Università degli Studi di Milano
Facoltà di Scienze Matematiche, Fisiche e Naturali

Practical Timing Side Channel Attacks Against Kernel Space ASLR

Ralf Hund, Carsten Willems, Thorsten Holz
Horst-Görtz Institute for IT-Security, Ruhr-University Bochum

Studente: Michele Corrias
Matricola: 808746

Summary

Goal

- generic method to mitigate software exploits
- protect against memory-corruption/control-flow-hijacking attacks
- observation: attacker must know target memory address (e.g., shellcode, ROP gadgets)

Goal

- generic method to mitigate software exploits
- protect against memory-corruption/control-flow-hijacking attacks
- observation: attacker must know target memory address (e.g., shellcode, ROP gadgets)

Address Space Layout Randomization (ASLR)

- randomize base addresses of key areas of the memory
- attacker cannot know code addresses beforehand
- ASLR de facto standard in all modern OS (desktop/server/mobile)

Features

- focus on kernelspace ASLR
- randomize kernel and driver base addresses on each boot
- userspace code has *no* information on kernelspace
- attacker must guess
  - only *one* try
  - each wrong guess induces *system crash*

Contribution

1. new generic-hardware timing-based side channel attack against kernelspace ASLR

Contribution

1. new generic-hardware timing-based side channel attack against kernelspace ASLR
   - both x86 e x86_64 architectures

Contribution

1. new generic-hardware timing-based side channel attack against kernelspace ASLR
   - both x86 e x86_64 architectures
     - Intel i7-870 (Bloomfield, Quad-Core)

Contribution

1. new generic-hardware timing-based side channel attack against kernelspace ASLR
   - both x86 e x86_64 architectures
     - Intel i7-870 (Bloomfield, Quad-Core)
     - Intel i7-950 (Lynnfield, Quad-Core)

Contribution

1. new generic-hardware timing-based side channel attack against kernelspace ASLR
   - both x86 e x86_64 architectures
     - Intel i7-870 (Bloomfield, Quad-Core)
     - Intel i7-950 (Lynnfield, Quad-Core)
     - Intel i7-2600 (Sandybridge, Quad-Core)

Contribution

1. new generic-hardware timing-based side channel attack against kernelspace ASLR
   - both x86 e x86_64 architectures
     - Intel i7-870 (Bloomfield, Quad-Core)
     - Intel i7-950 (Lynnfield, Quad-Core)
     - Intel i7-2600 (Sandybridge, Quad-Core)
     - AMD Athlon II X3 455 (Triple-Core)

Contribution

1. new generic-hardware timing-based side channel attack against kernelspace ASLR
   - both x86 e x86_64 architectures
     - Intel i7-870 (Bloomfield, Quad-Core)
     - Intel i7-950 (Lynnfield, Quad-Core)
     - Intel i7-2600 (Sandybridge, Quad-Core)
     - AMD Athlon II X3 455 (Triple-Core)
   - different OS (Windows 7, Linux)

Contribution

1. new generic-hardware timing-based side channel attack against kernelspace ASLR
   - both x86 e x86_64 architectures
     - Intel i7-870 (Bloomfield, Quad-Core)
     - Intel i7-950 (Lynnfield, Quad-Core)
     - Intel i7-2600 (Sandybridge, Quad-Core)
     - AMD Athlon II X3 455 (Triple-Core)
   - different OS (Windows 7, Linux)
   - also within virtual machines (VMWare Player 4.0.2 on Intel i7-870)

Contribution

1. new generic-hardware timing-based side channel attack against kernelspace ASLR
   - both x86 e x86_64 architectures
     - Intel i7-870 (Bloomfield, Quad-Core)
     - Intel i7-950 (Lynnfield, Quad-Core)
     - Intel i7-2600 (Sandybridge, Quad-Core)
     - AMD Athlon II X3 455 (Triple-Core)
   - different OS (Windows 7, Linux)
   - also within virtual machines (VMWare Player 4.0.2 on Intel i7-870)
   - 3 different implementations of attack

Contribution

1. new generic-hardware timing-based side channel attack against kernelspace ASLR
   - both x86 e x86_64 architectures
     - Intel i7-870 (Bloomfield, Quad-Core)
     - Intel i7-950 (Lynnfield, Quad-Core)
     - Intel i7-2600 (Sandybridge, Quad-Core)
     - AMD Athlon II X3 455 (Triple-Core)
   - different OS (Windows 7, Linux)
   - also within virtual machines (VMWare Player 4.0.2 on Intel i7-870)
   - 3 different implementations of attack
2. reverse-engineered Windows kernelspace ASLR

Contribution

1. new generic-hardware timing-based side channel attack against kernelspace ASLR
   - both x86 e x86_64 architectures
     - Intel i7-870 (Bloomfield, Quad-Core)
     - Intel i7-950 (Lynnfield, Quad-Core)
     - Intel i7-2600 (Sandybridge, Quad-Core)
     - AMD Athlon II X3 455 (Triple-Core)
   - different OS (Windows 7, Linux)
   - also within virtual machines (VMWare Player 4.0.2 on Intel i7-870)
   - 3 different implementations of attack
2. reverse-engineered Windows kernelspace ASLR
3. several mitigation solutions

Windows Kernelspace ASLR

Details

- implemented since Windows Vista
- 2 different module randomizations
  - base addresses of *kernel image* and *Hardware Abstraction Layer* (*HAL*)
  - base address of drivers
- randomization of at least 5 bits of base address

Kernel Image

- kernel image (*ntoskrnl.exe*) and *hal.sys* driver loaded adjacently

Kernel Image

- kernel image (*ntoskrnl.exe*) and *hal.sys* driver loaded adjacently
  1. at boot the loader allocates a sufficiently large address region (*kernel_region*) whose base address is costant

Kernel Image

- kernel image (*ntoskrnl.exe*) and *hal.sys* driver loaded adjacently
  1. at boot the loader allocates a sufficiently large address region (*kernel_region*) whose base address is costant
  2. random $r \in \{0 \dots 31\} * 0x1000$ (page size) $\rightarrow$ random base slot

Kernel Image

- kernel image (*ntoskrnl.exe*) and *hal.sys* driver loaded adjacently
  1. at boot the loader allocates a sufficiently large address region (*kernel_region*) whose base address is costant
  2. random $r \in \{0 \dots 31\} * 0x1000$ (page size) $\rightarrow$ random base slot
  3. random load order of kernel image and HAL

Kernel Image

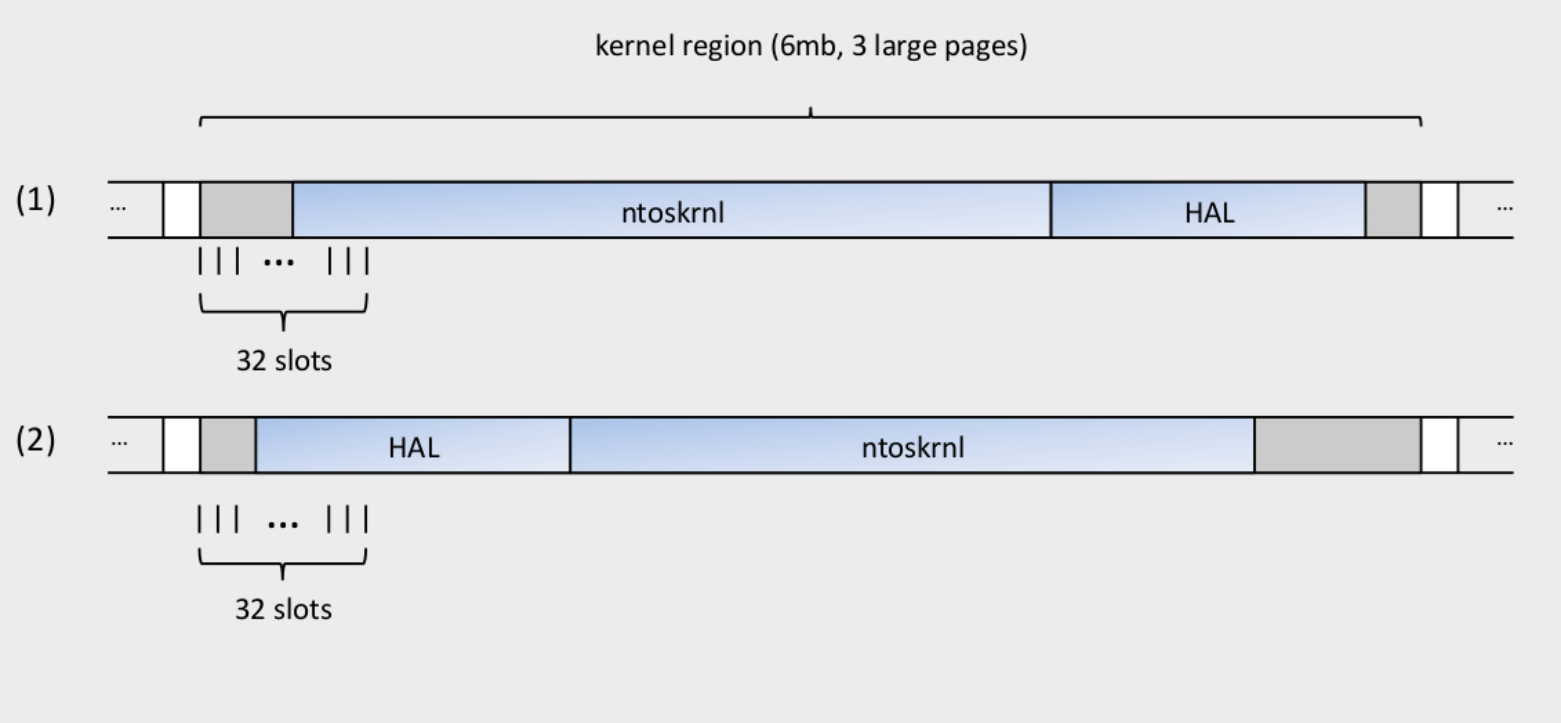
- kernel image (*ntoskrnl.exe*) and *hal.sys* driver loaded adjacently
  1. at boot the loader allocates a sufficiently large address region (*kernel_region*) whose base address is costant
  2. random $r \in \{0 \dots 31\} * 0x1000$ (page size) $\rightarrow$ random base slot
  3. random load order of kernel image and HAL
- *64* different slots to which kernel image and HAL each can be loaded

Windows Kernelspace ASLR

Kernel Image

- kernel image (*ntoskrnl.exe*) and *hal.sys* driver loaded adjacently
  1. at boot the loader allocates a sufficiently large address region (*kernel_region*) whose base address is costant
  2. random $r \in \{0 \dots 31\} * 0x1000$ (page size) $\rightarrow$ random base slot
  3. random load order of kernel image and HAL
- *64* different slots to which kernel image and HAL each can be loaded
- n.b.: kernel & HAL mapped in *large pages* (2MB) for performance. Both require 3 ones (6MB)

Windows Kernelspace ASLR



Kernel Base Address

$$k\_base = kernel\_region + (r * 0x1000) + (p * HALsize)$$

- $r \in \{0 \dots 31\}$
- $p \in \{0, 1\}$

Virtual Addresses

- randomization already applied to physical load addresses of the kernel image:

$$VA = 0x80000000 + PA$$

Windows Kernelspace ASLR

Virtual Addresses

- randomization already applied to physical load addresses of the kernel image:

$$VA = 0x80000000 + PA$$

- lower 31 bits of virtual kernel addresses are identical to the physical address.

Windows Kernelspace ASLR

Virtual Addresses

- randomization already applied to physical load addresses of the kernel image:

$$VA = 0x80000000 + PA$$

- lower 31 bits of virtual kernel addresses are identical to the physical address.
- only true for addresses in the kernel_region.

Windows Kernelspace ASLR

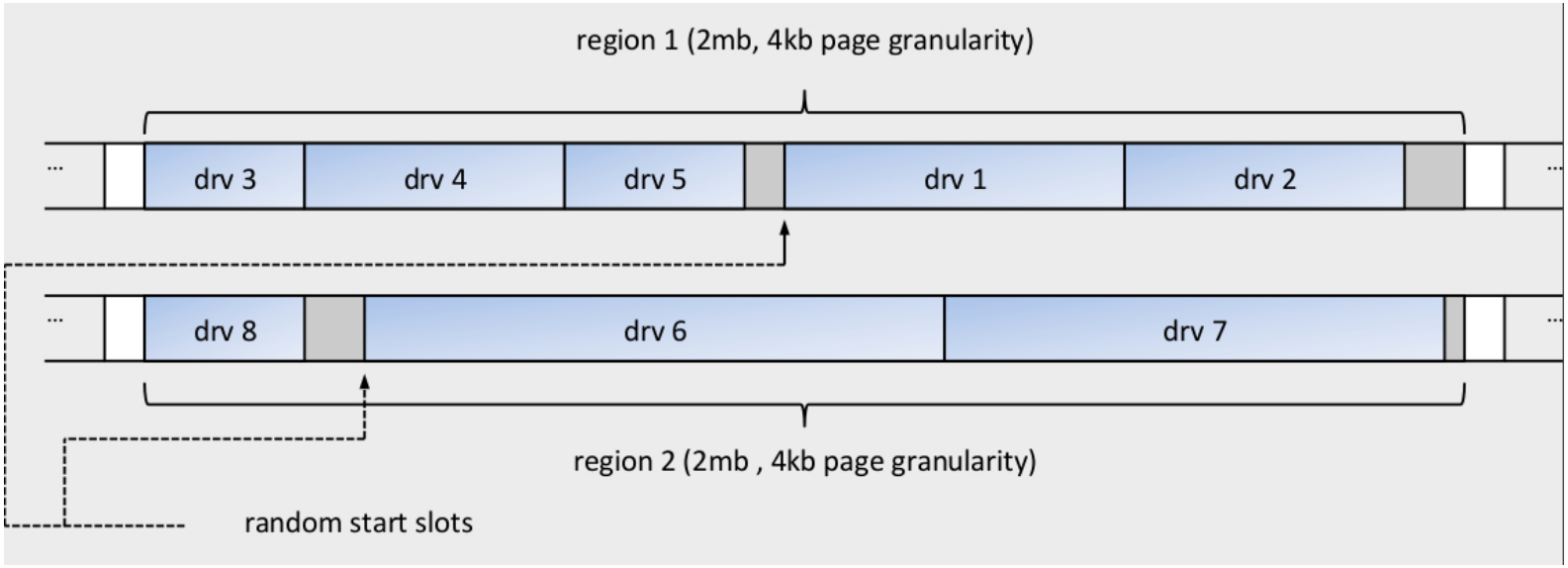
Drivers

- driver modules loaded similarly
- allocate 2MB *driver_region*
- choose randomly one of 64 start slots in region
- append subsequent drivers
- region full? new *driver_region* with random start slot allocated

Windows Kernelspace ASLR

Drivers

- driver modules loaded similarly
- allocate 2MB *driver_region*
- choose randomly one of 64 start slots in region
- append subsequent drivers
- region full? new *driver_region* with random start slot allocated



Memory Hierarchy

Cache

- speed up memory access for code and data
- for each CPU core
  1. *Level 1* (L1) cache, split into *ICACHE* (instructions) and *DCACHE* (data)
  2. *Level 2* (L2) cache
  3. *Level 3* (L3) cache
- cached chunks of bytes: a *cacheline* is 64 bytes

Memory Hierarchy

Cache

- speed up memory access for code and data
- for each CPU core
  1. *Level 1* (L1) cache, split into *ICACHE* (instructions) and *DCACHE* (data)
  2. *Level 2* (L2) cache
  3. *Level 3* (L3) cache
- cached chunks of bytes: a *cacheline* is 64 bytes

Associativity

- caches operate in *n-way set associative mode*
- slots are grouped into sets of size n
- each memory chunk can be stored in all slots of a particular set
- target set determined by *cache index* bits

Memory Hierarchy

Example

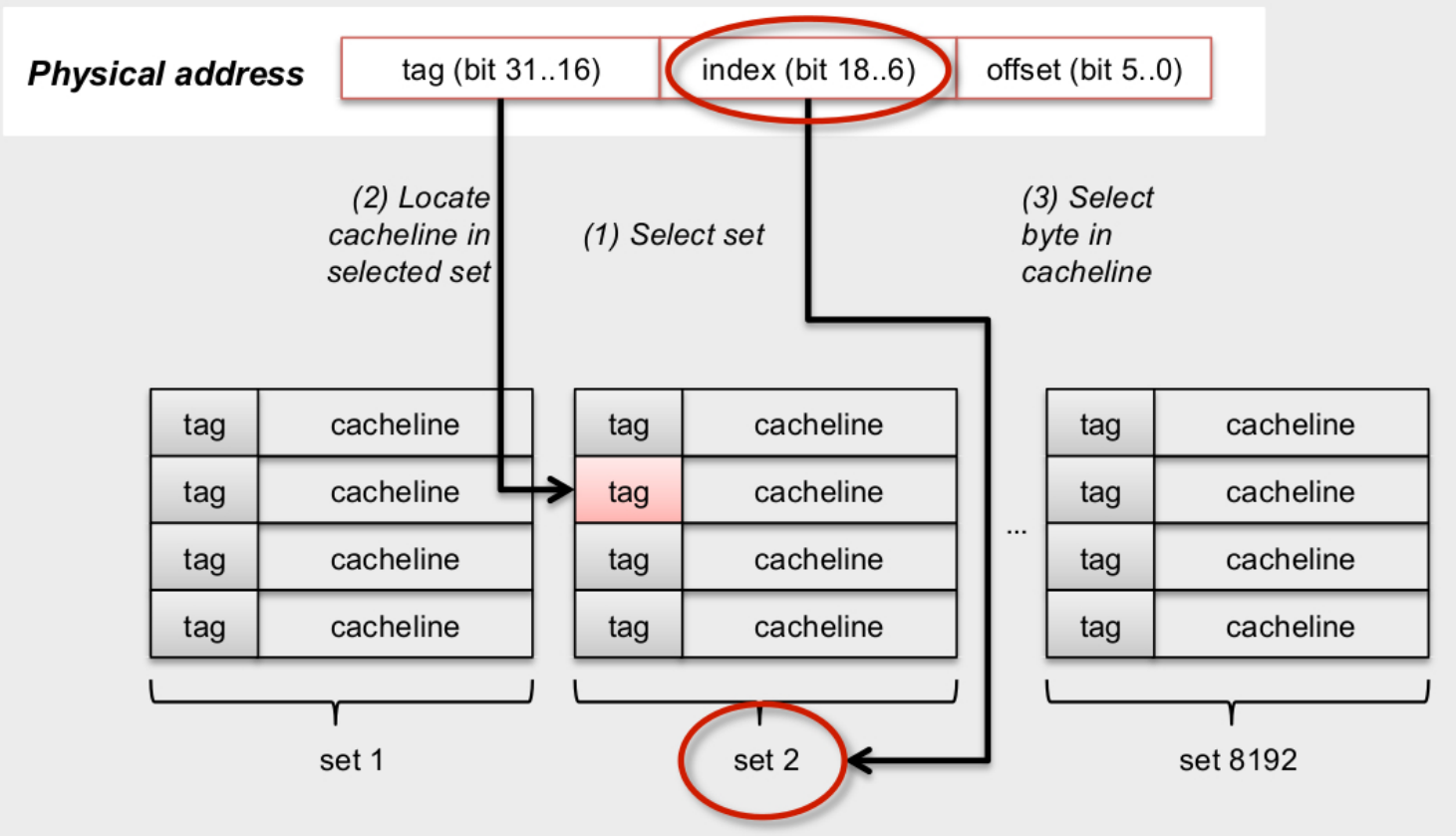
32-bit address and L3 cache of 8MB 16-way set associative:

$$(8192 * 1024)/64 = 131072 \text{ single slots } /16 = 8192 \text{ different sets}$$

Example

32-bit address and L3 cache of 8MB 16-way set associative:

$$(8192 * 1024)/64 = 131072 \text{ single slots } /16 = 8192 \text{ different sets}$$



Last Recently Used

- addresses with identical *cache index* bits compete against free slots of one set

Last Recently Used

- addresses with identical *cache index* bits compete against free slots of one set
- at memory access entry not accessed for the longest time replaced

Last Recently Used

- addresses with identical *cache index* bits compete against free slots of one set
- at memory access entry not accessed for the longest time replaced
- *Pseudo-LRU*: reference bit in each cacheline, set on each access

Last Recently Used

- addresses with identical *cache index* bits compete against free slots of one set
- at memory access entry not accessed for the longest time replaced
- *Pseudo-LRU*: reference bit in each cacheline, set on each access
- all reference bits of set enabled? $\rightarrow$ all cleared again

Memory Hierarchy

Paged Virtual Memory and Address Translation

- memory space divided into equally sized *pages*
- virtual address (*VA*) must be translated into physical address (*PA*)
- *Memory Management Unit* (*MMU*) make this in a *page walk*
  - *VA* split into several parts, as array index for certain level of *Page Table* (*PT*)
  - *Page Table Entry* (*PTE*) contains resulting physical *frame number*

Memory Hierarchy

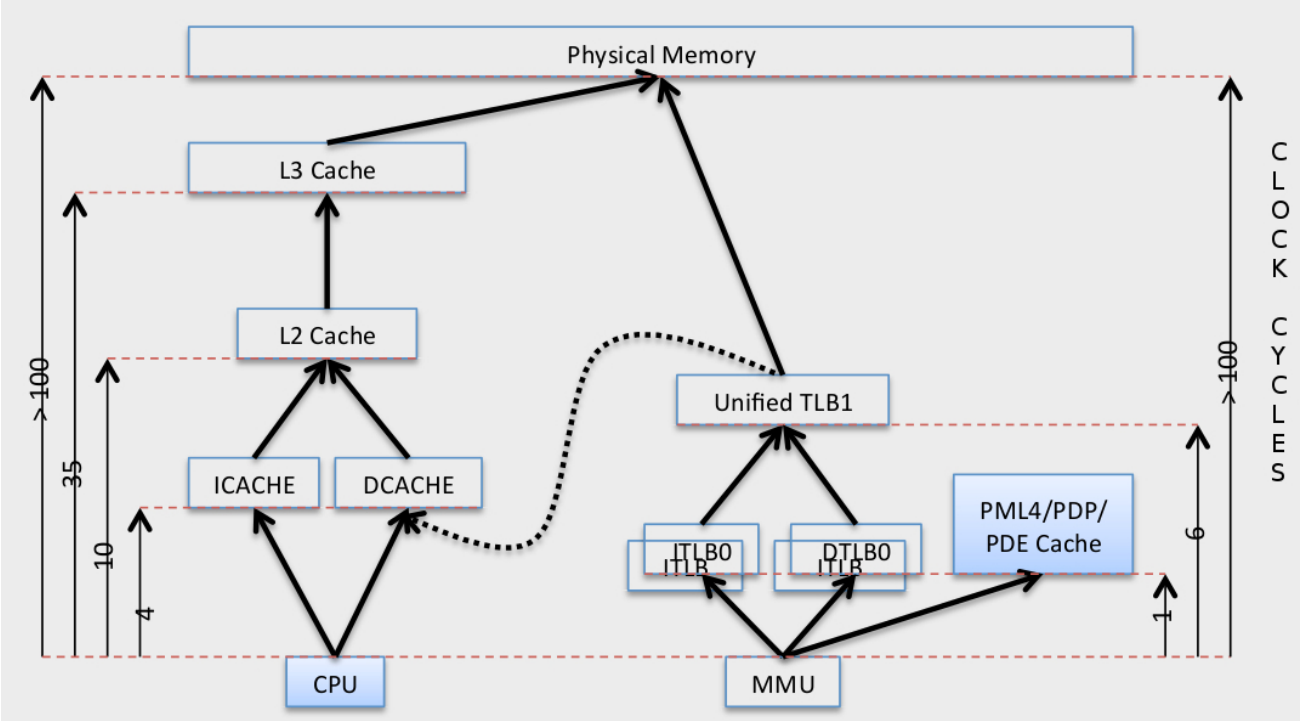
Paged Virtual Memory and Address Translation

- memory space divided into equally sized *pages*
- virtual address (*VA*) must be translated into physical address (*PA*)
- *Memory Management Unit* (*MMU*) make this in a *page walk*
  - *VA* split into several parts, as array index for certain level of *Page Table* (*PT*)
  - *Page Table Entry* (*PTE*) contains resulting physical *frame number*

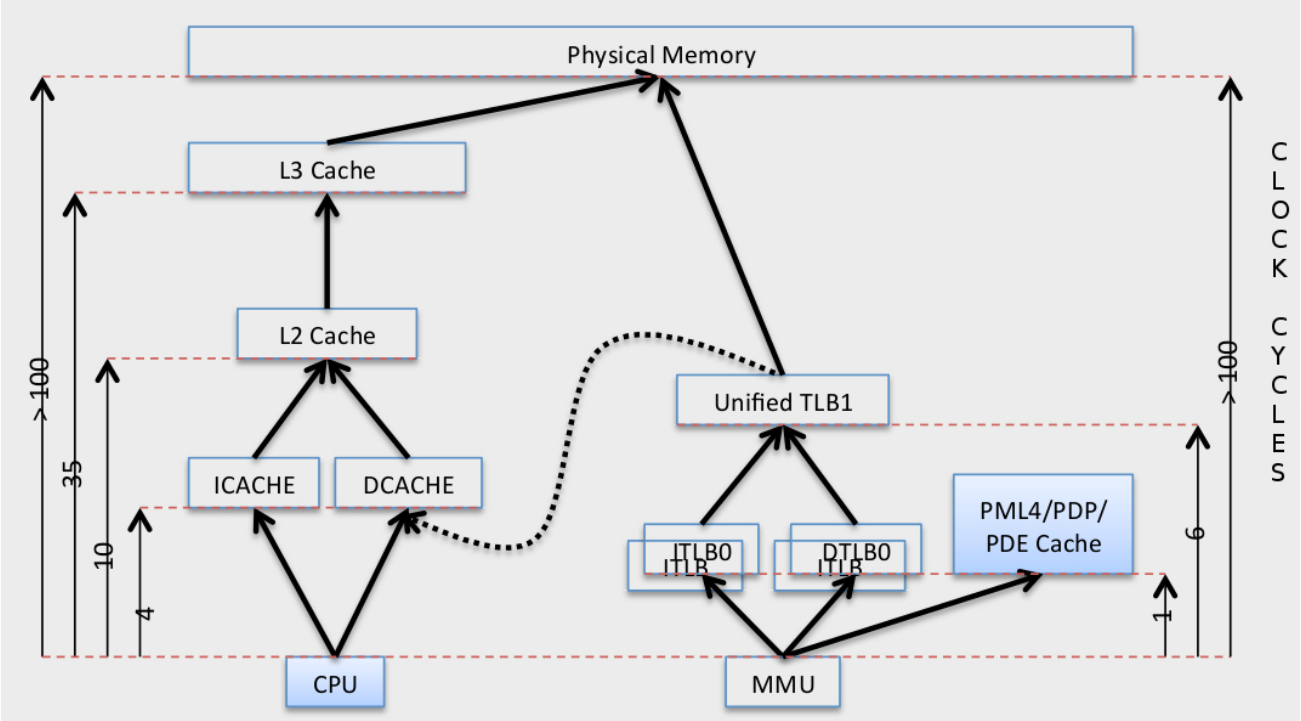
Translation Lookaside Buffer (TLB)

- speed up address translation process: containing resolved address mappings
- for each MMU
  1. *TLB0* split into *ITLB* (instructions) and *DTLB* (data)
  2. *TLB1* unified

Memory Hierarchy



Memory Hierarchy



Page Fault

- virtual address cannot be resolved
  - page swapped out
  - memory accessed first time
  - protection violation (check on flags)
- *Page Fault Handler* (*PFH*) invoked: time delay

Rationale

Usermode code and kernelmode code share hardware resources.

Attack

Rationale

Usermode code and kernelmode code share hardware resources.

Goal

Deduce info about kernel address space layout (to enable attacks based on ROP technique).

Attack

Rationale

Usermode code and kernelmode code share hardware resources.

Goal

Deduce info about kernel address space layout (to enable attacks based on ROP technique).

Assumptions

1. attacker is local user with restricted privileges
2. system fully supports ASLR and enforces $W \oplus X$ property

Attack

Rationale

Usermode code and kernelmode code share hardware resources.

Goal

Deduce info about kernel address space layout (to enable attacks based on ROP technique).

Assumptions

1. attacker is local user with restricted privileges
2. system fully supports ASLR and enforces $W \oplus X$ property

Method	Requirements	Results	Environment	Success
Cache Probing	*large pages* or PA of *eviction buffer*, partial information about *kernel_region* location	`ntoskrnl.exe` and `hal.sys`	all	✓
Double Page Fault	none	allocation map, several drivers	all but AMD	✓
Cache Preloading	none	`win32k.sys`	all	✓

Table I
SUMMARY OF TIMING SIDE CHANNEL ATTACKS AGAINST KERNEL SPACE ASLR ON WINDOWS.

Attack

General Approach

1. set the system in a specific way from usermode
2. measure the duration of a certain memory access
   - L1/L2/L3-based tests
   - TLB-based tests
3. noise (performance optimizations, parallelism . . .)
   - 2 step-test operations without interruption

Attack

General Approach

1. set the system in a specific way from usermode
2. measure the duration of a certain memory access
   - L1/L2/L3-based tests
   - TLB-based tests
3. noise (performance optimizations, parallelism . . .)
   - 2 step-test operations without interruption

Accessing Privileged Memory

- cannot access kernelspace memory directly from *usermode*
- indirectly force fixed execution paths and known data access patterns in the kernel, performing from usermode code:
  - a system call (*sysenter*)
  - an interrupt (*int*)
- measure the induced exception duration

Rational

- instruction/data caches may reveal parts of memory addresses
- caches are n-way-associative
- addresses mapped into same cache set, compiting for free slots

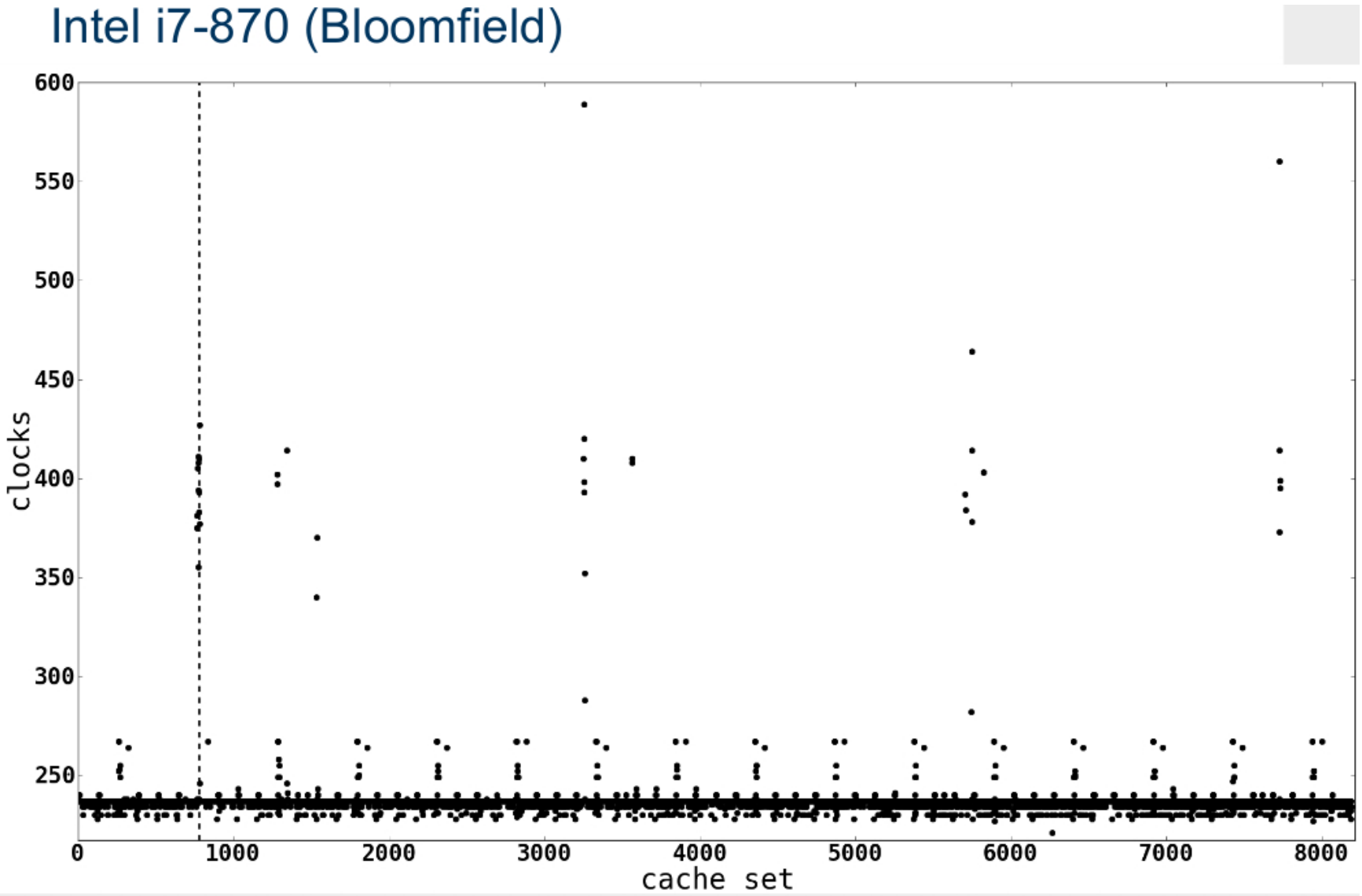
Implementation #1
Cache Probing

Rational

- instruction/data caches may reveal parts of memory addresses
- caches are n-way-associative
- addresses mapped into same cache set, compiting for free slots

Algorithm

1. caching wanted code/data indirectly (execute short system call)
2. invalidate cache set s (accessing to user-controlled *eviction buffer*)
3. execute system call again and measure time
4. time higher? accessed "eviction addresses" mapped into same set
5. can obtain *cache index* from known *colliding locations*


Intel i7-870 (Bloomfield)
clocks
cache set
600
550
500
450
400
350
300
250
0
1000
2000
3000
4000
5000
6000
7000
8000

Assuming the attacker knows...

- physical address of *eviction buffer* (at least *cache index*)
- corresponding *VA* of the kernel module from its *PA*
  - in Windows solved using *large pages*

Assuming the attacker knows...

- physical address of *eviction buffer* (at least *cache index*)
- corresponding *VA* of the kernel module from its *PA*
  - in Windows solved using *large pages*

Result

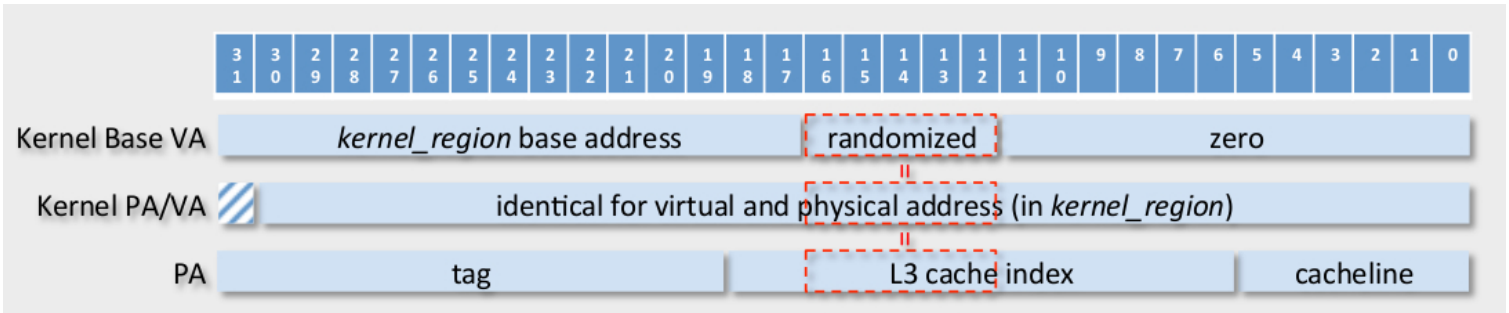
- timing measurements reveal L3 index (bits 6 ... 18)
- reveals parts of physical address
- no false positives in evaluation ($\sim$ 180 tests, $\sim$ 2 secs)

Assuming the attacker knows...

- physical address of *eviction buffer* (at least *cache index*)
- corresponding *VA* of the kernel module from its *PA*
  - in Windows solved using *large pages*

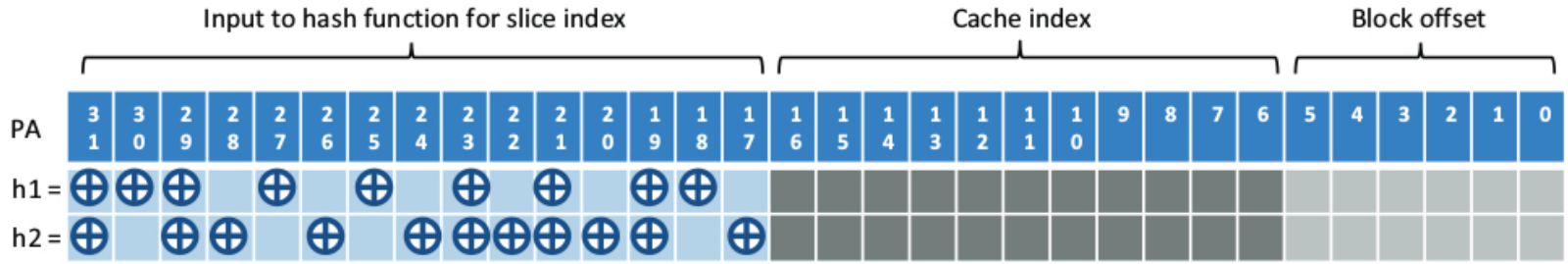
Result

- timing measurements reveal L3 index (bits 6 ... 18)
- reveals parts of physical address
- no false positives in evaluation ($\sim$ 180 tests, $\sim$ 2 secs)



Hash Function

- distributed last level cache for all CPU cores
- equally sized cache slides, each dedicated to one core
- accesses ordered by hash function undocumented



$$h = (h_1, h_2)$$

$$h_1 = b_{31} \oplus b_{30} \oplus b_{29} \oplus b_{27} \oplus b_{25} \oplus b_{23} \oplus b_{21} \oplus b_{19} \oplus b_{18}$$

$$h_2 = b_{31} \oplus b_{29} \oplus b_{28} \oplus b_{26} \oplus b_{24} \oplus b_{23} \oplus b_{22} \oplus b_{21} \oplus b_{20} \oplus b_{19} \oplus b_{17}$$

Figure 3. Results for the reconstruction of the undocumented Sandybridge hash function

Implementation #2
Double Page Fault Probing

Abstract idea

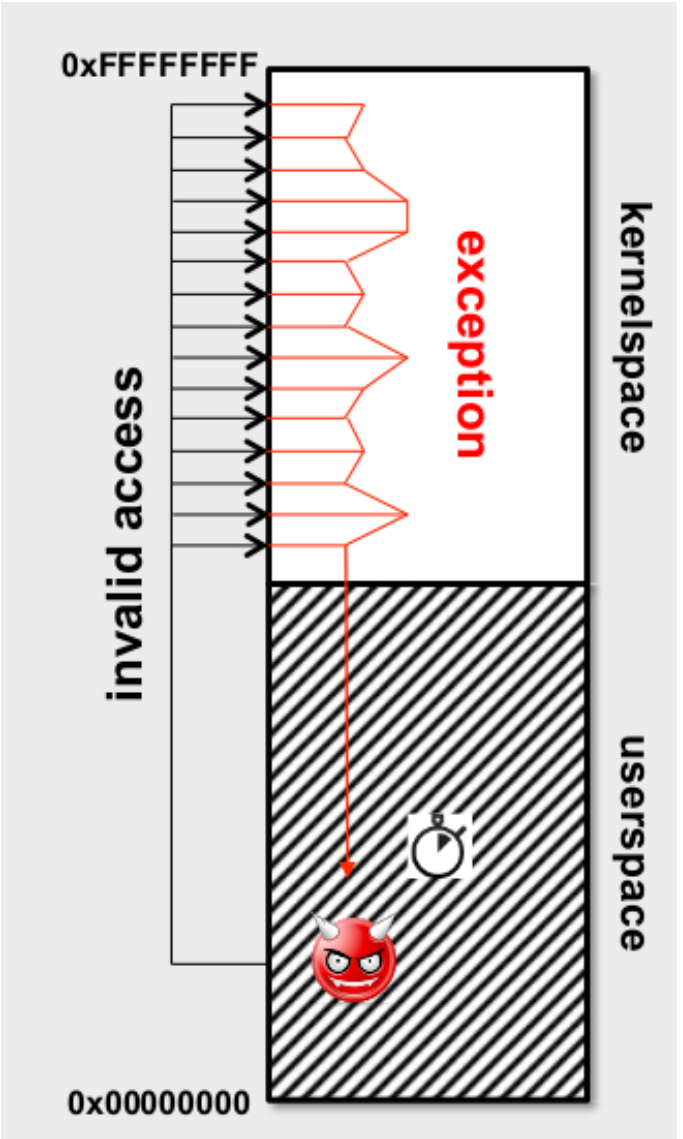
1. access kernelspace addresses two times
2. measure time duration until exception delivered

Implementation #2
Double Page Fault Probing

Abstract idea

1. access kernelspace addresses two times
2. measure time duration until exception delivered

- one probe of entire kernelspace takes ~ 2 seconds (32-bit)
- $2^{19} (\sim 500k)$ measurements



TLB of Intel CPUs

- access kernelspace page from usermode

TLB of Intel CPUs

- access kernelspace page from usermode
- case #1: refers to mapped page

TLB of Intel CPUs

- access kernelspace page from usermode
- case #1: refers to mapped page
  - TLB entry is created

TLB of Intel CPUs

- access kernelspace page from usermode
- case #1: refers to mapped page
  - TLB entry is created
  - page fault occurs (permission check fail)

TLB of Intel CPUs

- access kernelspace page from usermode
- case #1: refers to mapped page
  - TLB entry is created
  - page fault occurs (permission check fail)
  - access again: page fault delivered faster

TLB of Intel CPUs

- access kernelspace page from usermode
- case #1: refers to mapped page
  - TLB entry is created
  - page fault occurs (permission check fail)
  - access again: page fault delivered faster
- case #2: refers to unmapped page

TLB of Intel CPUs

- access kernelspace page from usermode
- case #1: refers to mapped page
  - TLB entry is created
  - page fault occurs (permission check fail)
  - access again: page fault delivered faster
- case #2: refers to unmapped page
  - no TLB entry is created

TLB of Intel CPUs

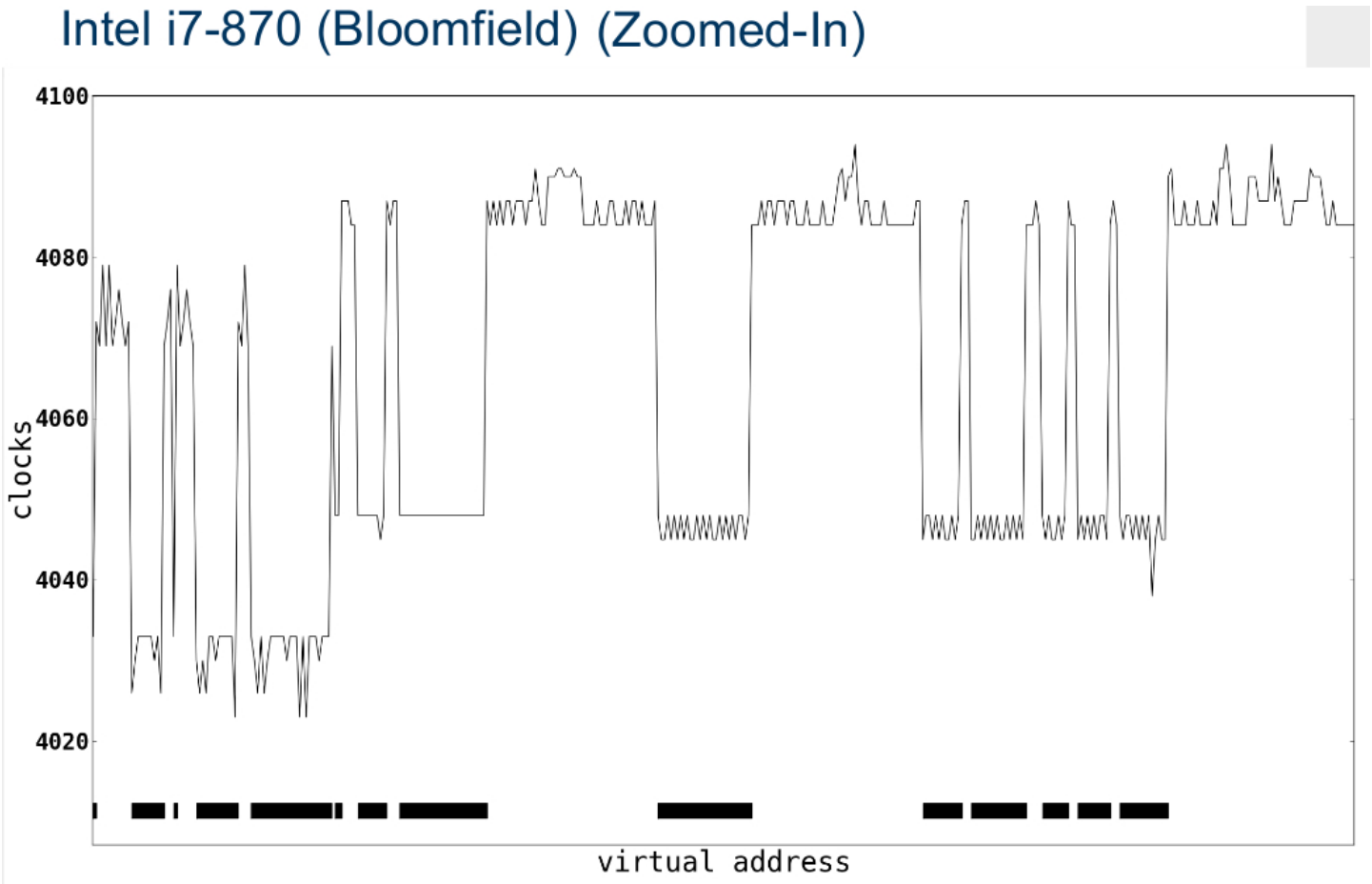
- access kernelspace page from usermode
- case #1: refers to mapped page
  - TLB entry is created
  - page fault occurs (permission check fail)
  - access again: page fault delivered faster
- case #2: refers to unmapped page
  - no TLB entry is created
  - page fault occurs (permission check fail)

TLB of Intel CPUs

- access kernelspace page from usermode
- case #1: refers to mapped page
  - TLB entry is created
  - page fault occurs (permission check fail)
  - access again: page fault delivered faster
- case #2: refers to unmapped page
  - no TLB entry is created
  - page fault occurs (permission check fail)
  - access again: page fault not delivered faster

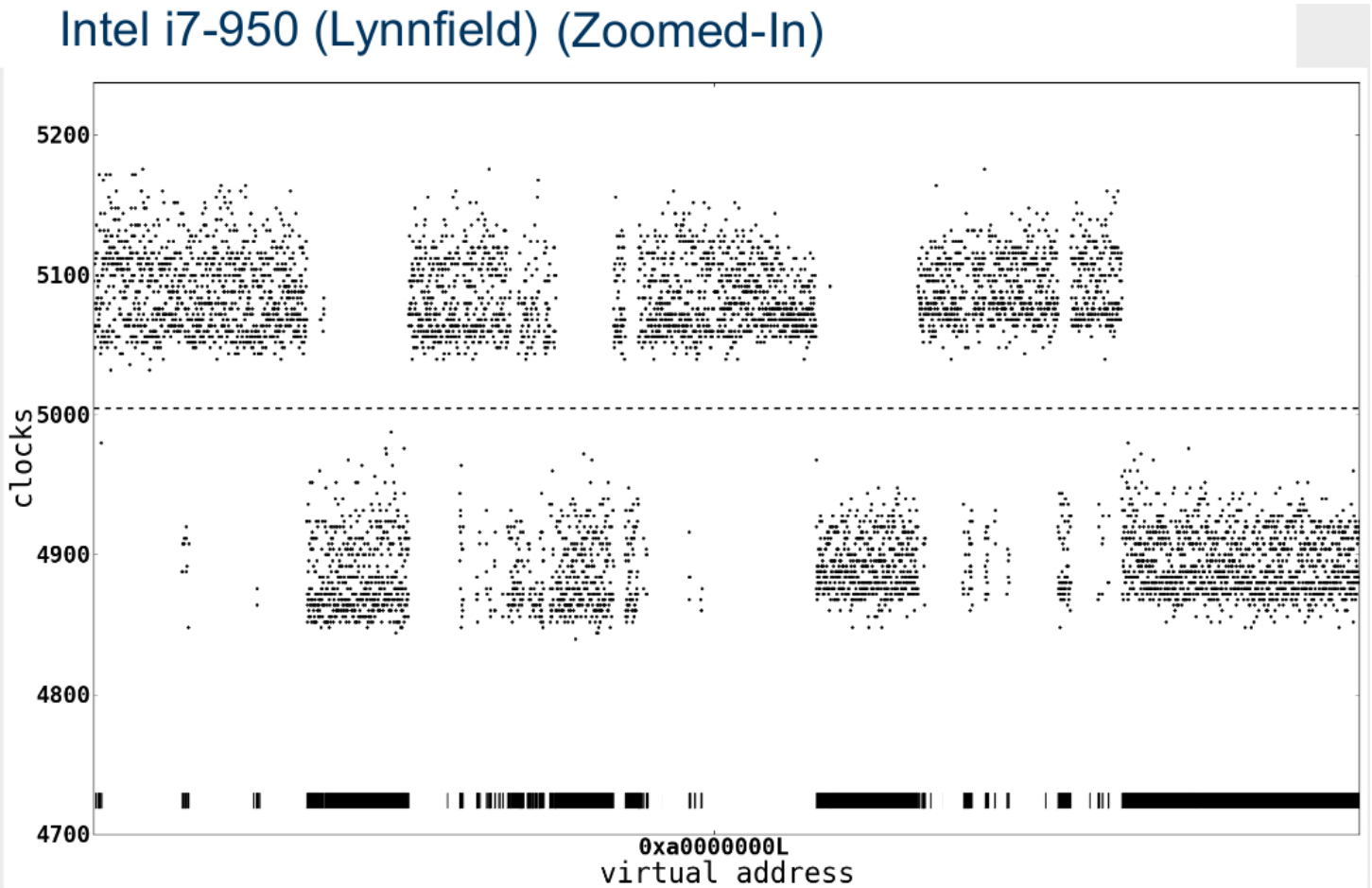
Threshold algorithm

Allocations reconstructed from timing values, differentiating allocated from unallocated pages, according to a thresold value.



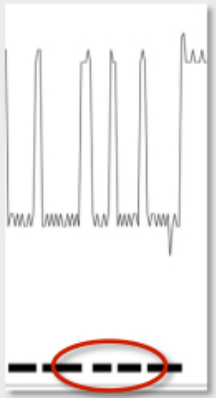
Change Point Detection (*CPD*) algorithm

Detecting transitions from allocated to unallocated memory by looking at the pitch of the timing curve.



Evaluation

- tests on three different Intel CPUs
  - correctly revealed 95% – 99% of kernelspace allocation
  - probes take between 20 and 70 seconds
- spot concrete driver using memory allocation signature matching
- enough known code to mount arbitrary attacks!
- fail on AMD CPUs



CPU Model	Matches	Code Size
(1) i7-870 (Bloomfield)	21	7,431 KB
(2) i7-950 (Lynnfield)	9	4,184 KB
(3) i7-2600 (Sandybr.)	5	1,696 KB
(4) VMware on (1)	18	7,079 KB
(1) with signatures of (2)	9	2,312 KB

Goal

Locate a certain driver.

Goal

Locate a certain driver.

Generic Approach

1. flush all caches to start with a *clean state*

Goal

Locate a certain driver.

Generic Approach

1. flush all caches to start with a *clean state*
2. preload TLBs by calling into kernel code (*sysenter*)

Goal

Locate a certain driver.

Generic Approach

1. flush all caches to start with a *clean state*
2. preload TLBs by calling into kernel code (*sysenter*)
3. intentionally generate a page fault by jumping to some kernelspace address

Goal

Locate a certain driver.

Generic Approach

1. flush all caches to start with a *clean state*
2. preload TLBs by calling into kernel code (*sysenter*)
3. intentionally generate a page fault by jumping to some kernelspace address
4. measure the time between the jump and the *PFH* return

Goal

Locate a certain driver.

Generic Approach

1. flush all caches to start with a *clean state*
2. preload TLBs by calling into kernel code (*sysenter*)
3. intentionally generate a page fault by jumping to some kernelspace address
4. measure the time between the jump and the *PFH* return
5. shorter time if the faulting address lies near the preloaded kernel memory

Goal

Locate a certain driver.

Generic Approach

1. flush all caches to start with a *clean state*
2. preload TLBs by calling into kernel code (*sysenter*)
3. intentionally generate a page fault by jumping to some kernelspace address
4. measure the time between the jump and the *PFH* return
5. shorter time if the faulting address lies near the preloaded kernel memory
   - already cached address translation information

Implementation #3

Flushing caches

- cannot be done from user mode directly
- done indirectly by accessing many memory addresses to evict all other data from caches
- the eviction buffer is chosen large enough (full of *RET* instructions)

Implementation #3

Flushing caches

- cannot be done from user mode directly
- done indirectly by accessing many memory addresses to evict all other data from caches
- the eviction buffer is chosen large enough (full of *RET* instructions)

Results

- locate *system service handler win32k.sys*
- side effect: located *System Service Dispatch/Parameter Tables* (*SSDT/SSPT*)
- only revealed the memory page of the searched kernel module
  - still possible to reconstruct full *VA*, obtaining relative address offset of the probed code/data by inspecting the module image file

Evaluation

No remarkable limitations: only it makes some time to complete, depending on the size of the probed memory range and amount of necessary test iterations.

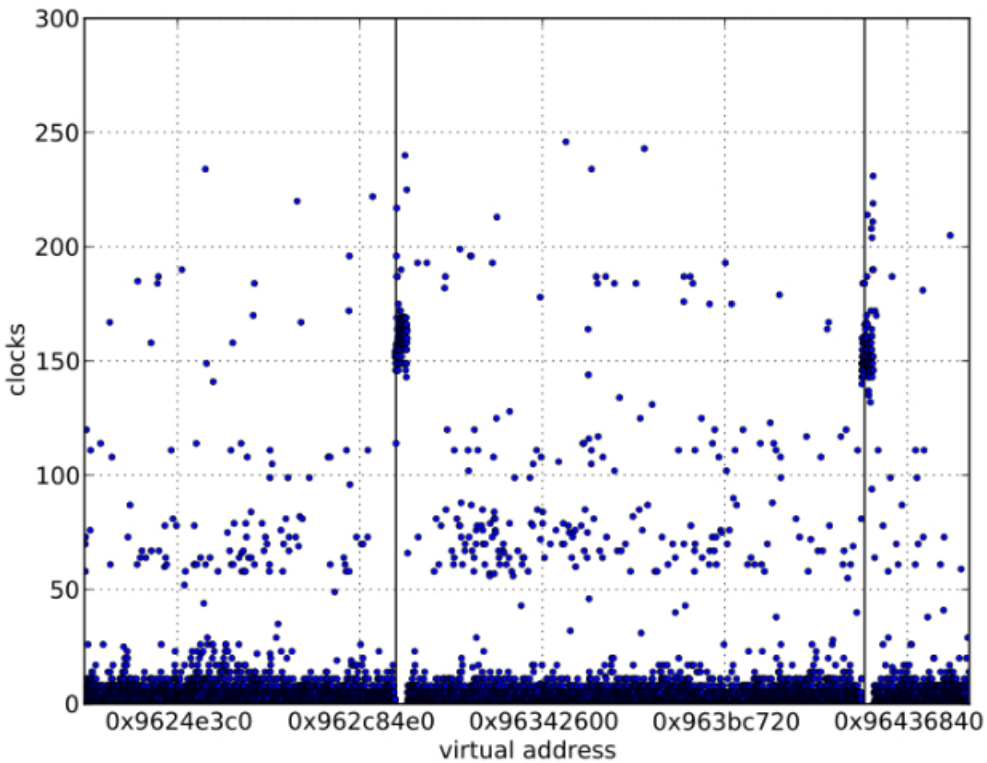


Figure 7. Extract of cache preloading measurements

Mitigations

Generic

- decouple cache facilities (cost?)
- increased randomization (performance?)

Mitigations

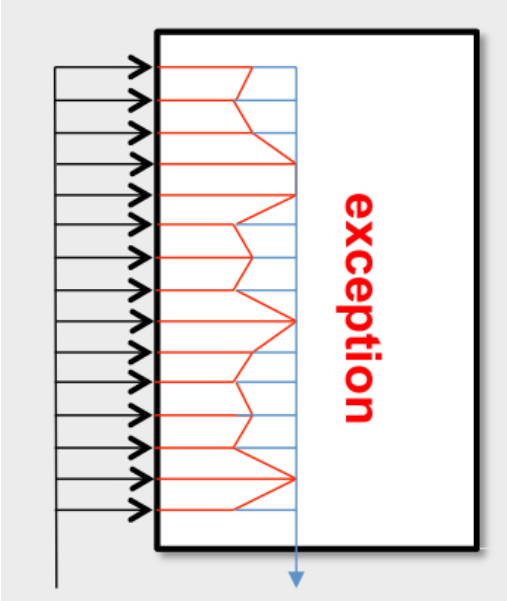
Generic

- decouple cache facilities (cost?)
- increased randomization (performance?)

Defense #1

Unify execution time of *PFH*

- no differences in timing measurements
- minimal performance impact



Mitigations

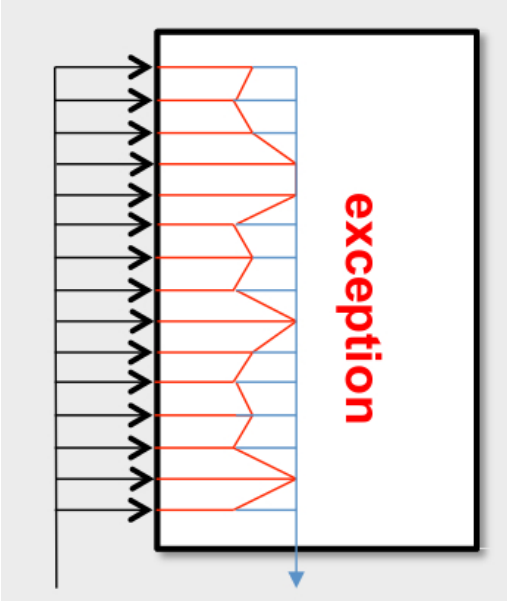
Generic

- decouple cache facilities (cost?)
- increased randomization (performance?)

Defense #1

Unify execution time of *PFH*

- no differences in timing measurements
- minimal performance impact



Defense #2

Apply ASLR to *VA* (not *PA*) $\rightarrow$ physical index bits not useful anymore

Conclusion and Future Work

Conclusion

- Kernelspace ASLR vulnerable to generic timing-based side channel attack
- **result: local restricted attacker can infer info about kernel memory layout**
- attacks are reliable and fast
- problem must be adressed in implementation

Conclusion and Future Work

Conclusion

- Kernelspace ASLR vulnerable to generic timing-based side channel attack
- **result: local restricted attacker can infer info about kernel memory layout**
- attacks are reliable and fast
- problem must be adressed in implementation

Future Work

- apply to other operating systems (e.g, Mac OS X)
- evaluate more architectures (e.g, ARM to attack Android ASLR)
- more virtualization software
- obtain the *PA* of a certain memory location from usermode

Grazie per l'attenzione!

Michele Corrias

Backup slides

Address Resolution on PAE systems

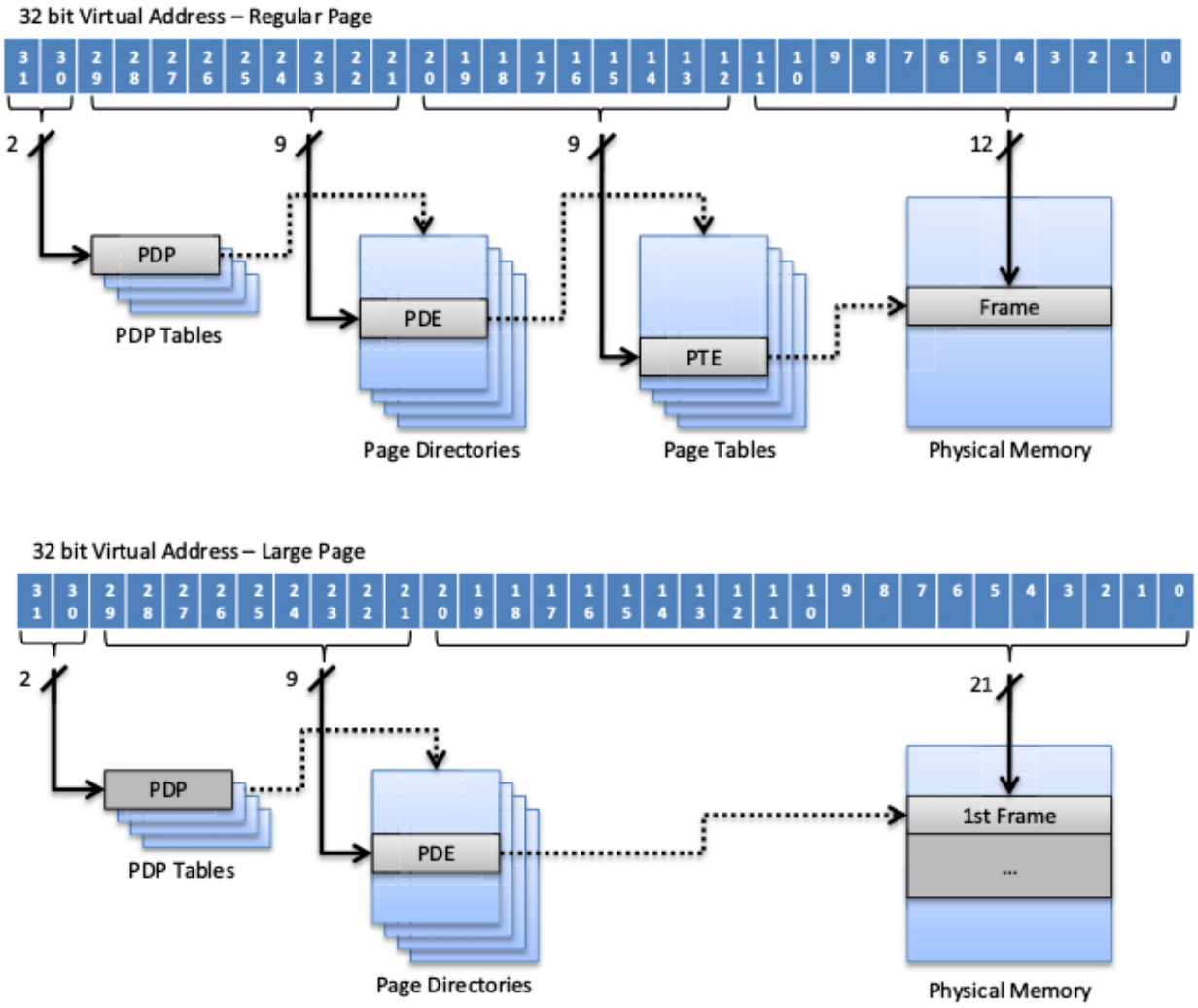


Figure 8. Address resolution for regular and large pages on PAE systems

Double Page Fault + *CPD*

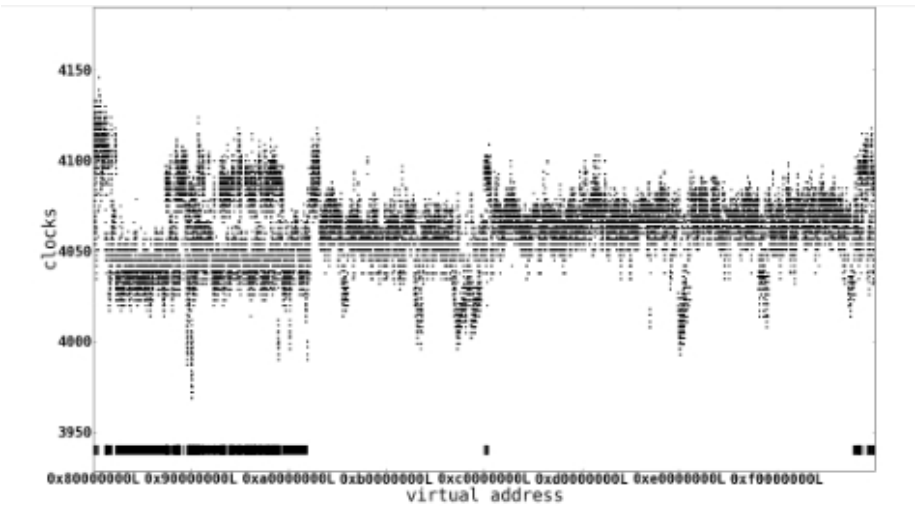


Figure 9. Double page fault measurements on Intel i7-870 (Bloomfield) processor

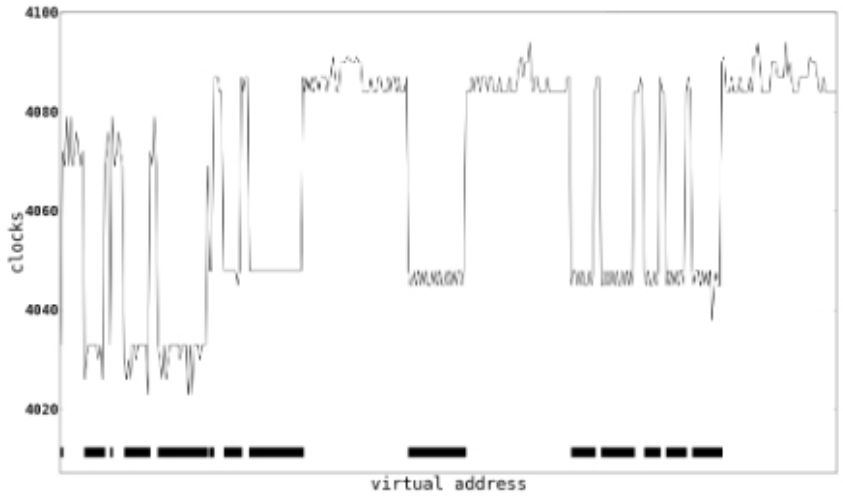


Figure 10. Zoomed-in view of Figure 9