SafeHidden: An Efficient and Secure Information Hiding Technique Using Re-randomization

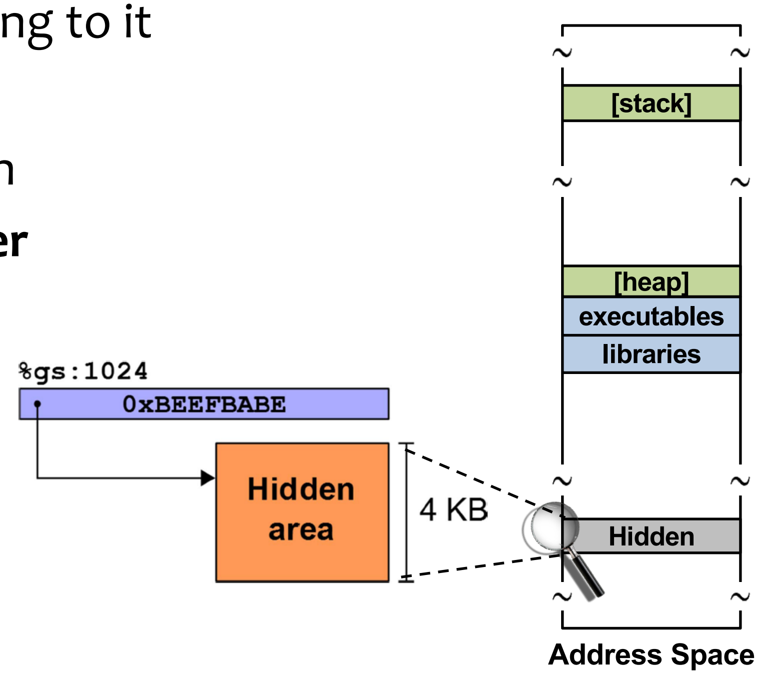
Zhe Wang[1], Chenggang Wu[1], Yinqian Zhang[2], Bowen Tang[1], Pen-Chung Yew[3], Mengyao Xie[1], Yuanming Lai[1], Yan Kang[1], Yueqiang Cheng[4], and Zhiping Shi[5]

[1]Institute of Computing Technology, Chinese Academy of Sciences,
[2]The Ohio State University,
[3]University of Minnesota at Twin-Cities,
[4]Baidu USA,
[5]The Capital Normal University

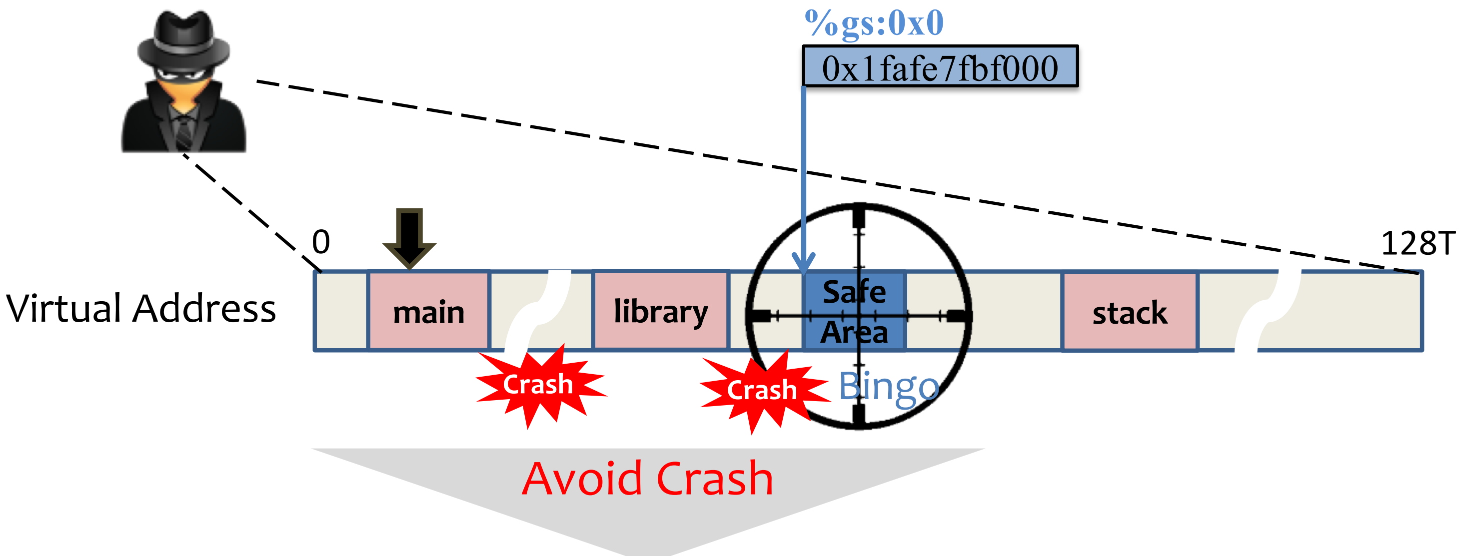
Information Hiding Technique

- **Information Hiding Technique**
  - Hiding an important area at a **random** location
  - Has **no pointers** in memory referring to it
  - Is as **small** as possible
  - Normal accesses are done through **an offset from a dedicated register**

- **It is widely used in**
  - Code Pointer Integrity
  - Control Flow Integrity
  - Code (Re-)Randomization

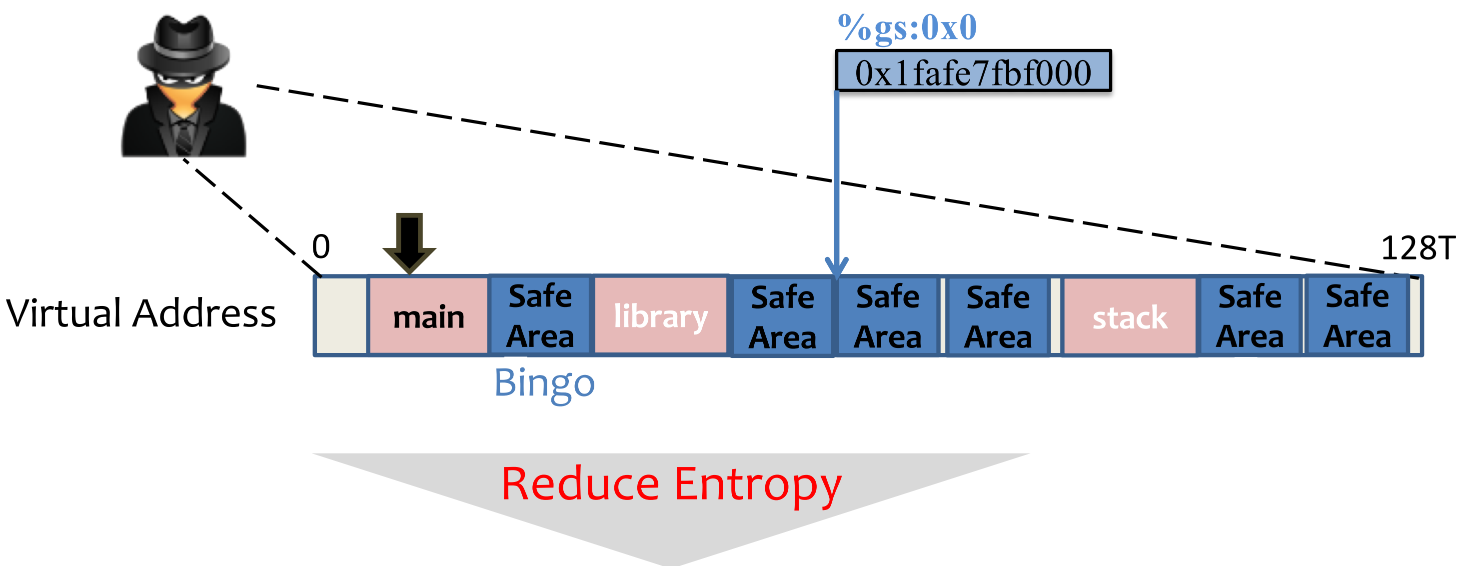


Attacks against Information Hiding



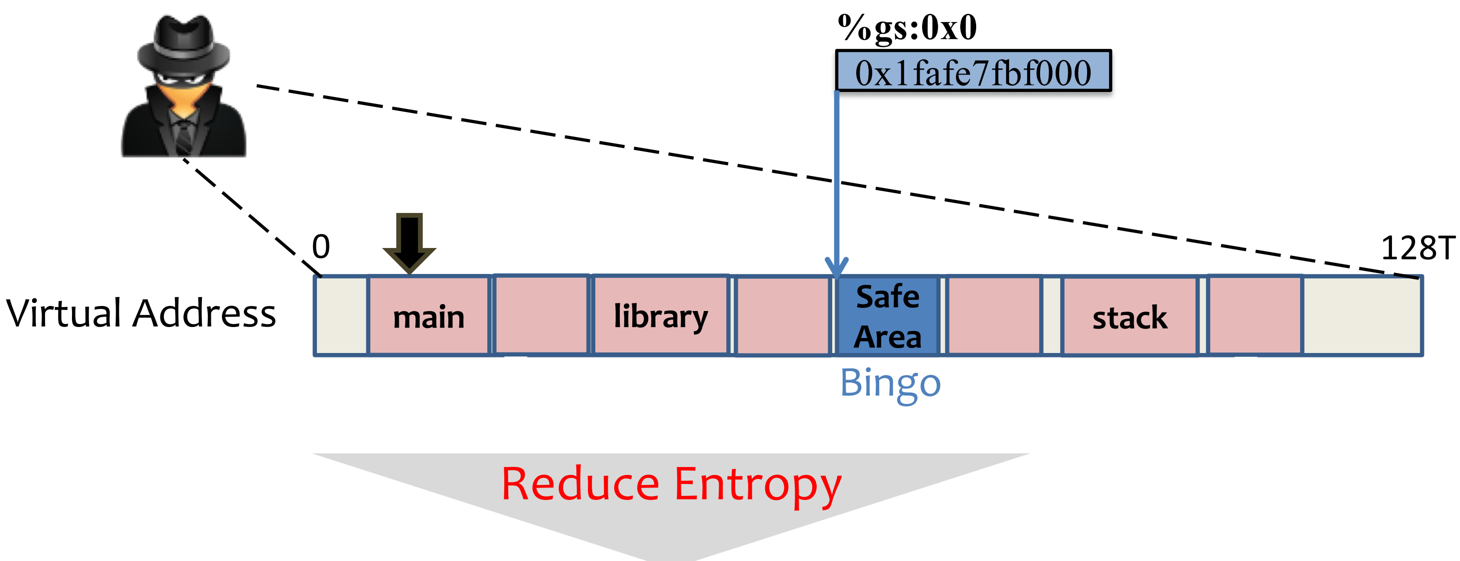
- CROP attack [NDSS'16]
  - Using the exception handling mechanism to avoid crash.
- Clone-probing attack [S&P'14]
  - Probing the child processes to avoid crash the parent process.

Attacks against Information Hiding



- Attack via spraying safe areas [SECURITY'16]
  - Spraying *thread-local* safe areas via spraying threads.

Attacks against Information Hiding



- Attack via spraying safe areas [SECURITY'16]
  - Spraying *thread-local* safe areas via spraying threads.
- Attack via filling memory holes [SECURITY'16]
  - Allocating memory to occupy the unmapped areas.

Attacks against Information Hiding

- Attack against Page Table Structure[NDSS'17]

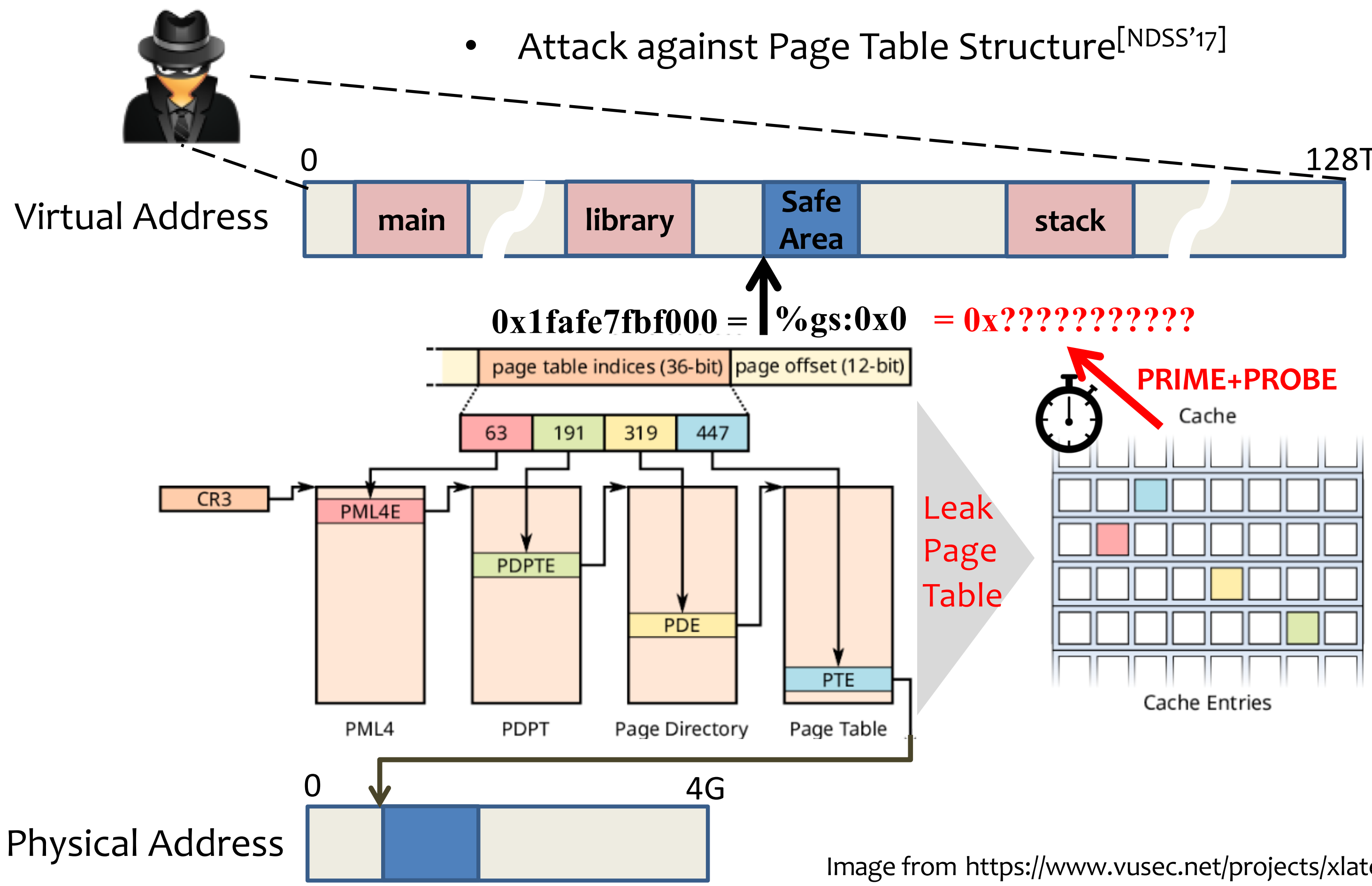


Image from https://www.vusec.net/projects/xlate/

Outline

- **Threat Model**
- Attack vectors
- Our design
- System Implementation
- Evaluation

Threat Model

- **We consider an IH-based defense that protects a vulnerable application against code reuse attacks.**
  - Web servers or browsers.
- **The design of this IH-based defense is not flawed:**
  - Before launching code reuse attacks, attackers must circumvent the defense by revealing the safe area.
- **Attackers' abilities**
  - Read and write arbitrary memory locations;
  - Allocate and free arbitrary memory areas;
  - Create any number of threads;

Attack Vectors —— Summary of Attacks

- **Vector-1** Gathering memory layout information to help to locate safe areas
- **Vector-2** Creating opportunities to probe without crashing the system
- **Vector-3** Reducing the entropy of the randomized safe area locations
- **Vector-4** Monitoring page-table access patterns using cache side channels

Outline

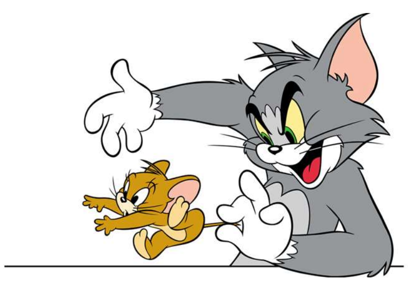
- Threat Model
- Attack vectors
- **Our design**
- System Implementation
- Evaluation

Our Design —— SafeHidden

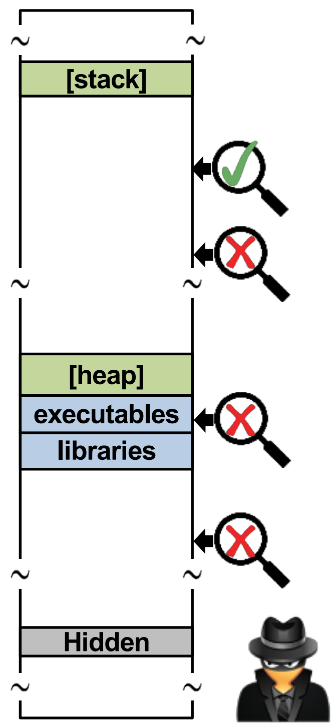
- **SafeHidden is proposed to block these attack vectors**
  - Mediating all types of probes that may leak the locations
  - Randomizing safe areas upon detecting suspicious probes
  - Isolating the thread-local safe areas
  - Raising security alarms when illegal probes are detected

Block Attack Vector-1

- **Vector-1** Gathering memory layout information to help to locate safe areas



But persistent attacks could always succeed.

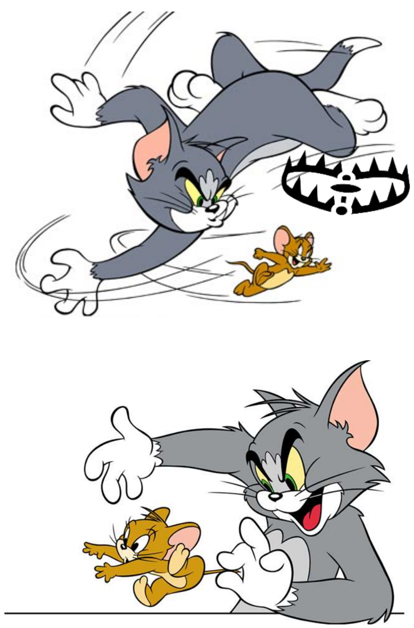


Address Space

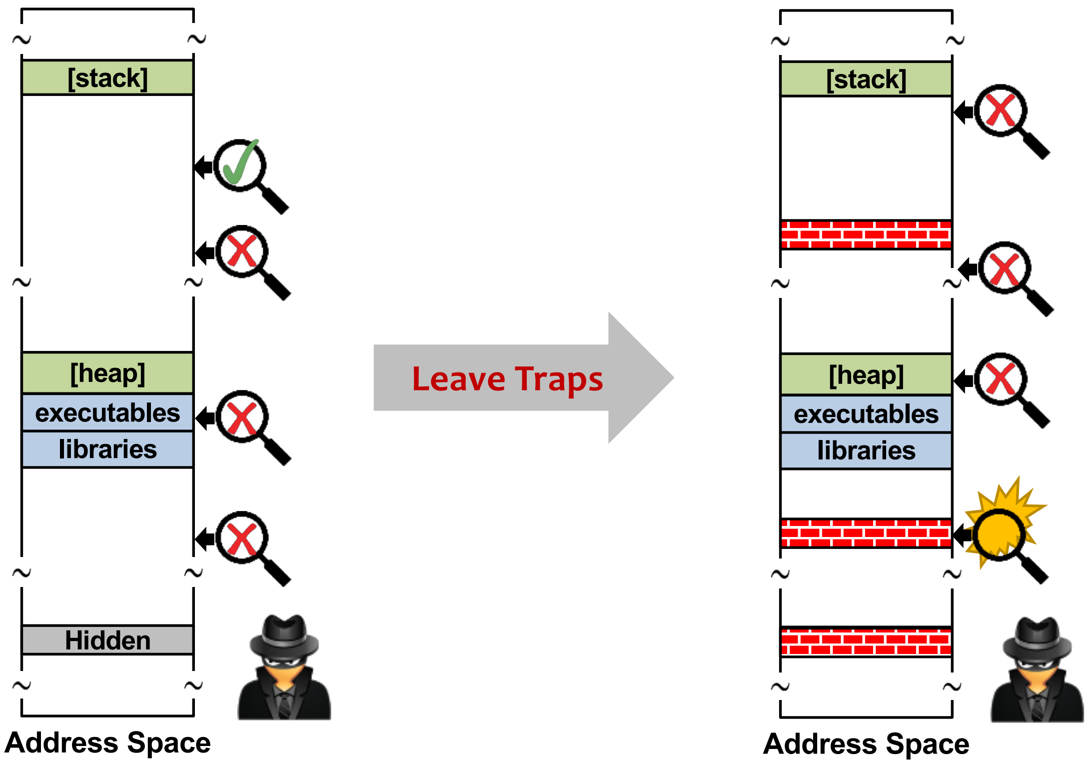
Events	Interception Points
memory management system calls	*mmap, mprotect, brk,…*
Syscalls that could return EFAULT	*read, write, access, send, …*
cloning memory space	*clone, fork, vfork*
memory access instructions	*page fault exception*

Block Attack Vector-2

- **Vector-2** Creating opportunities to probe safe areas without crashing the system



But persistent attacks could always succeed.



Block Attack Vector-3

- **Vector-3** Reducing the entropy of the randomized safe area locations

- **SafeHidden** prevents **unlimited shrink** of unmapped areas and **unrestricted growth** of safe areas.

  – The maximum size of the mapped area is set to **64 TB.**

  – Using thread-private memory mechanism to **isolate** *thread-local* safe areas.
    – The entropy will not be reduced by thread spraying.
    – **Using hardware-assisted virtualization techniques.**
    – Each thread will be assigned a thread-private EPT (Extended Page Table).

More Details are in Our Paper

Block Attack Vector-4

- **Vector-4** Monitoring page-table access patterns using cache side channels

- **Observation**
  - It needs hundreds of Prime+Probe or Evict+Time tests.
  - It is also imperative that the addresses of the PTEs corresponding to this memory area are not changed.

    →The cache entries mapped by these PTEs are not changed.

- **Solution: Re-randomization!**

Block Attack Vector-4

- SafeHidden also monitors legal accesses to the safe area that may be triggered by the attacker on purpose.

- Once such a legal access is detected, SafeHidden will randomize the location of the safe area.

- But, how to detect this legal access from the attacker?

Block Attack Vector-4

- The key step of cache side-channel attack against page table is to force a page table walk.

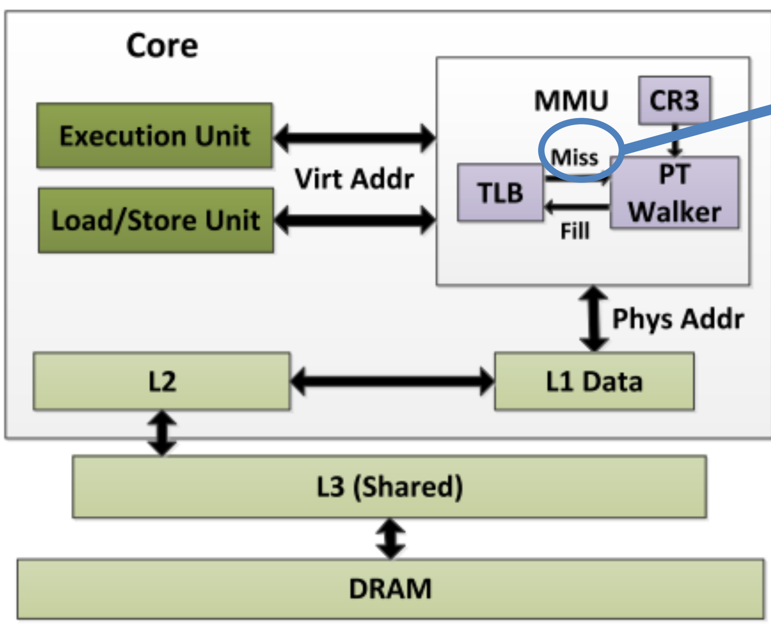
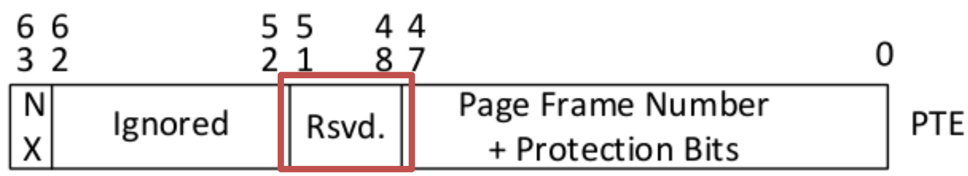


Image from https://www.vusec.net/projects/anc/

We could intercept TLB misses !!!

But, how to only intercept the TLB miss occurred in safe areas?

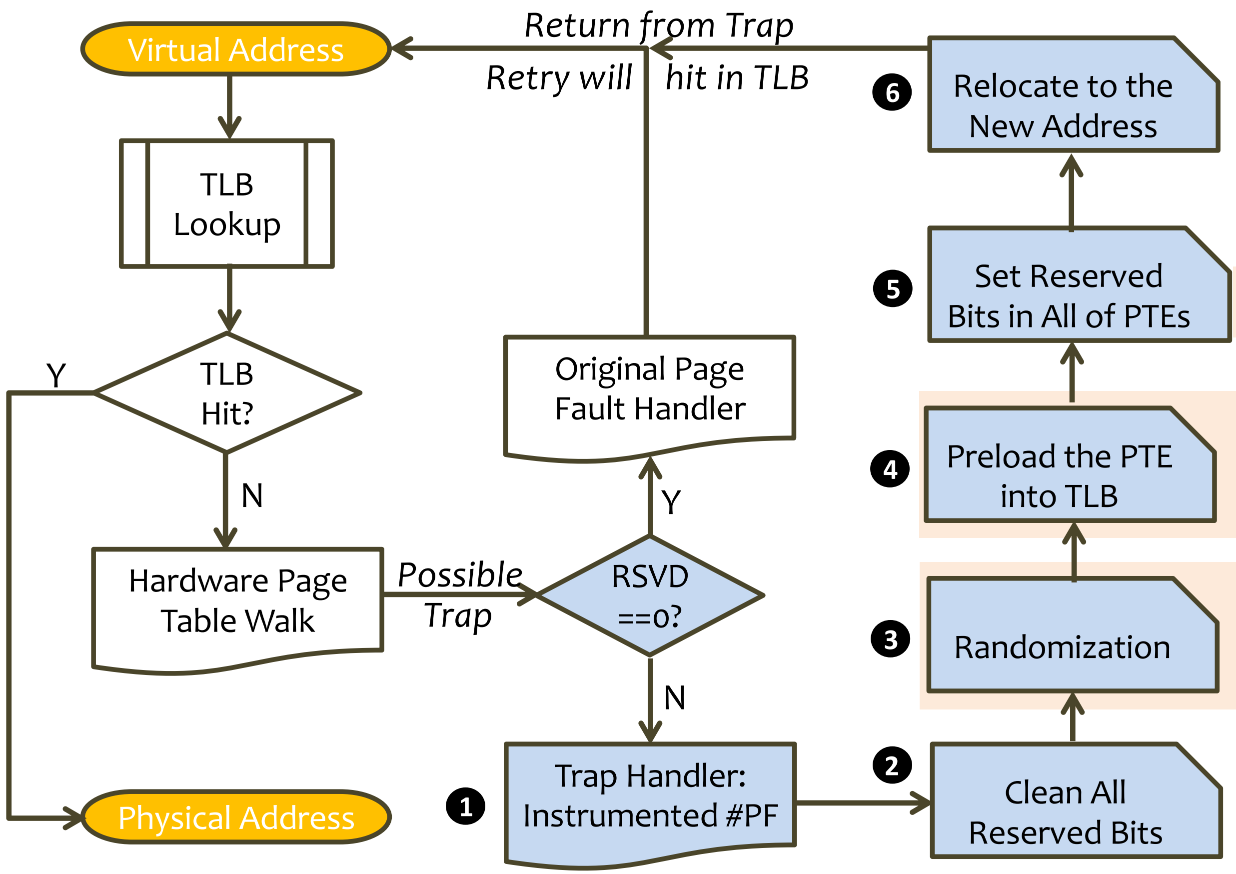
Convert TLB Miss to Page Fault Exception



- When the reserved bit is set, a page fault exception will be triggered during the page table walk.
- SafeHidden sets the reserved bit in all of the PTEs for the safe areas to detect the TLB misses.
  - When a TLB miss occurs, it is trapped into the pf handler.

Flowchart of Page Fault Handler

More Details are in Our Paper



Two Problems

① How to preload PTE into TLB under the KPTI technique?

② It could cause many false alarm TLB misses at new location.

Outline

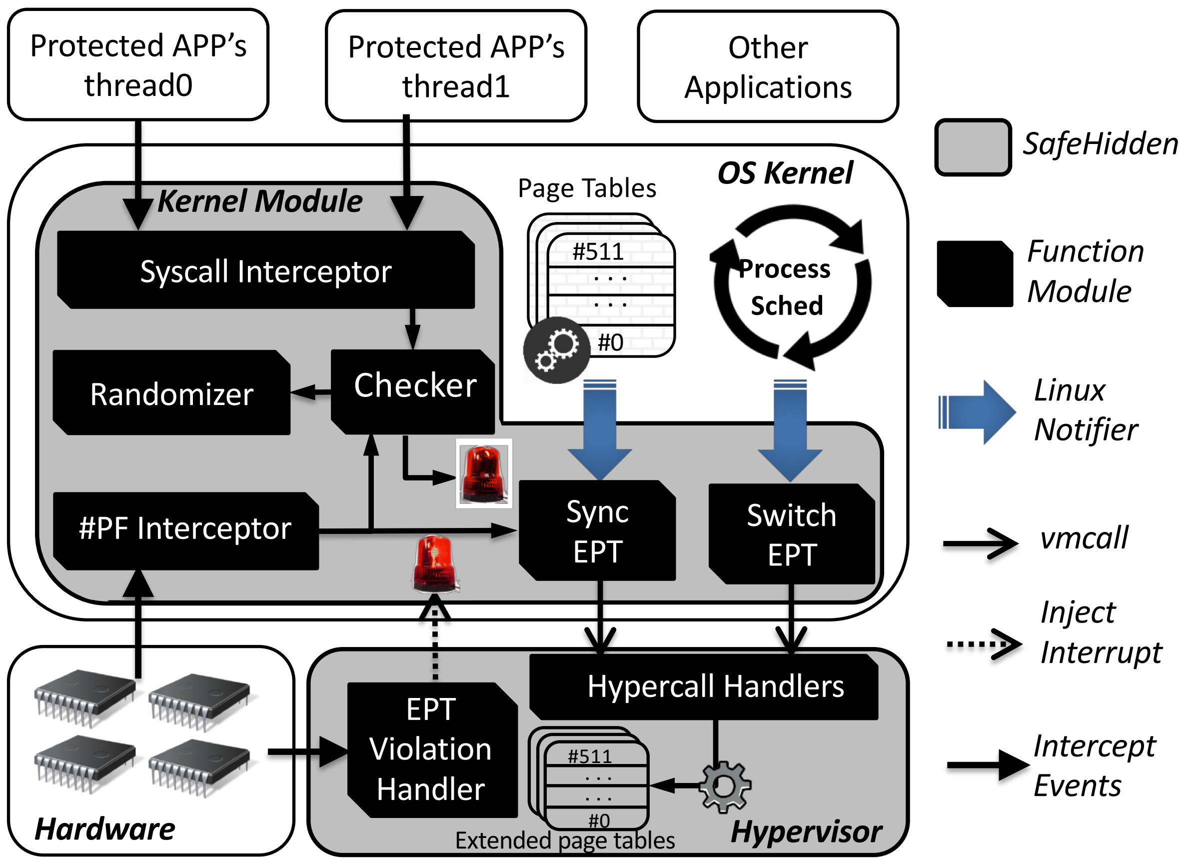
- Threat Model
- Attack vectors
- Our design
- **System Implementation**
- Evaluation

Architecture Overview

- SafeHidden is designed as a loadable kernel module.
  - — No need to modify the existing defenses.
  - — No need to re-compile the OS kernel.

- We integrated a thin hypervisor for a non-virtualized OS.
  - It virtualizes the running OS as the guest without rebooting the system.
  - The other components, called GuestKM, runs in guest kernel.

Architecture Overview



Outline

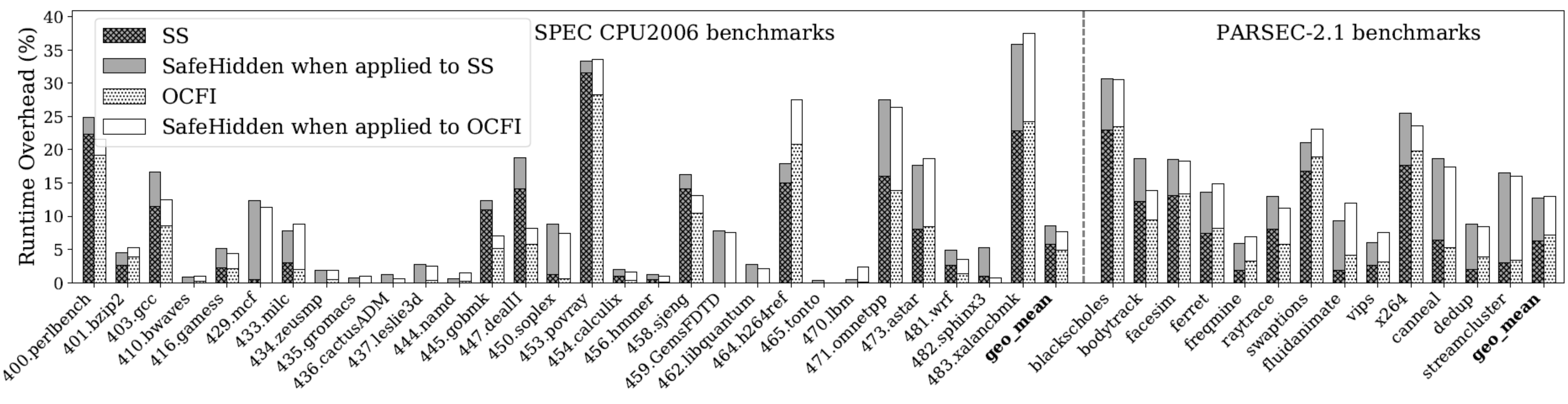
- Threat Model
- Attack vectors
- Our design
- System Implementation
- **Evaluation**

Experiment Setup

- **On X86_64/Linux Platform**
  - 3.4GHZ Intel(R) Core(TM) i7-6700 CPU with 4 cores and 16GB RAM.
  - Ubuntu 18.04 (Kernel 4.20.3 with KPTI enabled by default)
- **SafeHidden protects two defenses that using IH.**
  - Shadow stack and O-CFI.
  - The %gs is used to point to the safe area.
- **Benchmarks**
  - **CPU-intensive benchmarks:** SPEC CPU2006 and Multi-threaded Parsec-2.1.
  - **Network I/O:** Multiple processes Nginx and Multi-threaded Apache.
  - **Disk I/O:** Bonnie++ benchmark tool.

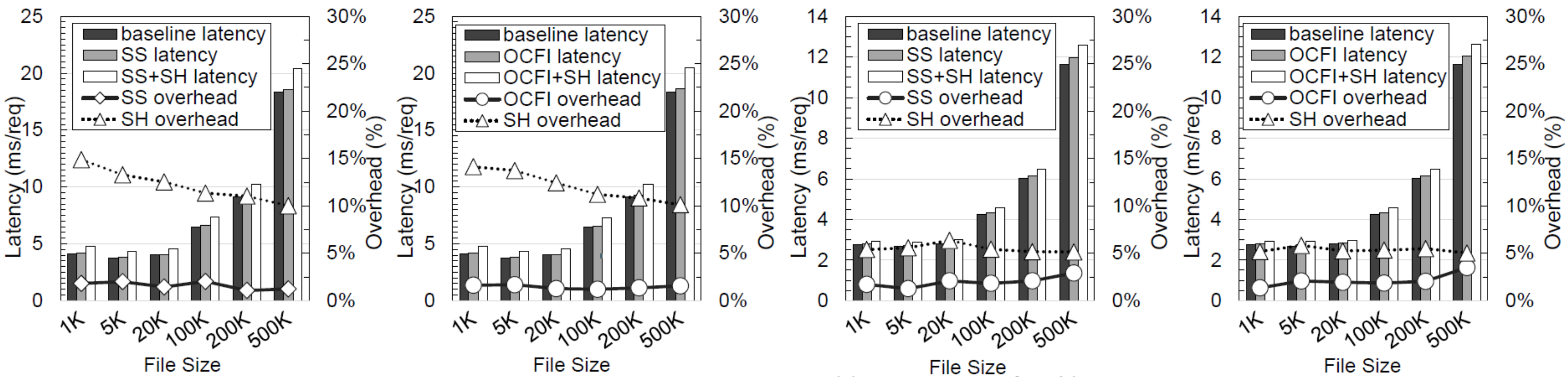
Performance Evaluation

- **CPU-intensive benchmarks**
  - SPEC CPU2006 benchmark with *ref* input
    - Incurred 2.75% and 2.76% when protecting O-CFI and Shadow Stack.
  - Multi-threaded Parsec-2.1 benchmark with *native* input
    - Incurred 5.78% and 6.44% when protecting O-CFI and Shadow Stack.



Performance Evaluation

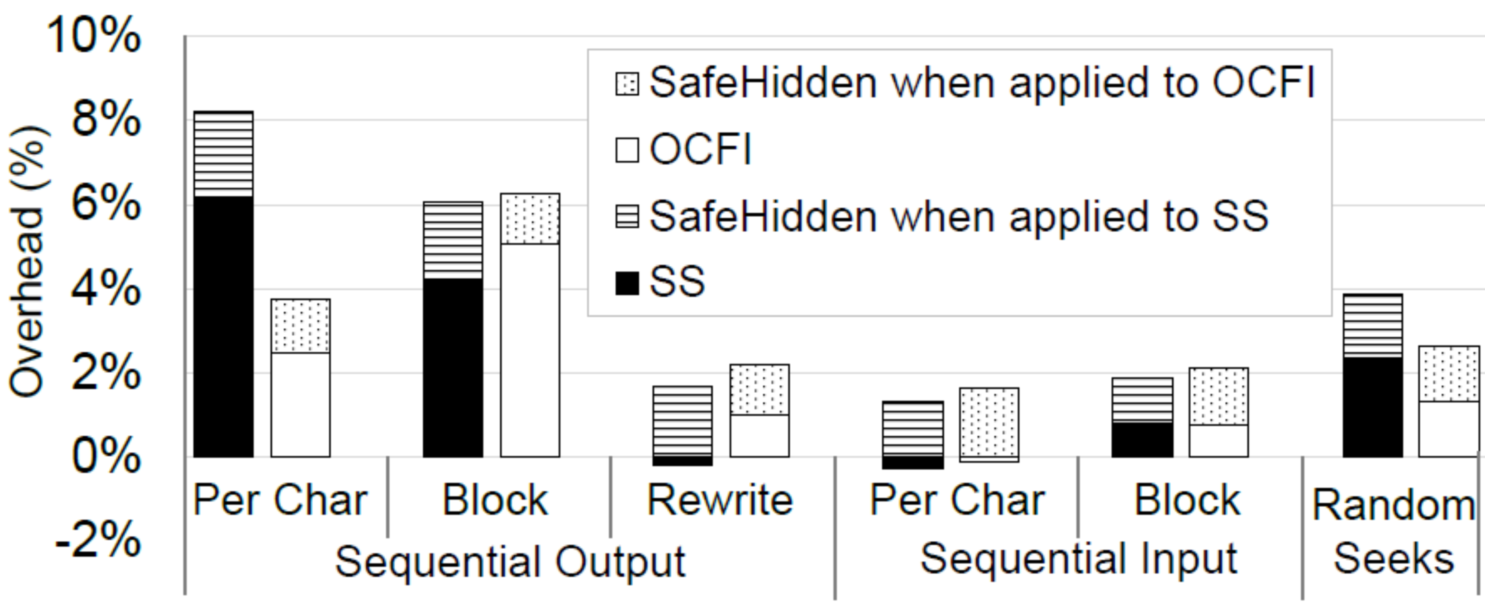
- **Network I/O benchmarks**
  - Apache is configured to work *mpm-worker* mode (8 threads).
    - Incurred 12.07% and 12.18% when protecting O-CFI and Shadow Stack.
  - Nginx is configured to work with 4 worker processes.
    - Incurred 5.35% and 5.51% when protecting O-CFI and Shadow Stack.



(a) Apache + SS + SafeHidden (b) Apache + OCFI + SafeHidden (c) Nginx + SS + SafeHidden (d) Nginx + OCFI + SafeHidden

Performance Evaluation

- **Disk I/O benchmarks**
  - Bonnie++ benchmark tool (read and write tests)
    - Incurred 1.76% and 2.18% when protecting O-CFI and Shadow Stack.



Conclusion

- **SafeHidden proposes the re-randomization based IH technique against all known attacks.**
- **SafeHidden introduces the use of thread-private memory to isolate thread-local safe areas.**
  - Using hardware-assisted extended page tables.
- **It devises a new technique to detect TLB misses.**
  - It is the key trait of cache side-channel attacks against the page tables.

Q & A

wangzhe12@ict.ac.cn

Security Analysis

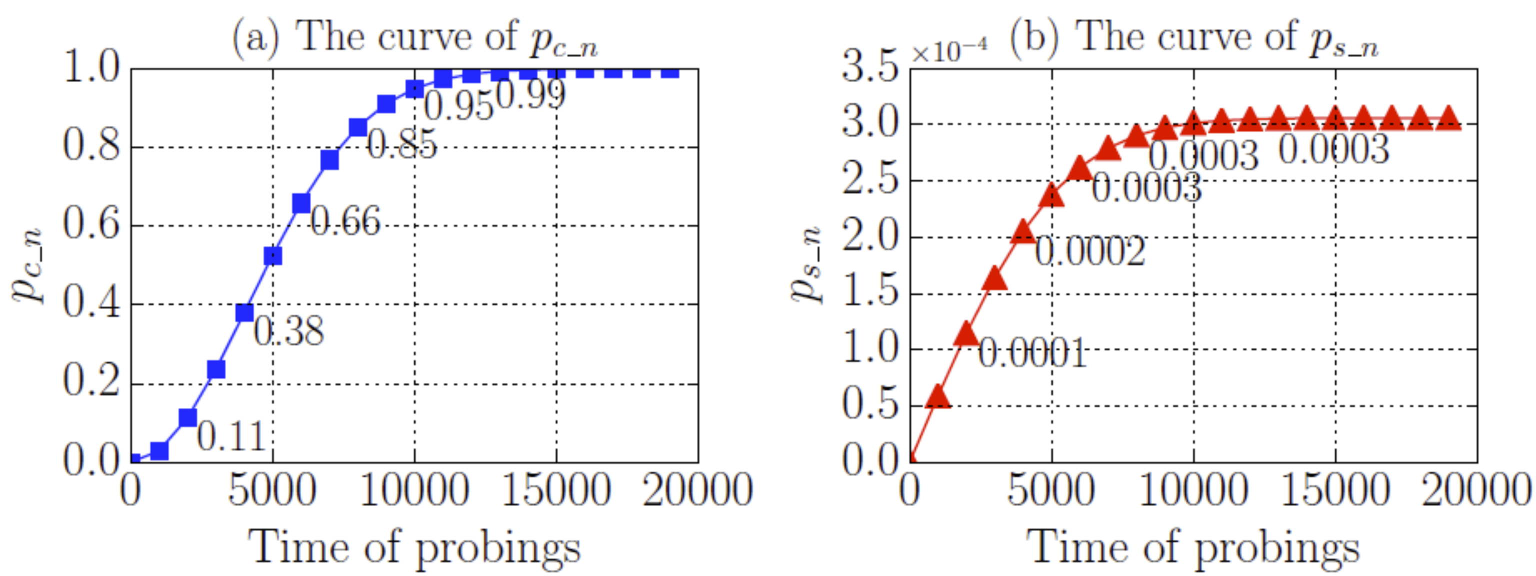
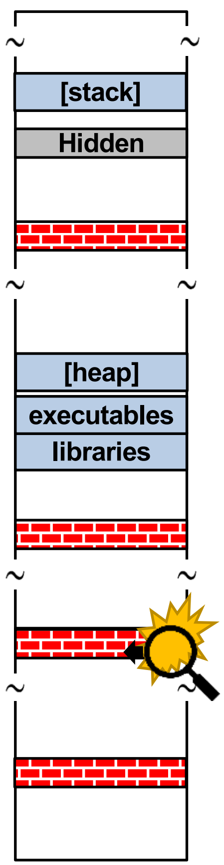


Figure 3: The probability of being captured by SafeHidden within N probes (a) and the probability of locating the safe areas within N probes successfully (b).

When to perform randomization?



Address Space

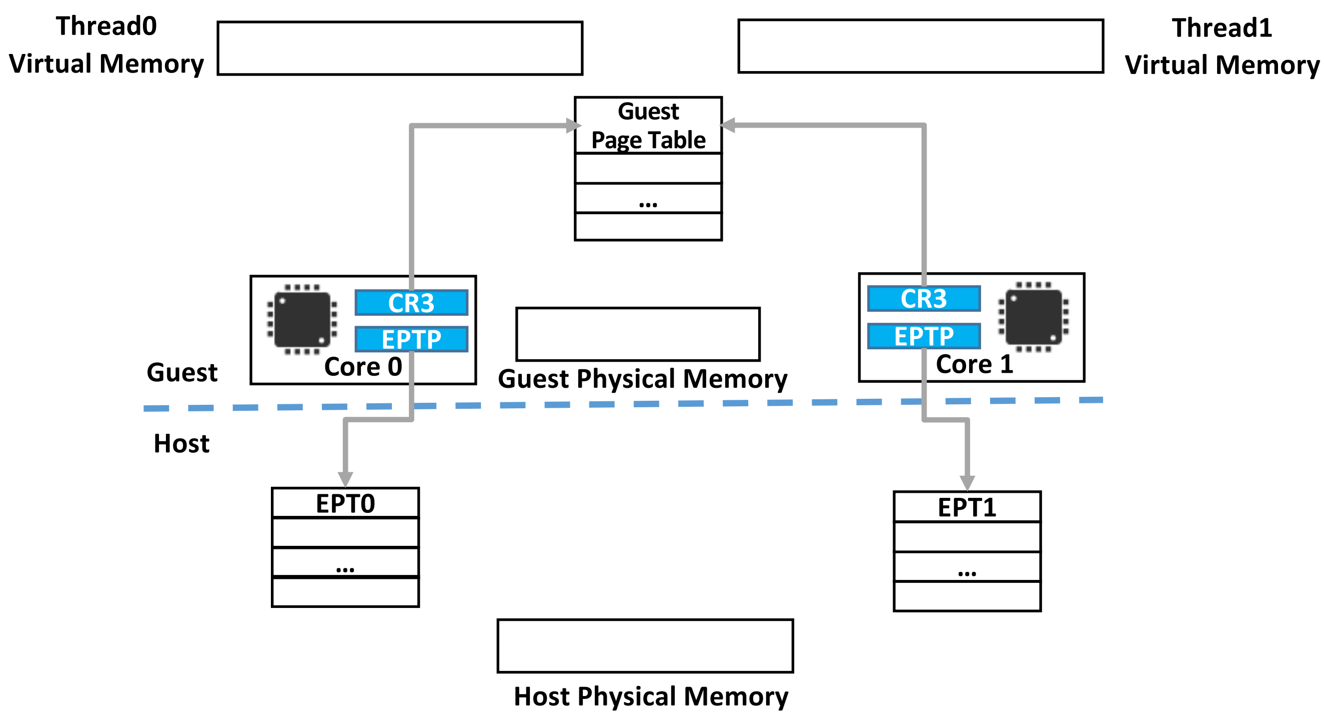
Events	Interception Points
memory management system calls	*mmap, munmap, mremap, mprotect, brk*
syscalls that could return EFAULT	*read, write, access, send, …*
cloning memory space	*clone, fork, vfork*
memory access instructions	*page fault exception*

Other Area (OA) | Trap Area (TA) | Safe Area (SA) | Unmapped Area (UA)

Events	Responses in SafeHidden			
	SA	**UA**	**TA**	**OA**
memory management system calls	Alarm	Rand	Alarm	—
syscalls that could return EFAULT	Alarm	Rand	Alarm	—
cloning memory space	Rand	Rand	Rand	Rand
memory access instructions	—	Rand	Alarm	—

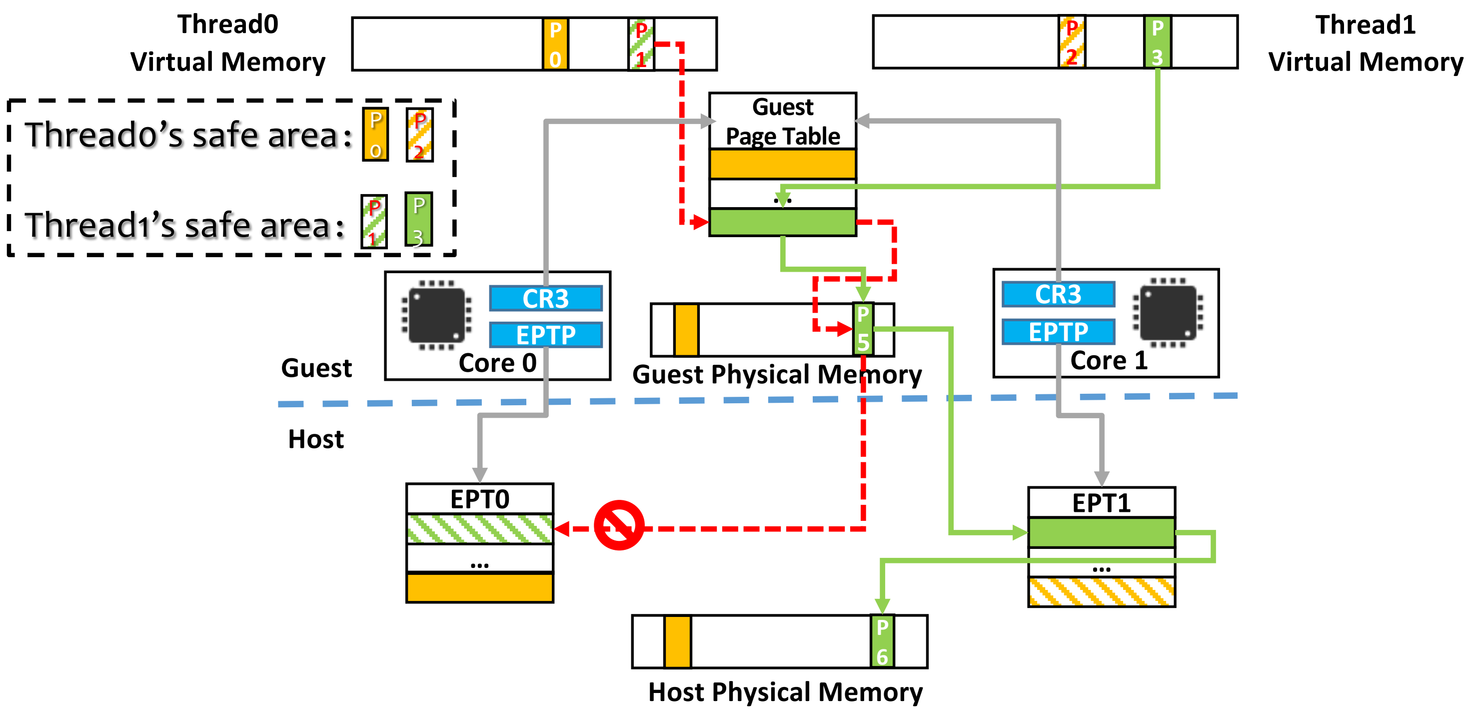
Thread-private Memory

- Instead of using the thread-private page table method, we use a thread-private EPT method to avoid the compatible problem.



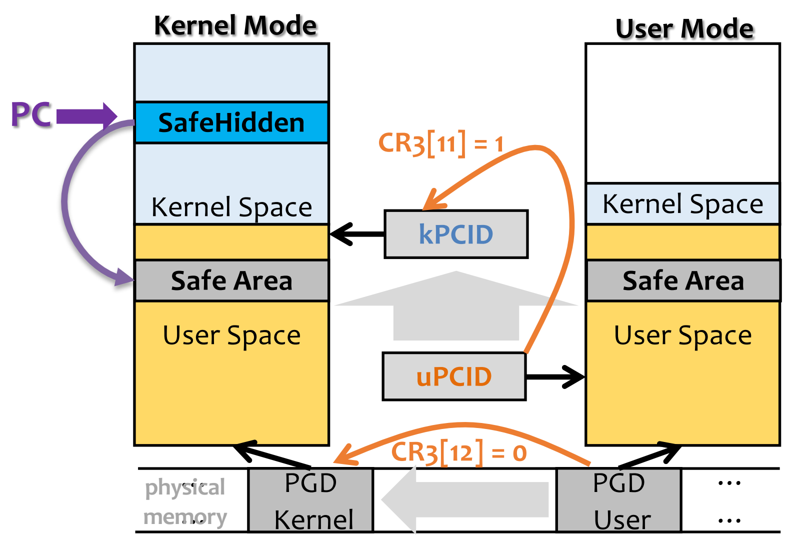
Thread-private Memory

- Instead of using the thread-private page table method, we use a thread-private EPT method to avoid the compatible problem.



How to Integrate SafeHidden with KPTI?

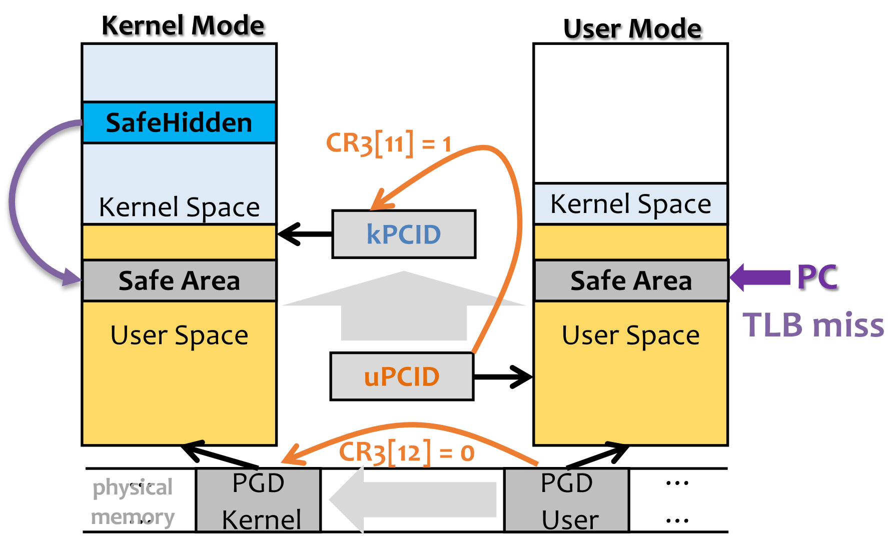
- **KPTI splits the page table for each process into a user-mode page table and a kernel-mode page table.**
  - PCID is used to avoid the TLB flush during context-switch.



TLB	
PCID	VPN->PFN
kPCID	0xsafehidden->0x…
kPCID	0xsafehidden->0x…
kPCID	0xsafehidden->0x…

How to Integrate SafeHidden with KPTI?

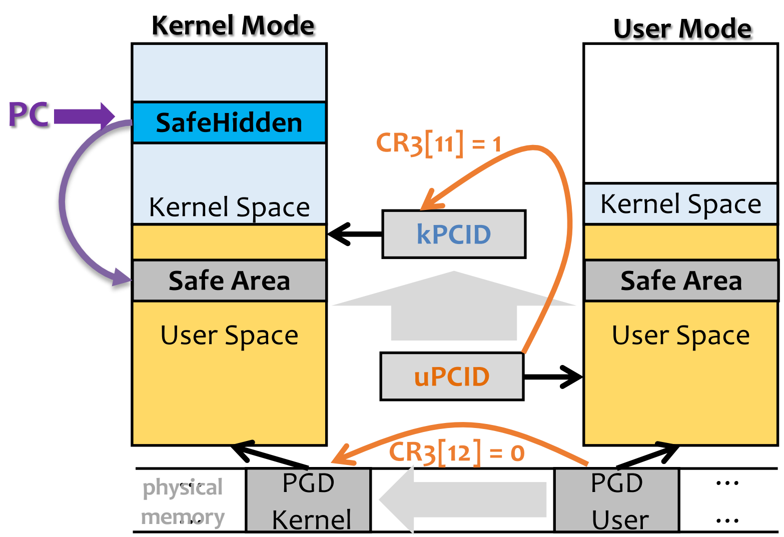
- **The TLB entry loaded in kernel-mode page table with kPCID cannot be used by user-mode code!**



TLB	
PCID	VPN->PFN
kPCID	0xsafehidden->0x…
kPCID	0xsafehidden->0x…
kPCID	0xsafehidden->0x…
kPCID	0xsafearea ->0x…

How to Integrate SafeHidden with KPTI?

- **SafeHidden proposed to bind kernel-mode page table with uPCID temporarily.**

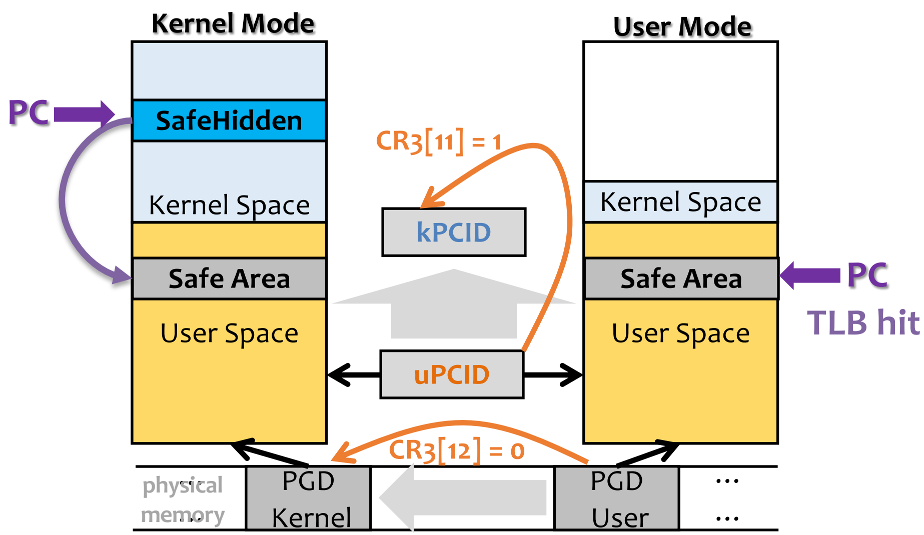


TLB

PCID	VPN->PFN
kPCID	0xsafehidden->0x…
kPCID	0xsafehidden->0x…
kPCID	0xsafehidden->0x…

How to Integrate SafeHidden with KPTI?

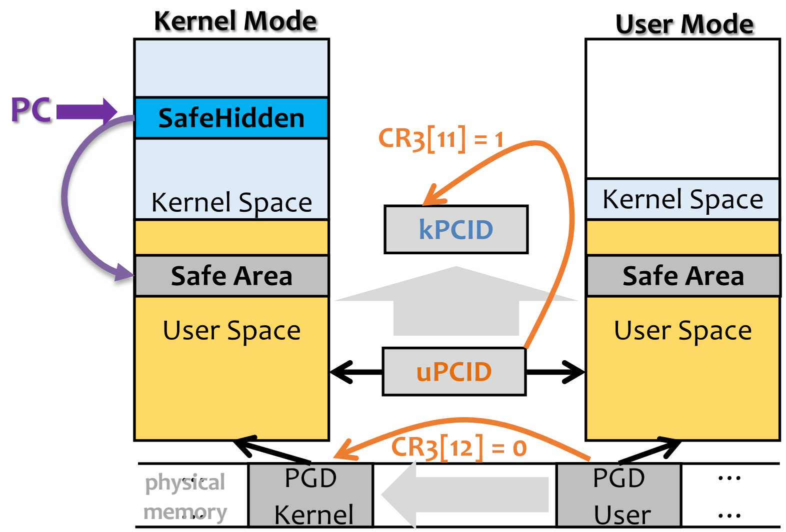
- **SafeHidden proposed to bind kernel-mode page table with uPCID temporarily.**
  – But some pages related to this operation are also loaded.



TLB	
PCID	VPN->PFN
kPCID	0xsafehidden->0x…
kPCID	0xsafehidden->0x…
kPCID	0xsafehidden->0x…
uPCID	0xsafearea ->0x…

How to Integrate SafeHidden with KPTI?

- **SafeHidden proposed to bind kernel-mode page table with uPCID temporarily.**
  - But some pages related to this operation are also loaded.



TLB

PCID	VPN->PFN
kPCID	0xsafehidden->0x…
kPCID	0xsafehidden->0x…
kPCID	0xsafehidden->0x…
uPCID	0xsafearea ->0x…
~~uPCID~~	~~0xsafehidden->0x…~~

To avoid these TLB entries to be exploited by the Meltdown attack, we flush them by using invcpid instructions

Reloading TLB Entries after Randomization

- **SafeHidden uses the Intel TSX to test which PTEs of safe areas are loaded in the TLB.**

- **And then loading them into TLB after randomization to avoid many false alarms of TLB misses.**

When MMU walk a poisoned PTE, it will trigger *#PF, and then captured by Intel TSX.*

```
if _xbegin() == _XBEGIN_STARTED:
    access a page in safe area
    _xend()
else
    fallback routine
```

Abort if it is not in TLB

Information Hiding is Not Secure Any More

- **Recent attacks have made it vulnerable again.**
  - Via breaking the**assumptions** of this technique !!!

- **Rethink the security assumptions of IH :**

1. Failed guesses could crash the program → Avoid crash
2. Safe area is designed very small (high entropy) → Reduce entropy
3. Normal accesses will not leak the location → Leak page table structure