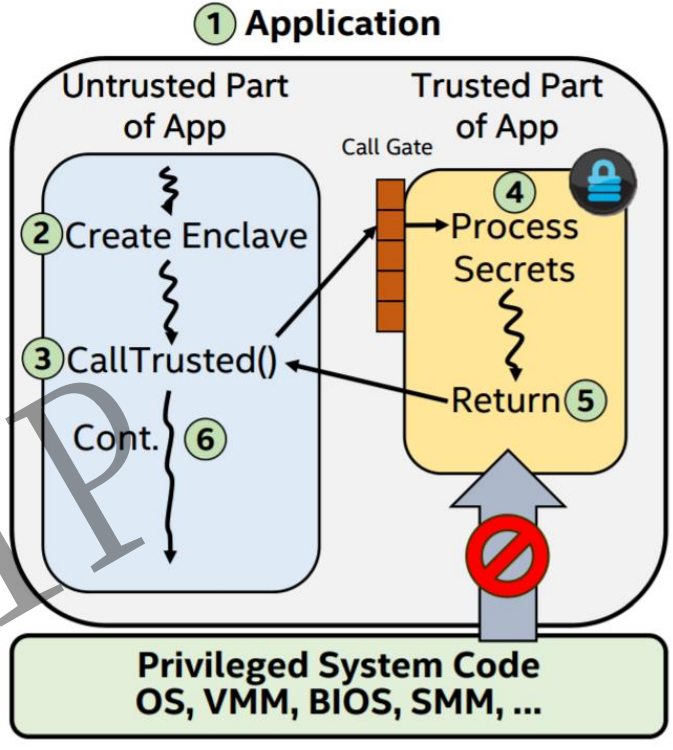
- With its own code and data
- Providing Confidentiality & Integrity
- Controlled entry points
- Multi-thread support
- Full access to app memory and processor performance



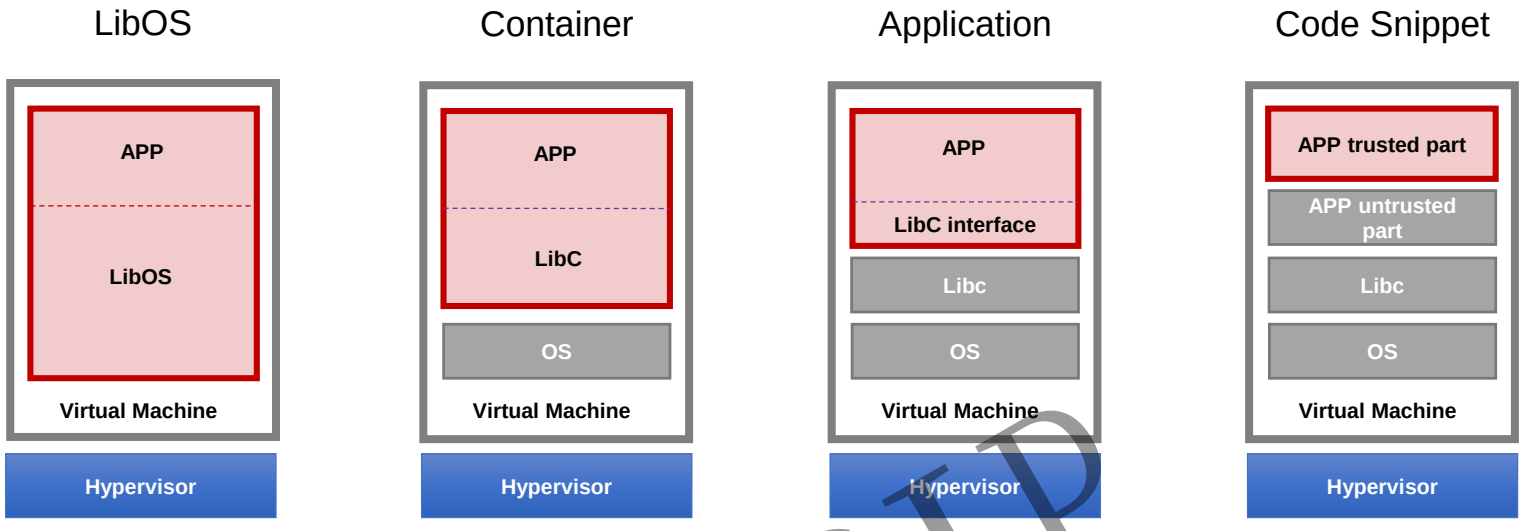
Protected execution environment embedded in a process

SGX Execution Flow

1. App built with trusted and untrusted parts
2. App runs & creates the enclave which is placed in trusted memory
3. Trusted function is called, execution transitioned to the enclave
4. Enclave sees all process data in clear; external access to enclave data is denied
5. Trusted function returns; enclave data remains in trusted memory
6. Application continues normal execution



Software Architectures of SGX



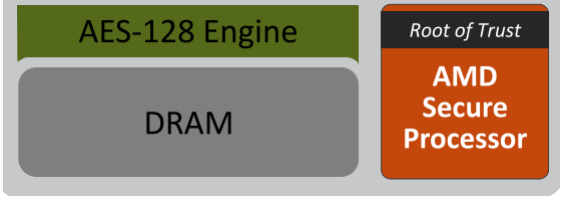
	Compatibility	TCB Size	Ocall Num	Attack Surface	Protect OS
LibOS	Part	Large	Few	Small	✓
Container	✓	Mid	Mid	Mid	✗
Application	✓	Mid	Many	Large	✗
Code Snippet	✗	Small	Few	Small	✗

SME: Secure Memory Encryption / SEV: Secure Encrypted Virtualization

AMD’S SME/SEV

2016: AMD x86 Memory Encryption Technologies

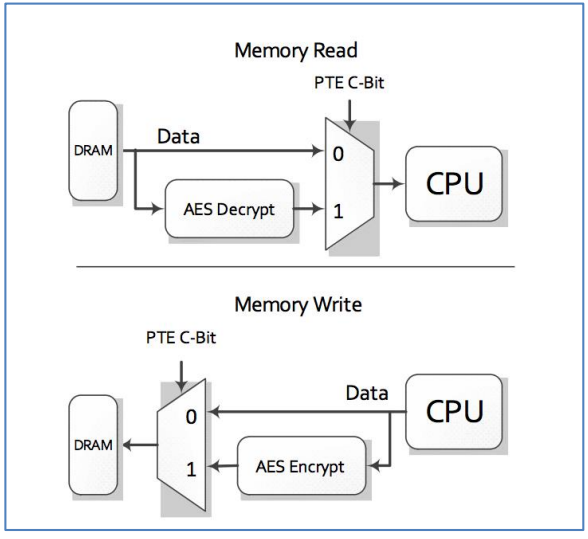
- Two Technologies
  - AMD Secure Memory Encryption (SME) & AMD Secure Encrypted Virtualization (SEV)
- Features
  - Hardware AES engine located in the memory controller performs inline encryption and decryption of DRAM
  - Minimal performance impact: Extra latency only taken for encrypted pages
  - No application changes required
  - Encryption keys are managed by the AMD Secure Processor and are hardware isolated
    - Not known to any software on the CPU

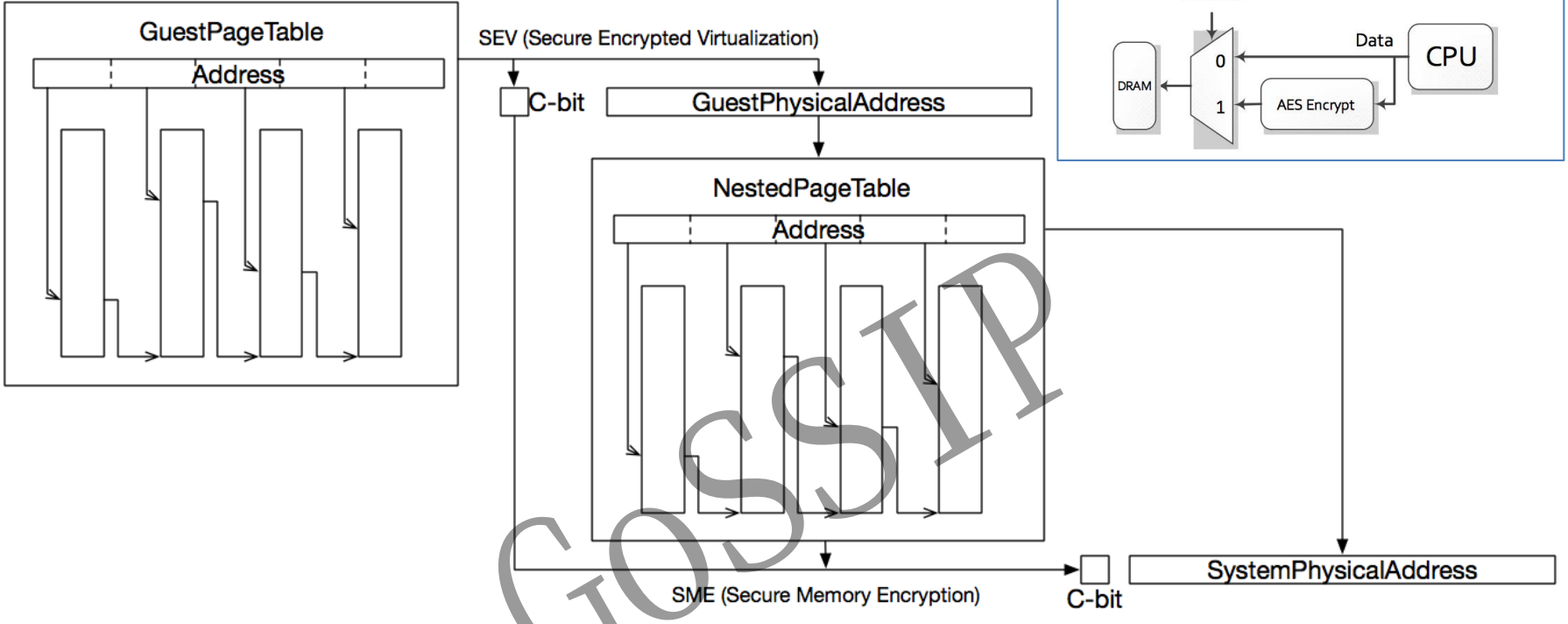


Comparing with Intel SGX

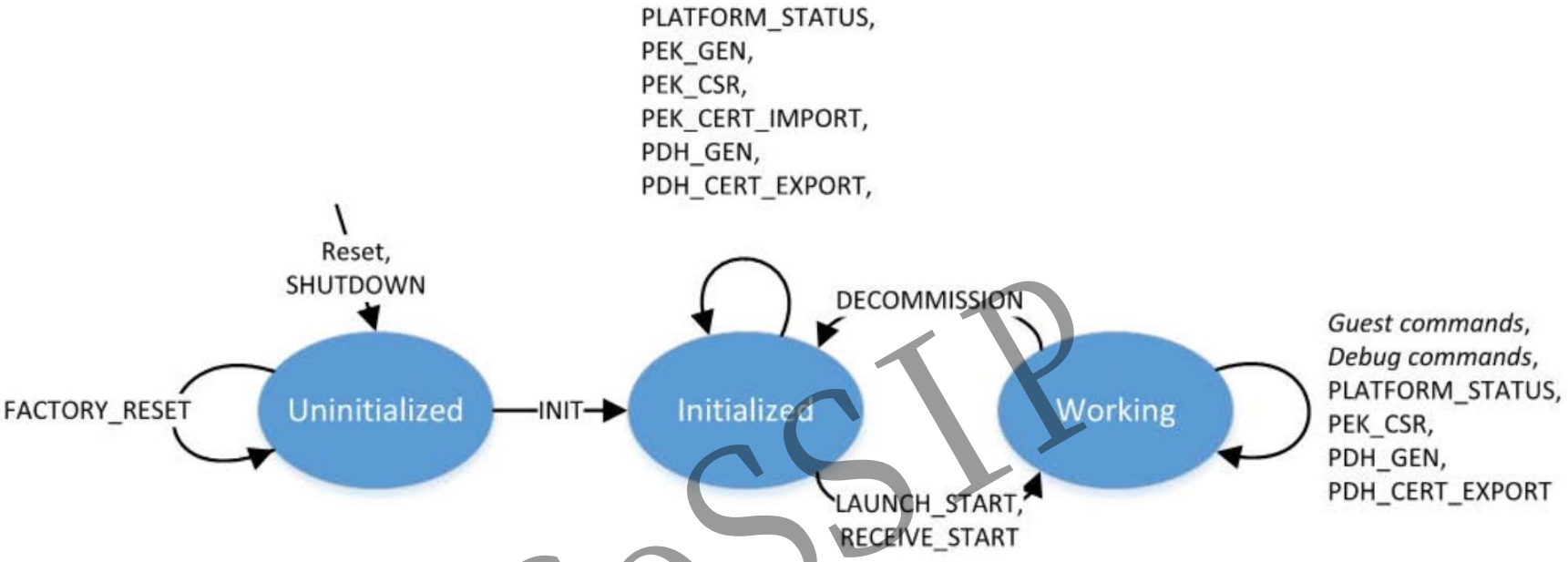
- The SME approach is different
  - It will not protect memory from an attacker who has compromised the kernel
  - It is intended to protect against cold-boot attacks, snooping on the memory bus, and the disclosure of transient data stored in persistent-memory arrays
- SEV focuses on virtualization
  - it can protect virtual machines from each other
  - keeping their contents secure even if one of them manages to compromise the host system
  - It should be able to protect virtual machines from the hypervisor itself

Usage of SEV

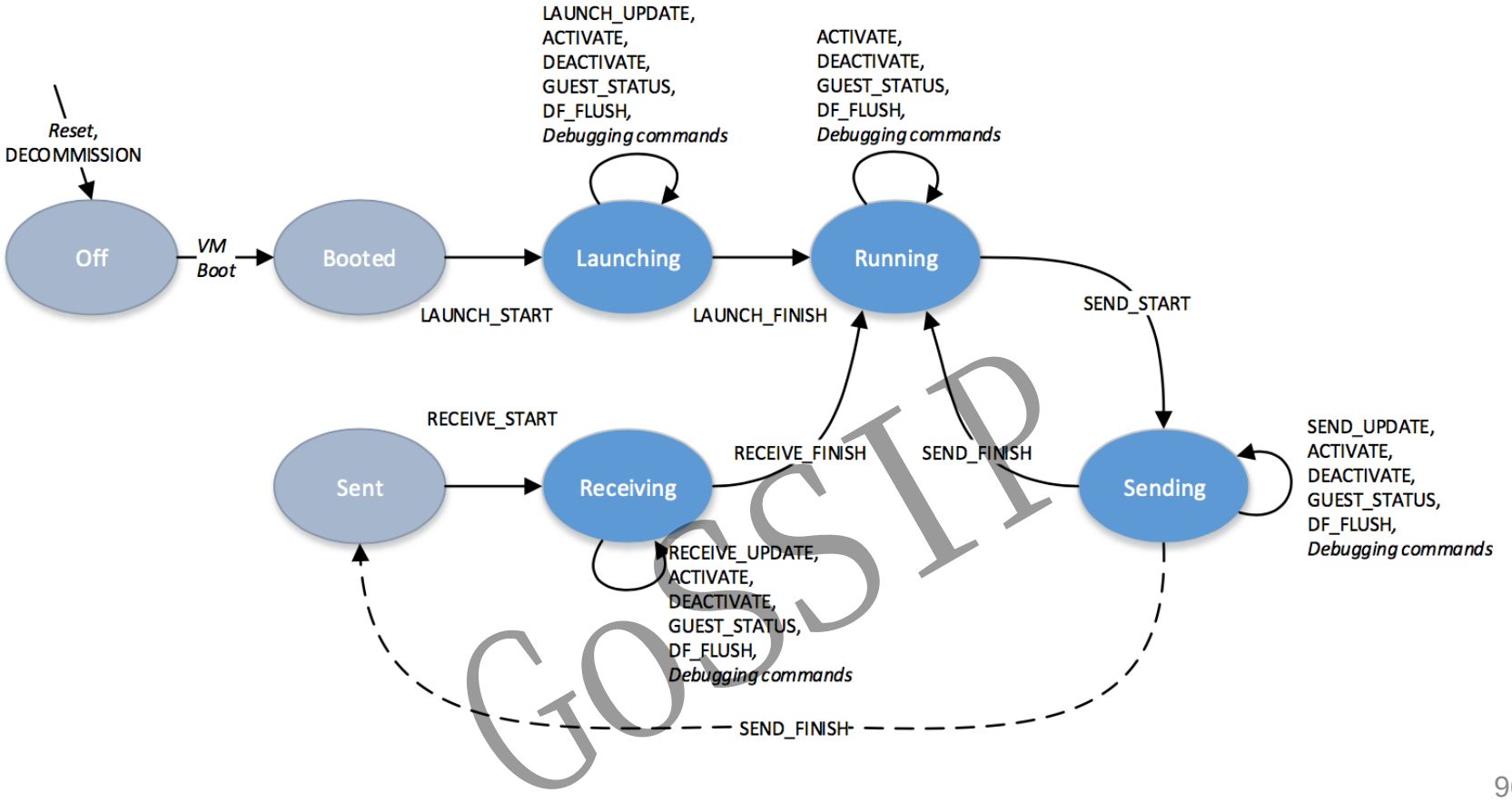




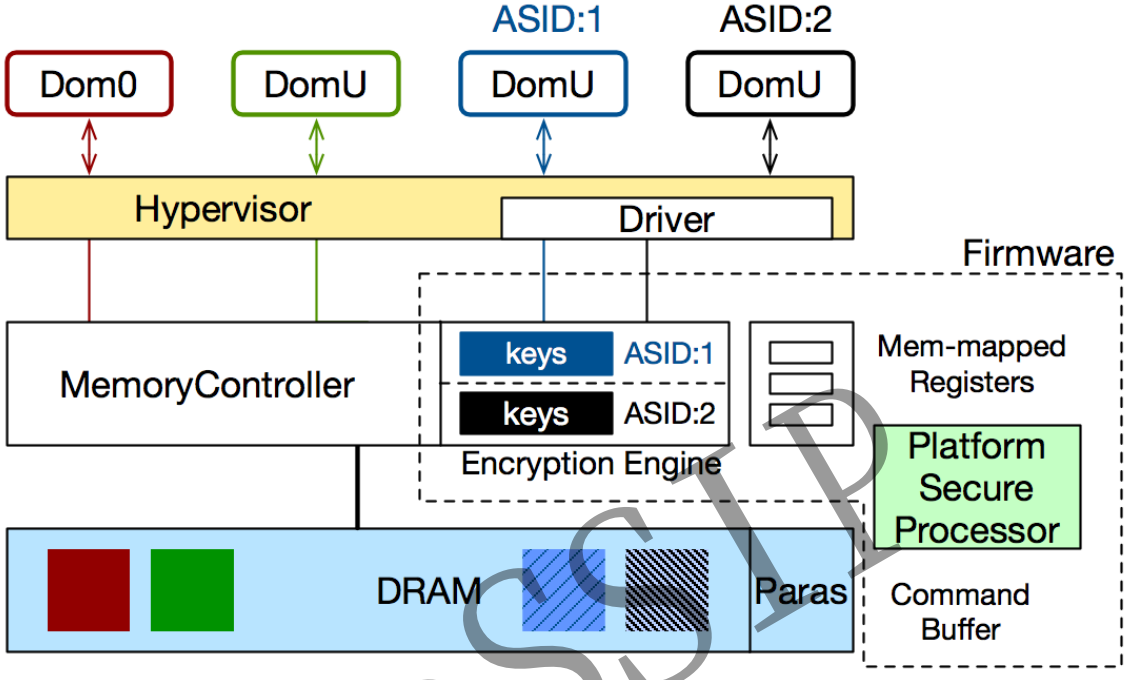
Lifecycle of Firmware



Lifecycle of Guest

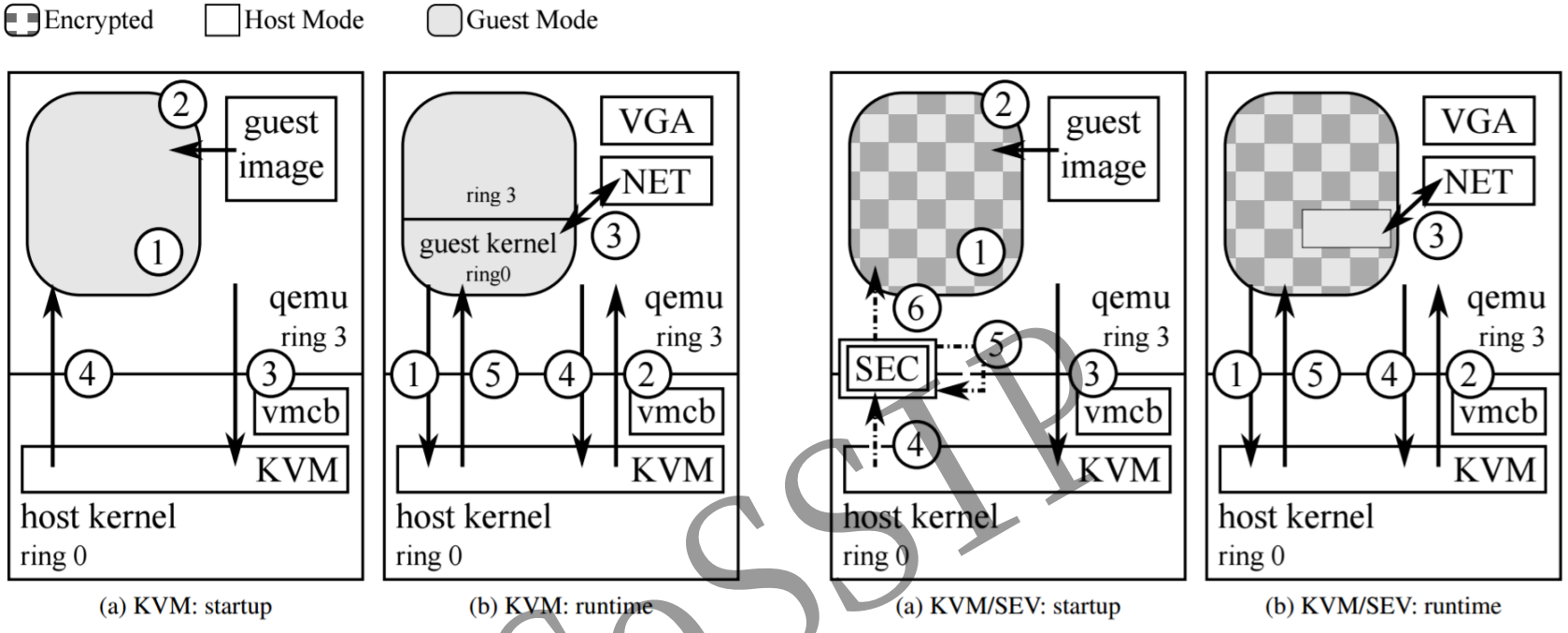


Architecture-Protect VM



Assume hypervisor is benign (not malicious, but not trusted)

Runtime Behavior of SEV



(a) KVM: startup (b) KVM: runtime (a) KVM/SEV: startup (b) KVM/SEV: runtime

Figure 1: QEMU/KVM architecture

Figure 2: SEV-enabled QEMU/KVM architecture

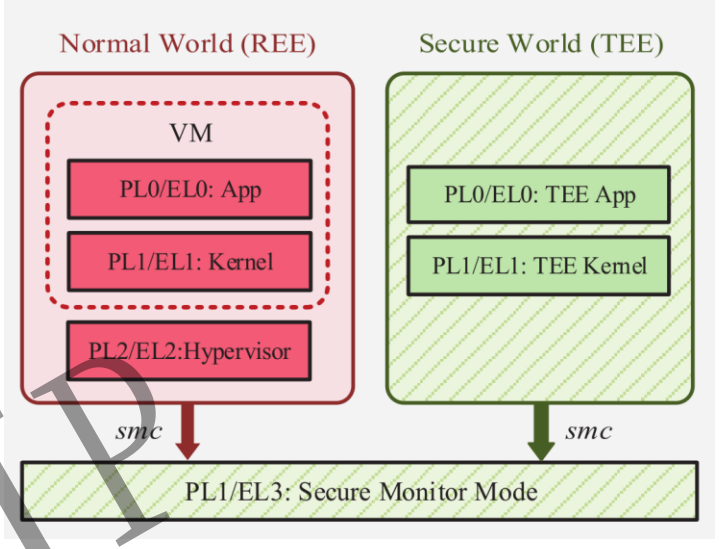
AMD SEV Security Considerations

1. The GP registers are not encrypted upon a vmexit
2. The vmcb is subject to manipulation by the hypervisor
3. There is no memory authentication scheme in use

ARM TRUSTZONE

TrustZone and TEE

- Two execution environments
  - REE: Normal world
  - TEE: Secure world
- REE (Rich Execution Environment)
  - Run rich OS and apps, e.g., Android
- TEE (Trusted Execution Environment)
  - Run small OS and apps
  - Has higher privilege



TEE on In-Vehicle Infotainment

- Secure Authentication
  - Start through fingerprint
  - Secure payment for digital content, oil, etc.

- Secure Connection
  - Internet: Through SoftSIM to switch between carriers
  - Connection with smartphone for unlocking and remote controlling

- Isolation with Entertainment
  - Use TEE for secure authentication and connection

TEE in Drones

- Secure Control Policies
  - No-fly zone: using GPS to restrict fly zone through TEE
  - Owner authentication: using biometrics on remote controller
  - Other fly-policies: return to specific spot under certain conditions
- Security Enforcement
  - Enforce policies through secure boot/secure storage
  - Tamper-resistant even under physical attacks

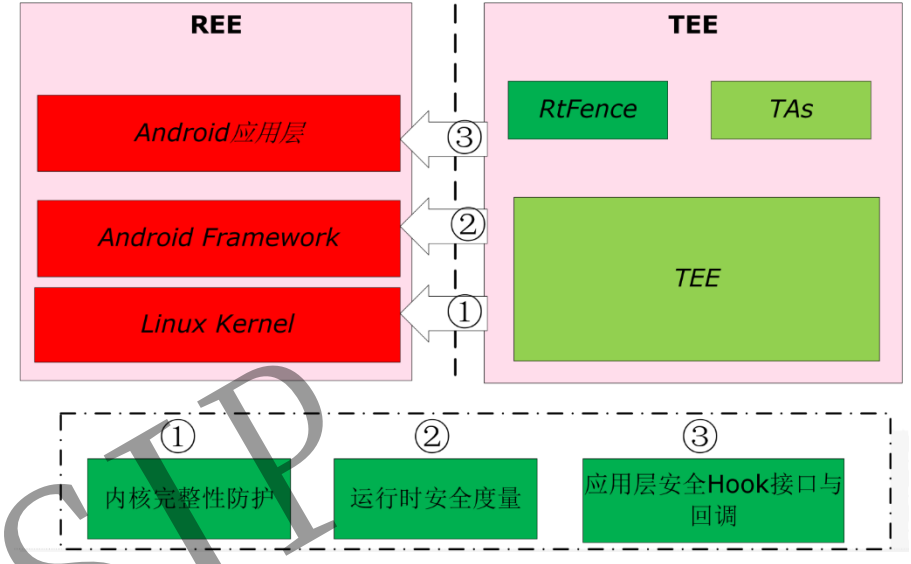
Security of Wearable Devices

- Binghamton Univ.
  - Yan Wang et al.
  - Infer PIN code through hand movement
  - Accuracy: 80%
  - After 3 times tries: >90%
- Security Suggestion
  - Apply TEE on devices
  - Use the other hand…

Wang, Chen, et al. "Friend or Foe?: Your Wearable Devices Reveal Your Personal PIN." AsiaCCS. ACM, 2016.

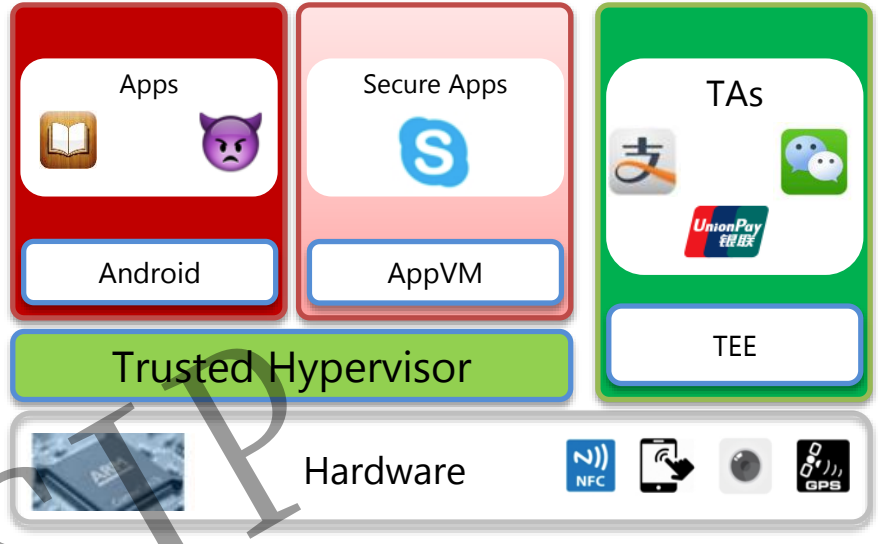
Real-time Kernel Protection on TEE

- **Kernel Integrity Monitoring**
  - Enforce the integrity of kernel's critical data and code
- **Runtime Measurement**
  - Protect the lifetime of critical data
- **Hook/callback of Secure Service**
  - Offer interface for up-level software

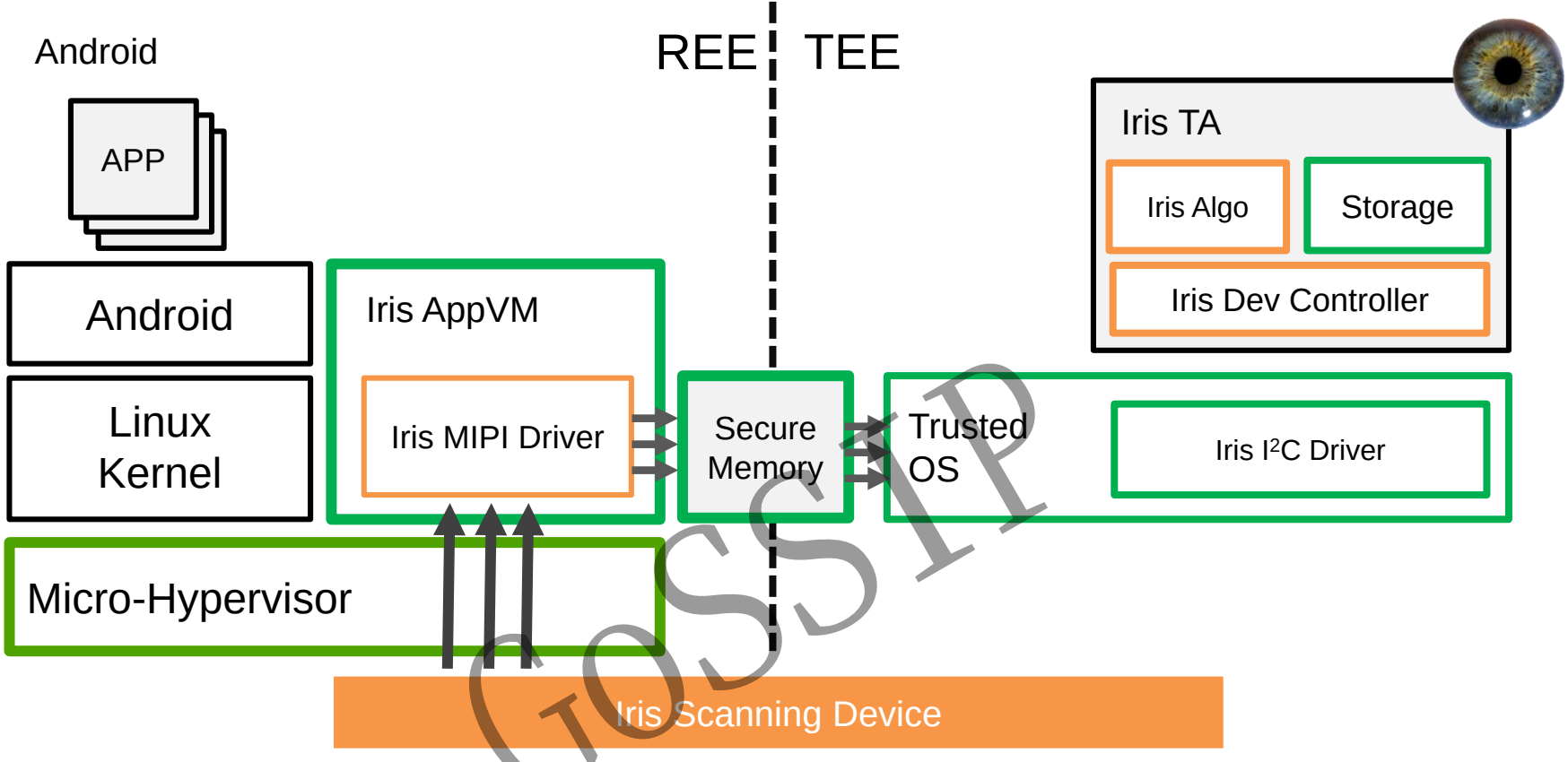


TEE & Hypervisor: More Flexible Architecture

- High-bandwidth Tasks
  - Current TEE is for low-bandwidth tasks
  - Hard to be compatible with current system architecture
  - E.g., secure iris, 4K video
- Micro-Hypervisor Solution
  - Lightweight way for security
  - Meet the demand of most devices



TEE-based Iris Protection Solution



Part-3

Hardware Features Not Designed for Security

TSX: Transactional Synchronization eXtensions

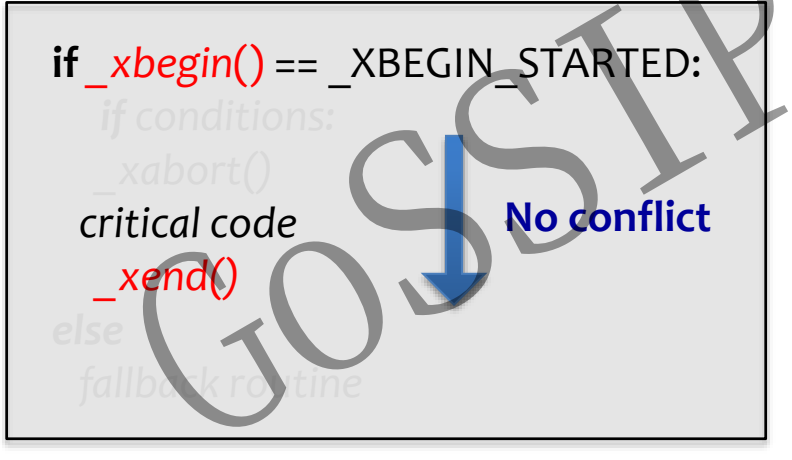
INTEL TSX

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 all-or-nothing and before-or-after
- **Real-world best-effort TM**
  - Limited read/write set
  - System events may abort an TX

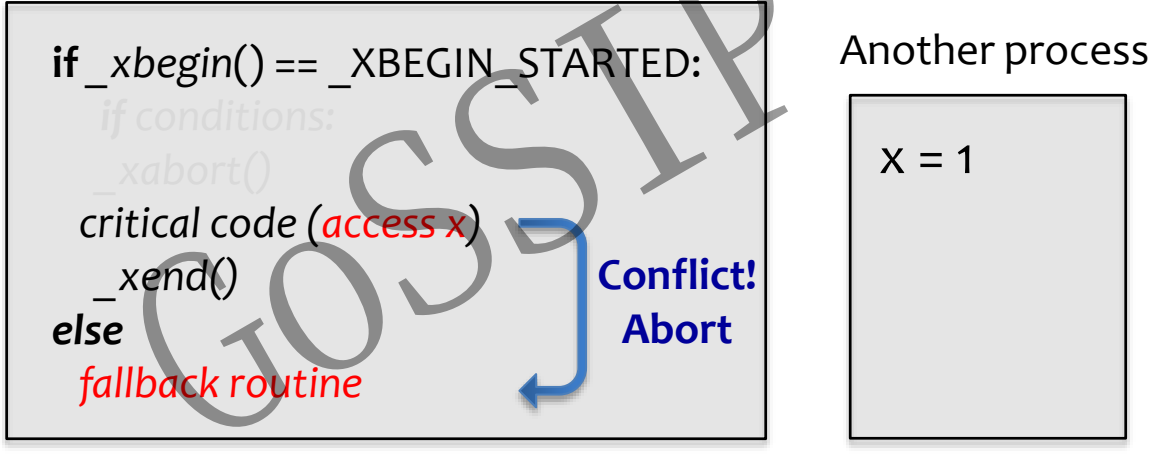
Programming with RTM

- **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



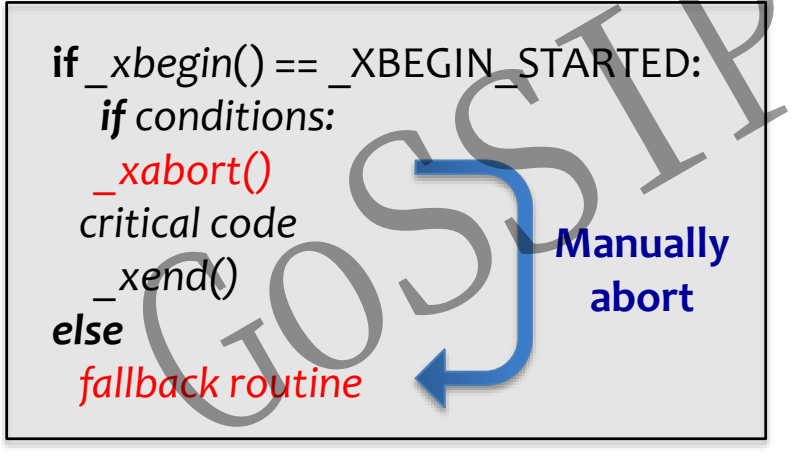
Programming with RTM

- **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



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**

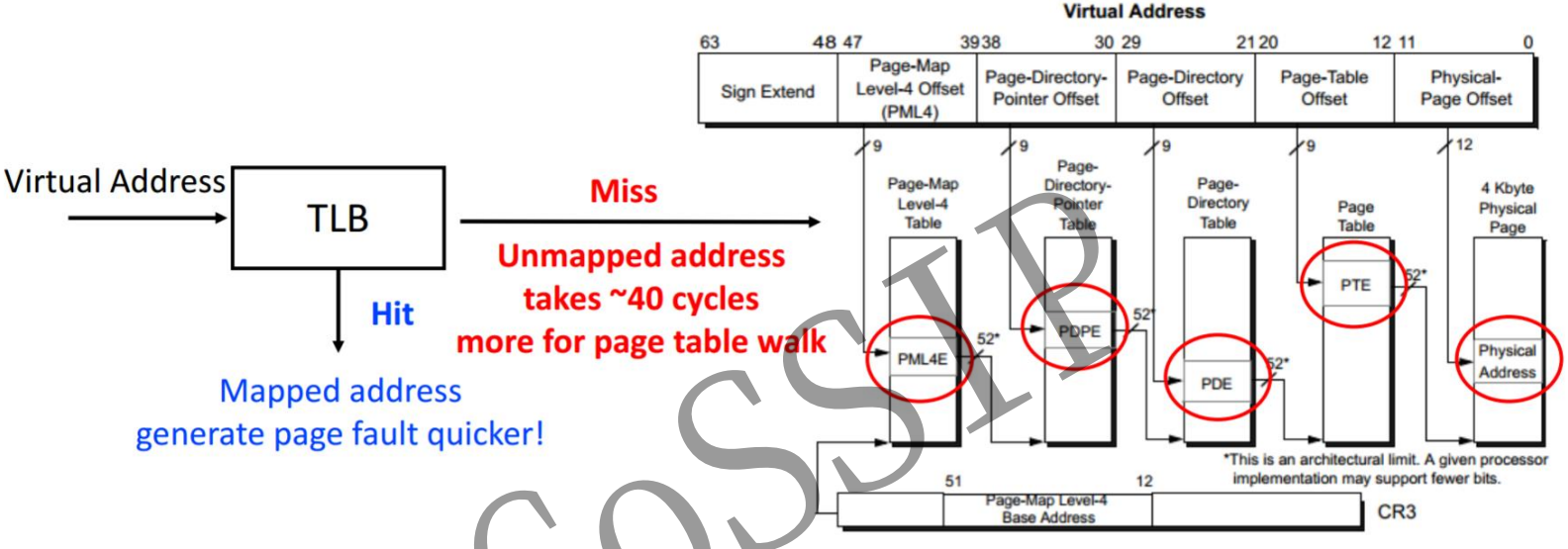


Using HTM for Data Protection

- 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

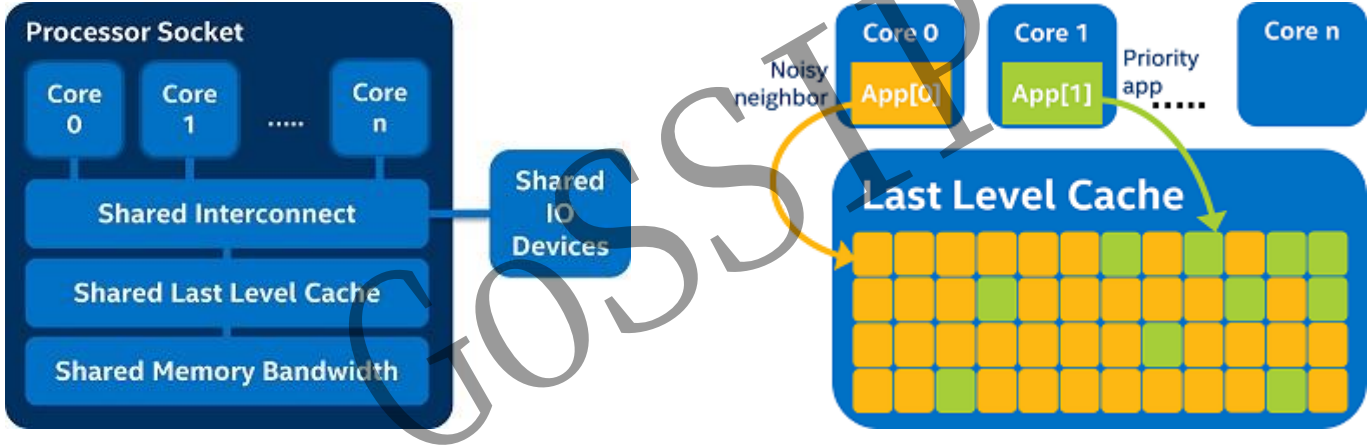


CAT: Cache Allocation Technology

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

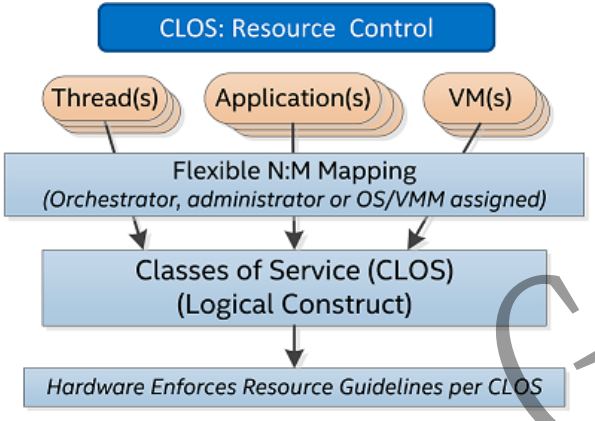


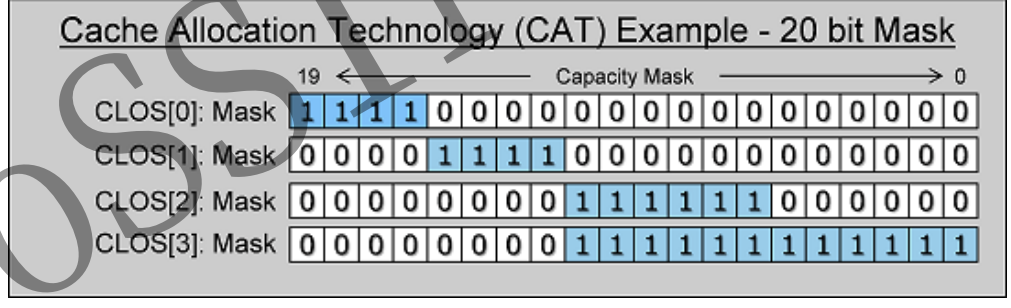
Software Controlled Cache Allocation

- 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

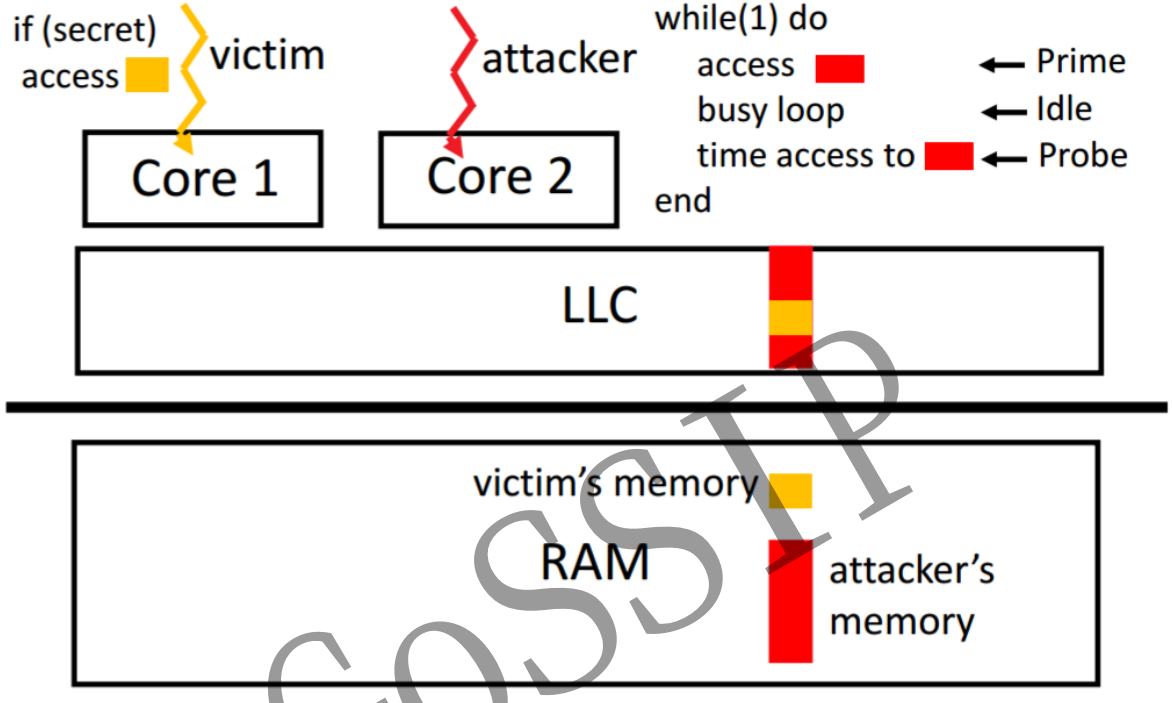
Class of Service (CLOS)

- 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



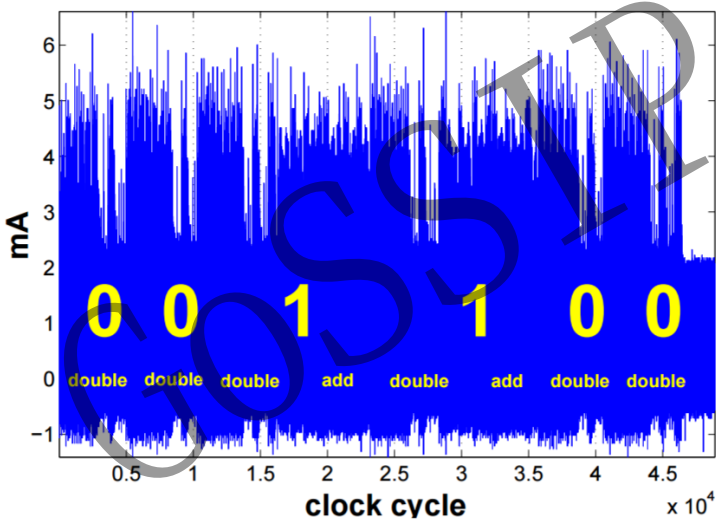


The PRIME+PROBE Attack



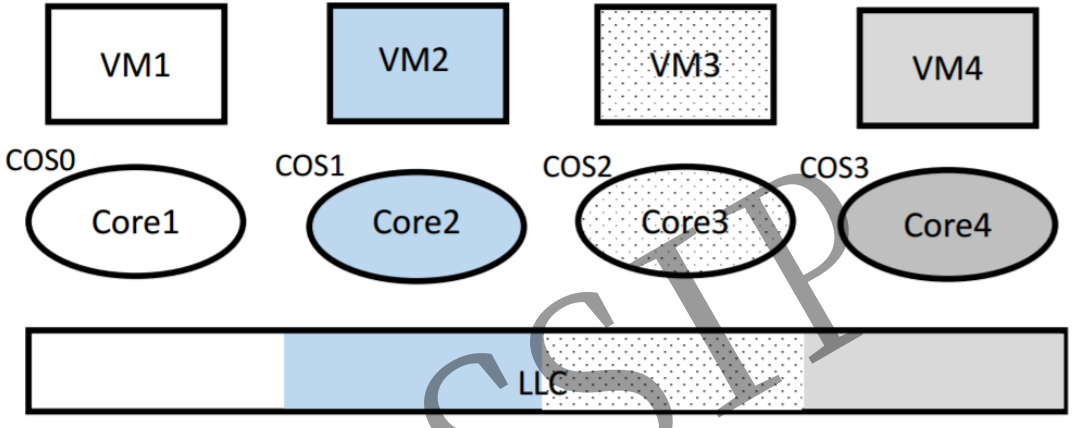
Cache Side-channel Attack

- 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



CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing

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

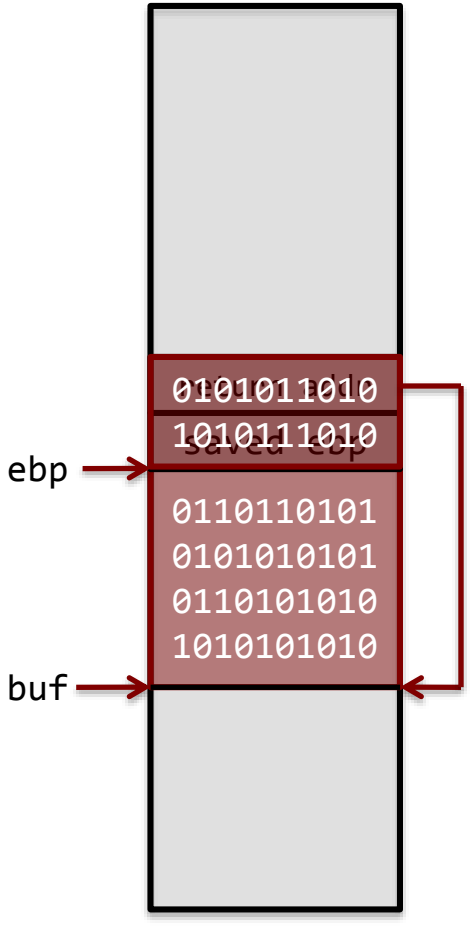
- A Typical Buffer Overflow Attack

```
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

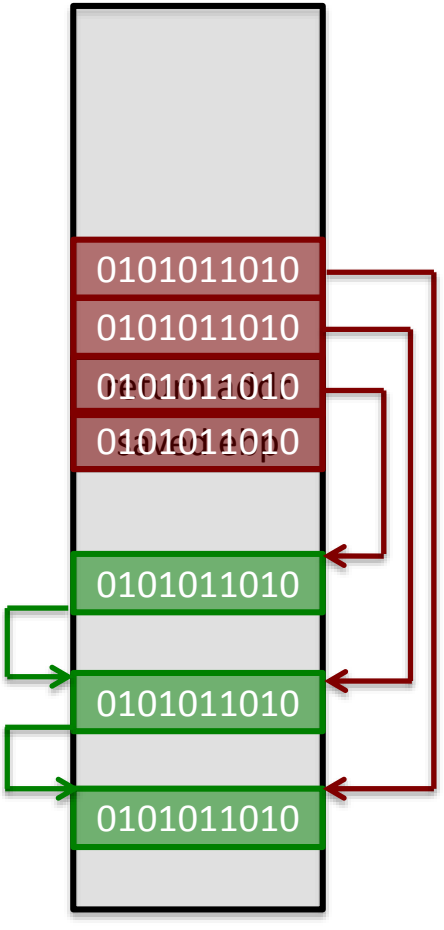
- **Solutions**
  - **StackGuard[Cowan'98], FormatGuard[Cowan'98]**
  - **Make data section non-executable**

- **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

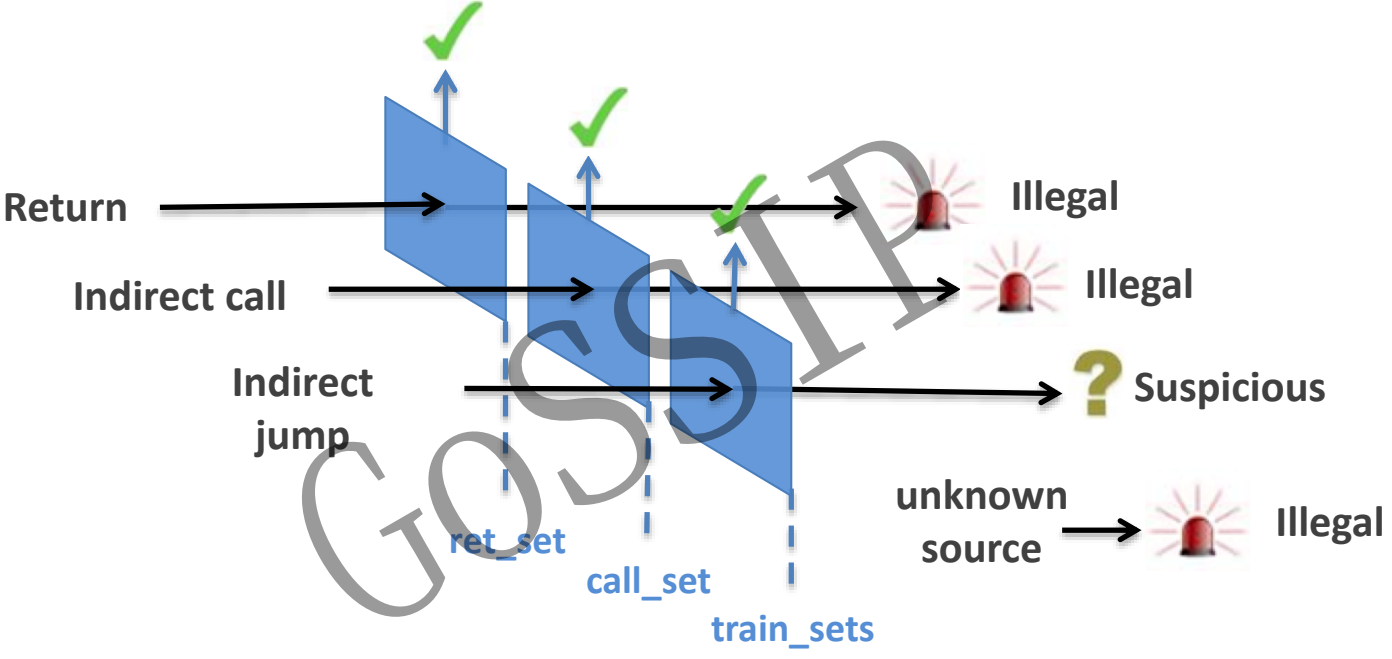
In Apache and its libraries

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



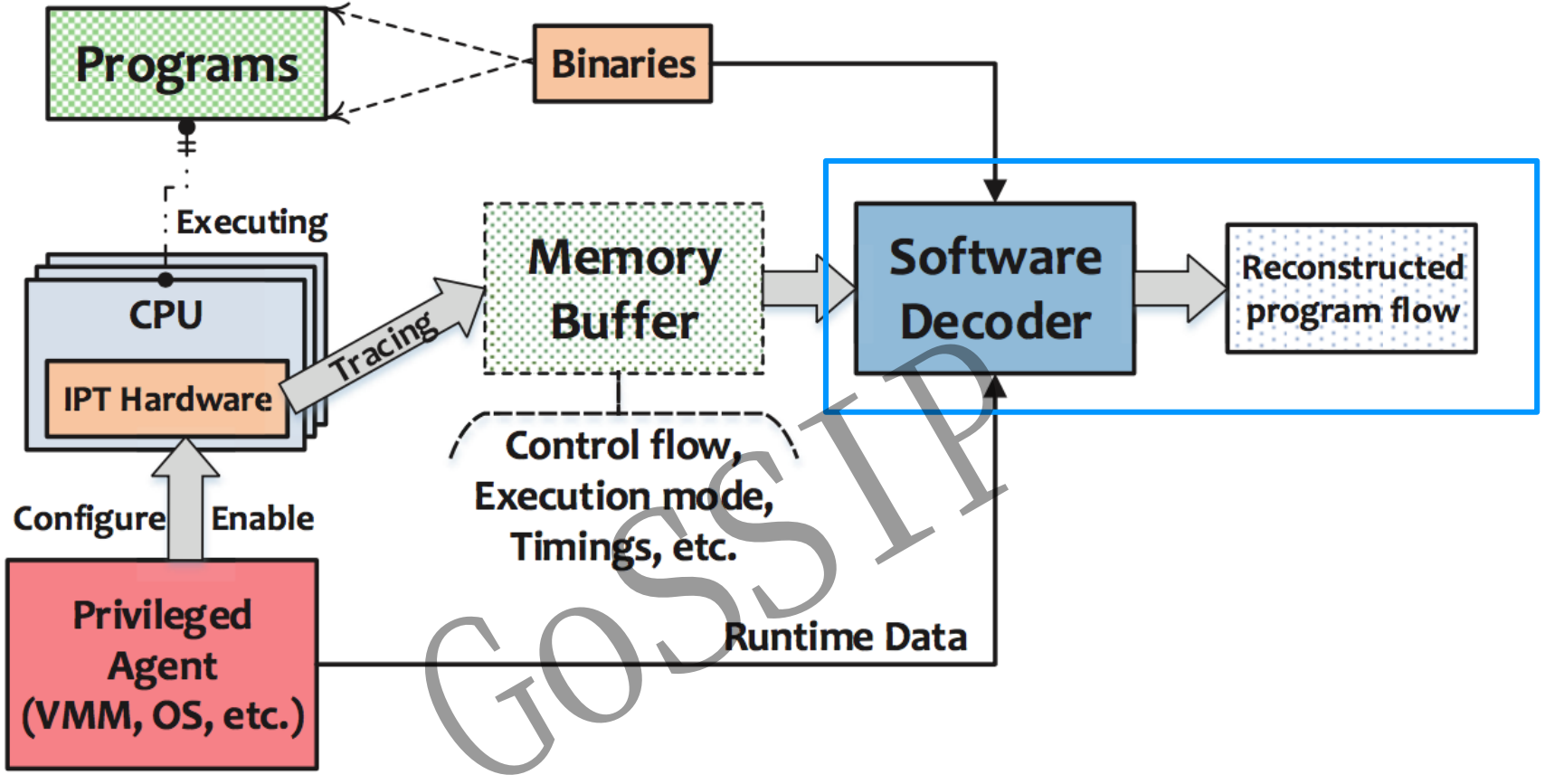
Intel PT: Intel Processor Tracing

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

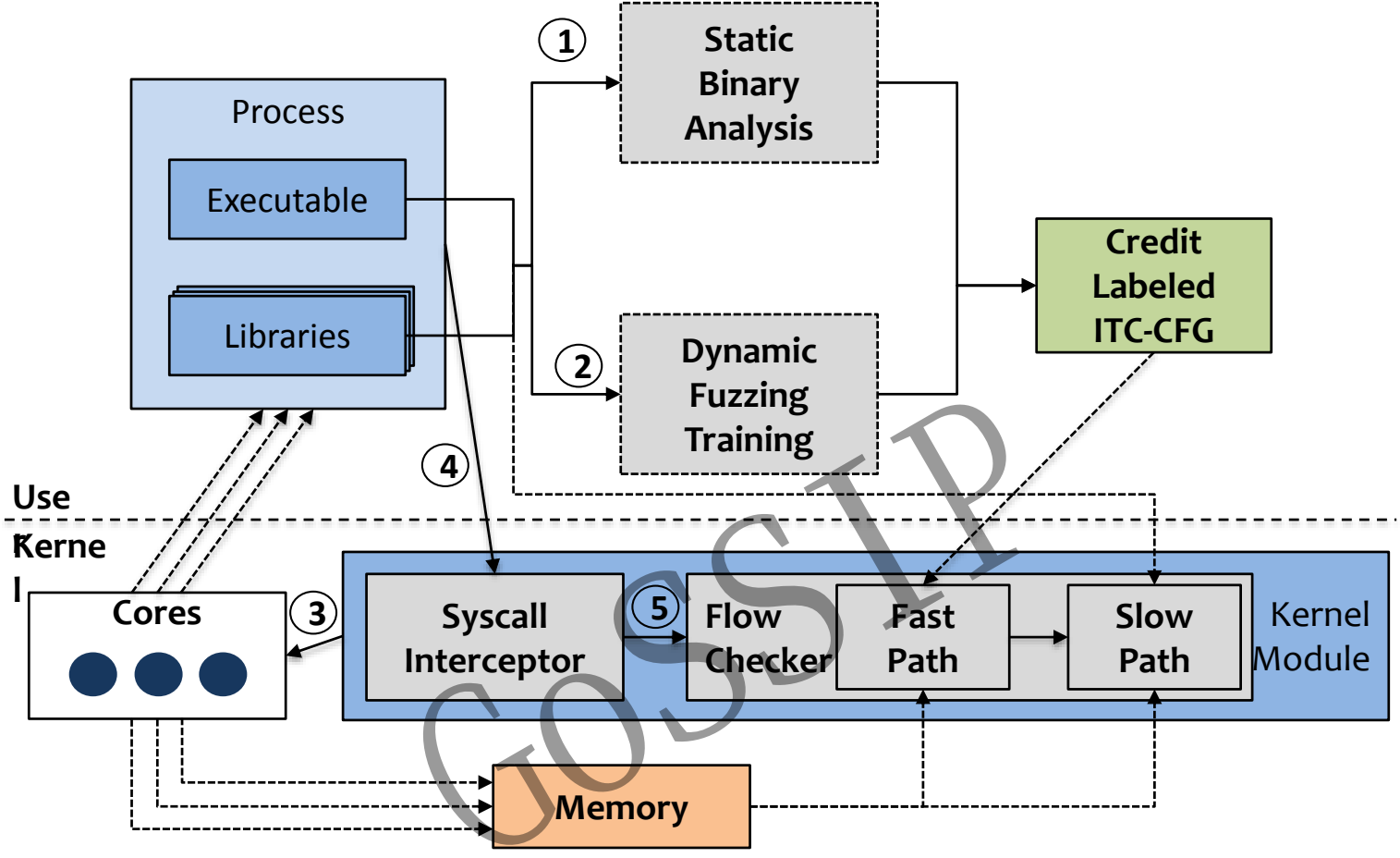
- Performance overhead is shifted from tracing to decoding
  - Decoding is **several orders of magnitude slower** than tracing

	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

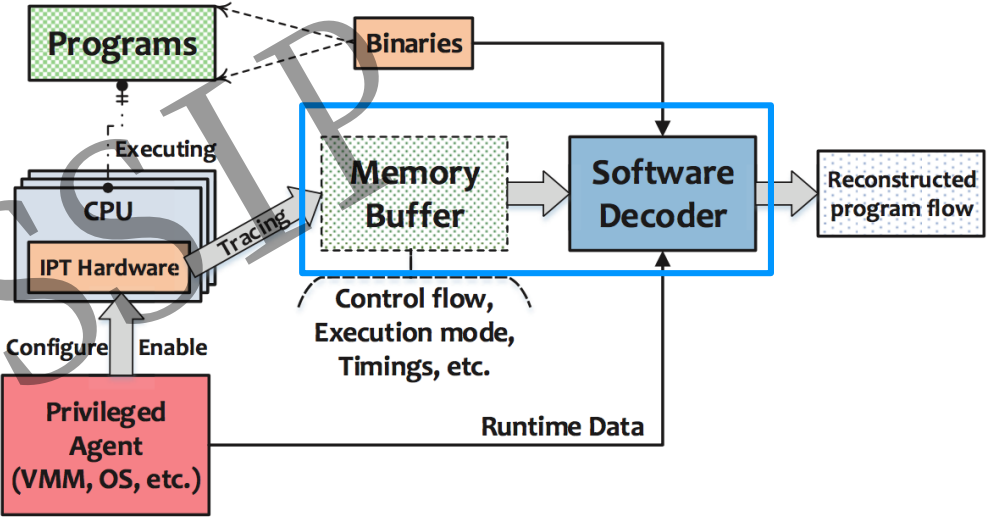
- FlowGuard: transparent, efficient and precise CFI
  - **Transparent**: no source code needed, no hardware change
  - **Precise**: enforce fine-grained CFI with dynamic information
  - **Efficient**: reconstruct CFG and separate fast and slow paths
- Evaluation results
  - Apply FlowGuard to real machine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

FlowGuard Architecture



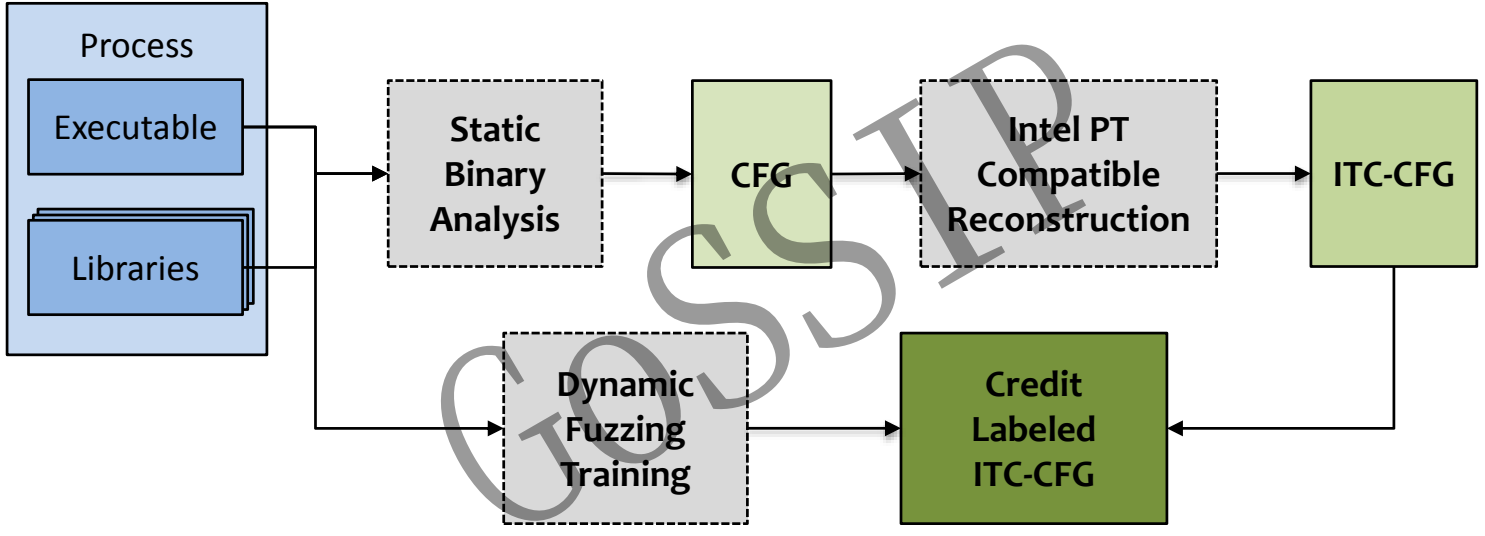
Using IPT for CFI

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



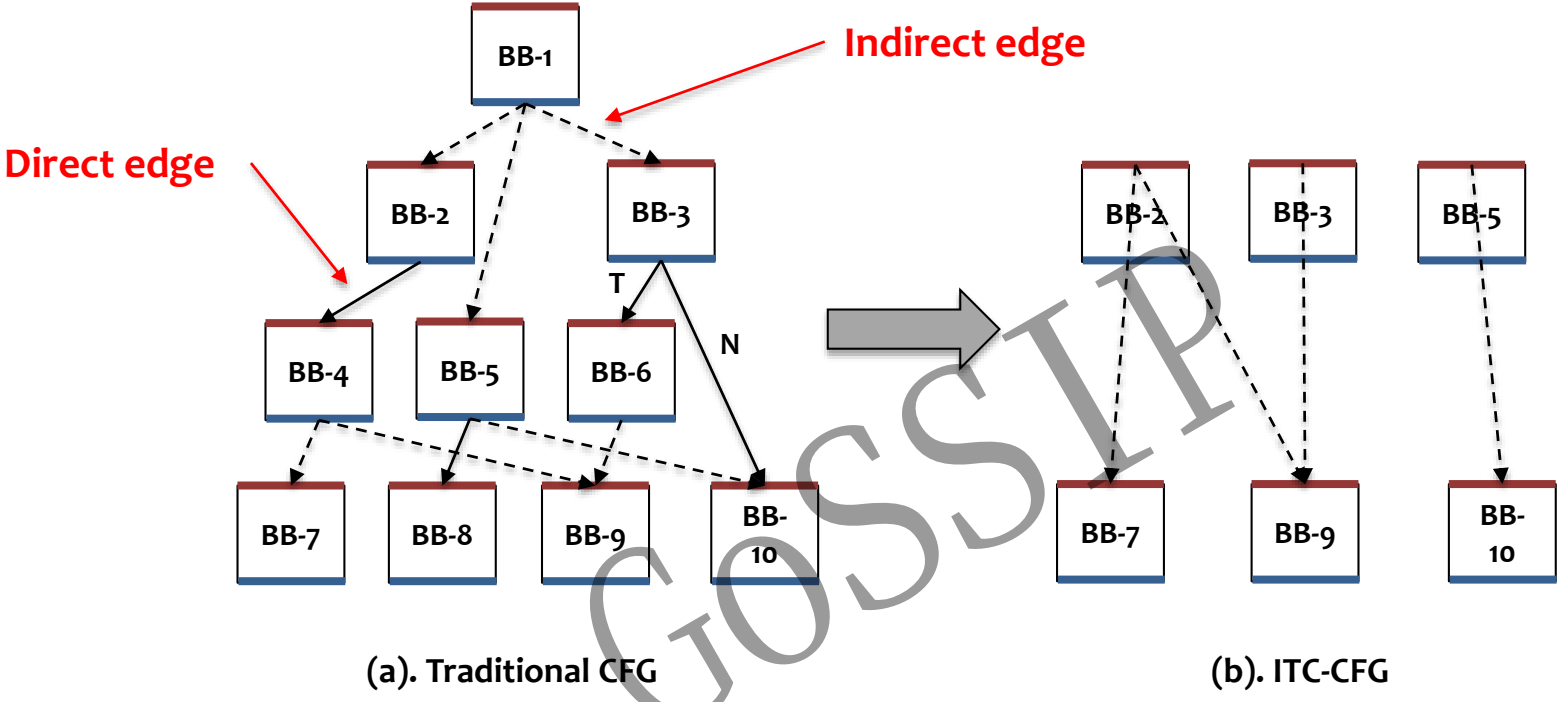
CFG Generation Overview

- 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



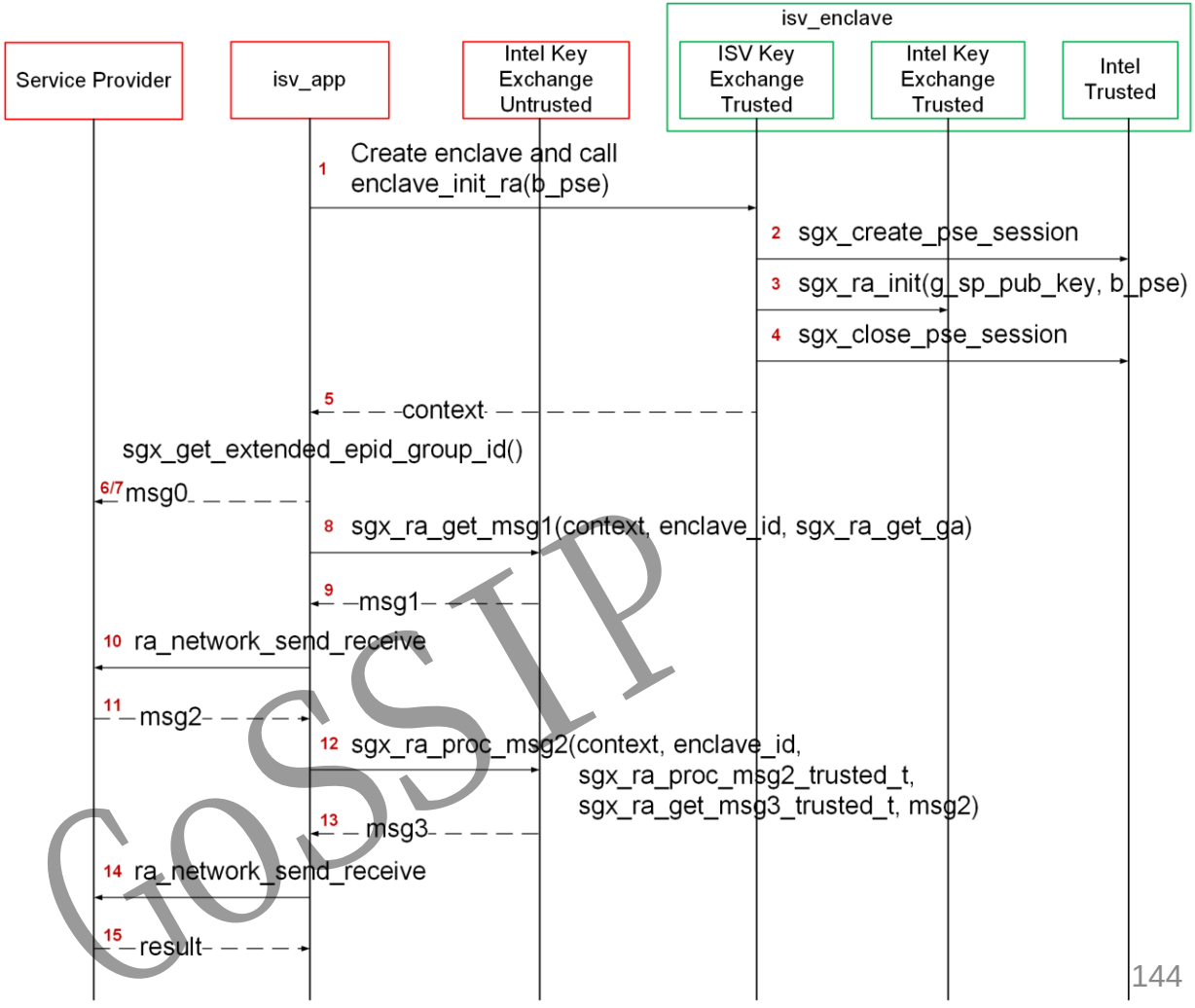
CONCLUSION

GOSSIP

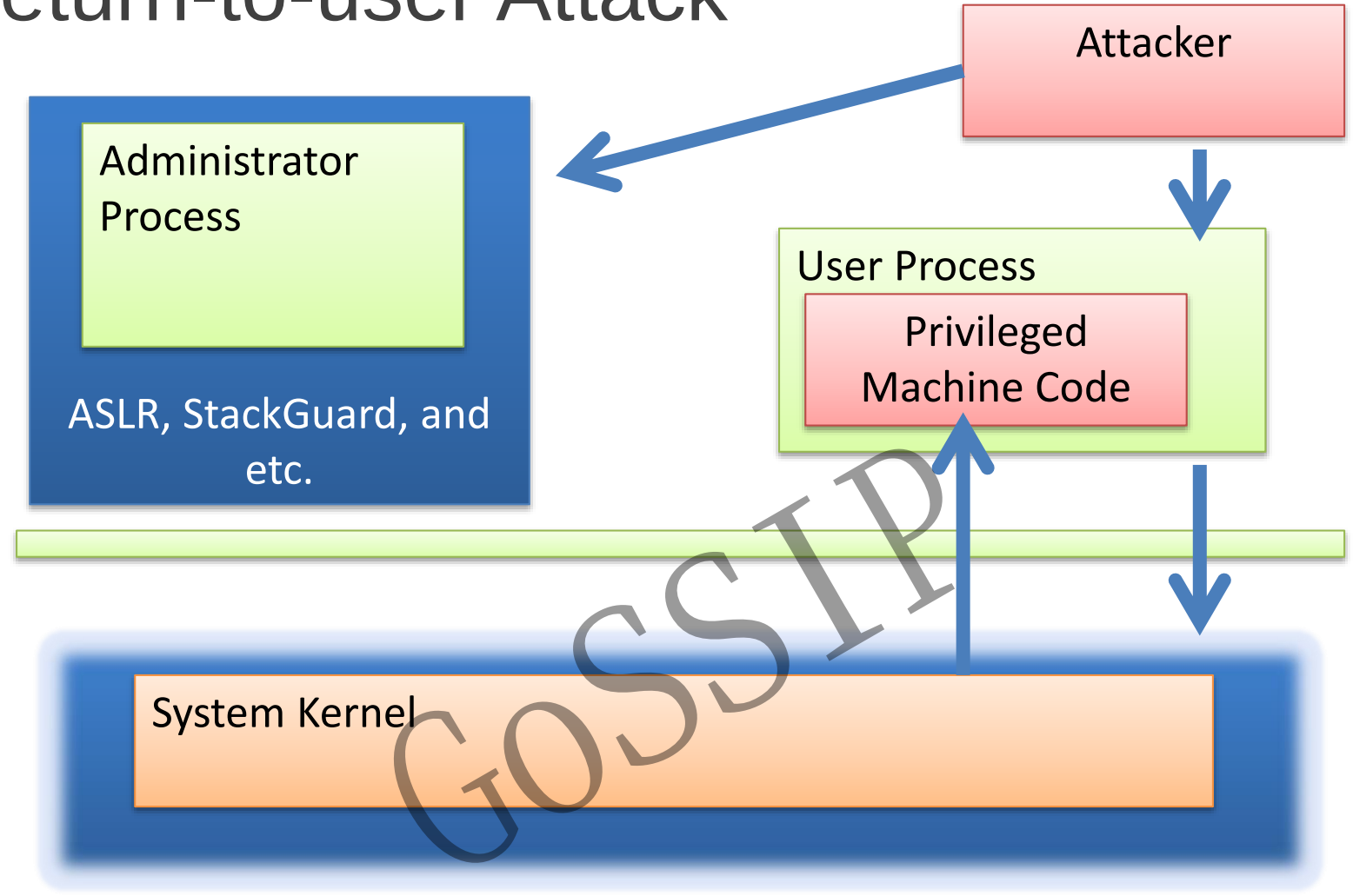
Review: Hardware Features

1	2	3
Features designed for Security	**Isolated Execution Environment**	**Features NOT for Security**
• SMEP/SMAP	• Virtualization	• Intel TSX
• Intel MPX/MPK	• Intel SGX	• Intel CAT
• ARM AP	• AMD SME/SEV	• PMU
• Intel CET	• ARM TrustZone	• Intel PT

THANKS!


isv_enclave
Service Provider
isv_app
Intel Key Exchange Untrusted
ISV Key Exchange Trusted
Intel Key Exchange Trusted
Intel Trusted
1 Create enclave and call enclave_init_ra(b_pse)
2 sgx_create_pse_session
3 sgx_ra_init(g_sp_pub_key, b_pse)
4 sgx_close_pse_session
5 context
sgx_get_extended_epid_group_id()
6/7 msg0
8 sgx_ra_get_msg1(context, enclave_id, sgx_ra_get_ga)
9 msg1
10 ra_network_send_receive
11 msg2
12 sgx_ra_proc_msg2(context, enclave_id, sgx_ra_proc_msg2_trusted_t, sgx_ra_get_msg3_trusted_t, msg2)
13 msg3
14 ra_network_send_receive
15 result

Return-to-user Attack



The FLUSH+RELOAD Attack