Side-Channel Security

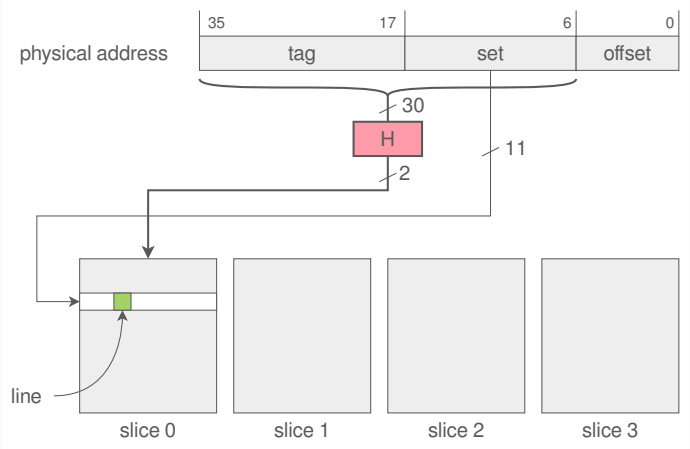
Chapter 3: Reverse Engineering Intel Last-Level Cache Mapping + AMD Way Predictor

Claudio Canella

March 16, 2021

Graz University of Technology

Last-level cache addressing



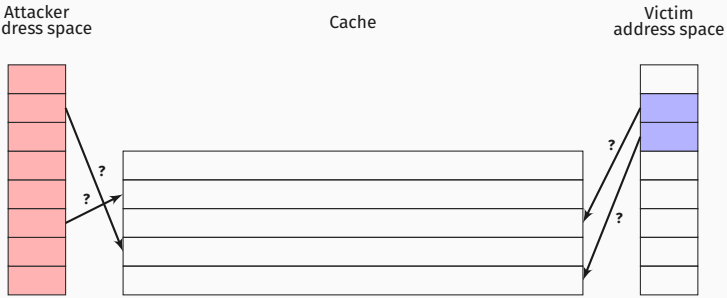
Last-level cache addressing

- last-level cache $\rightarrow$ as many slices as cores
- *undocumented* hash function that maps a physical address to a slice
- designed for performance

For 2^k slices:

Prime+Probe on recent procesors?

Complex addressing $\rightarrow$ impossible to *target a set*



Reverse engineering method

1. find some way to map *one address* to *one slice*

Reverse engineering method

1. find some way to map *one address* to *one slice*
2. *repeat* for every address with a 64B stride

Reverse engineering method

1. find some way to map *one address* to *one slice*
2. *repeat* for every address with a 64B stride
3. infer a *function* out of it

How to map addresses to slices?

How to map addresses to slices?

- with performance counters

How to map addresses to slices?

- with performance counters
- with a timing attack

How to map addresses to slices?

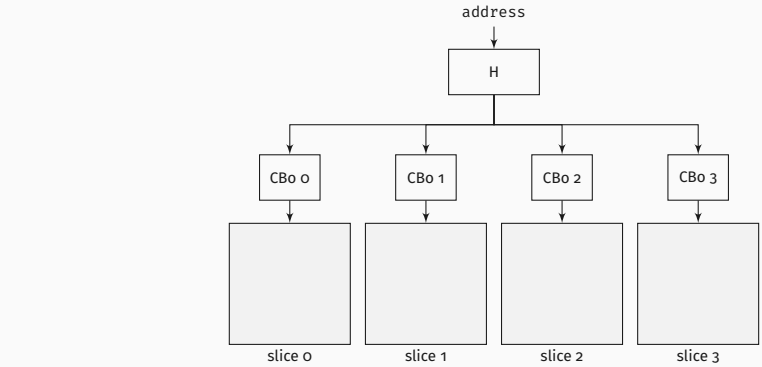
- with performance counters
- with a timing attack
  - using `clflush`

How to map addresses to slices?

- with performance counters
- with a timing attack
  - using `clflush`
  - using memory access

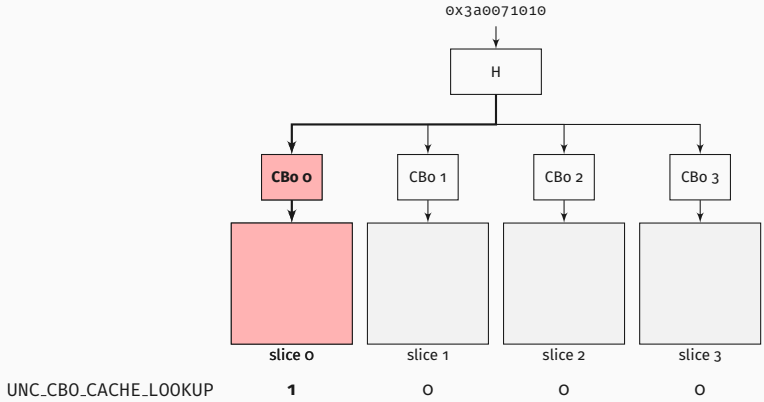
Method #1: Performance counters

- event UNC_CBO_CACHE_LOOKUP counts accesses to a slice



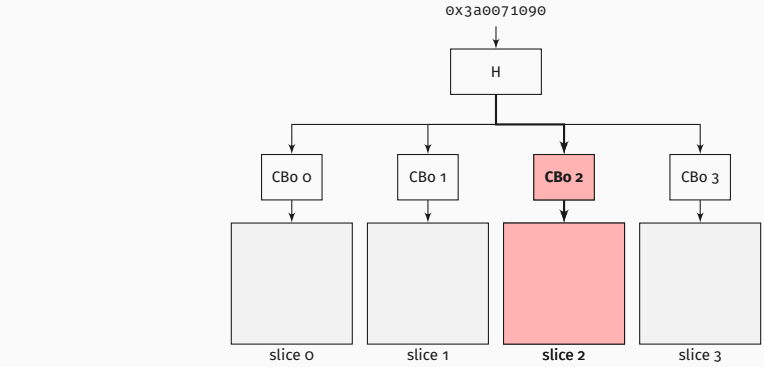
Method #1: Performance counters

- event UNC_CBO_CACHE_LOOKUP counts accesses to a slice



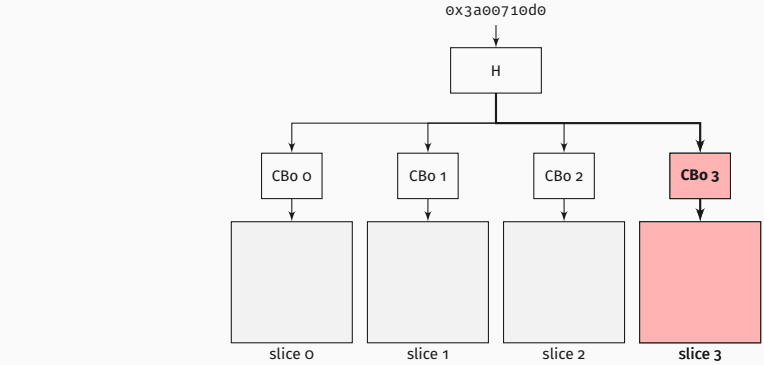
Method #1: Performance counters

- event UNC_CBO_CACHE_LOOKUP counts accesses to a slice



Method #1: Performance counters

- event UNC_CBO_CACHE_LOOKUP counts accesses to a slice



Mapping a physical addr. to a slice [Mau+15]

1. translate virtual to physical address with `/proc/pid/pagemap`

Mapping a physical addr. to a slice [Mau+15]

1. translate virtual to physical address with `/proc/pid/pagemap`
2. set up monitoring session

Mapping a physical addr. to a slice [Mau+15]

1. translate virtual to physical address with `/proc/pid/pagemap`
2. set up monitoring session
3. *repeat access* to a single address

Mapping a physical addr. to a slice [Mau+15]

1. translate virtual to physical address with `/proc/pid/pagemap`
2. set up monitoring session
3. *repeat access* to a single address
   → hint: `clflush` is already counted as an access

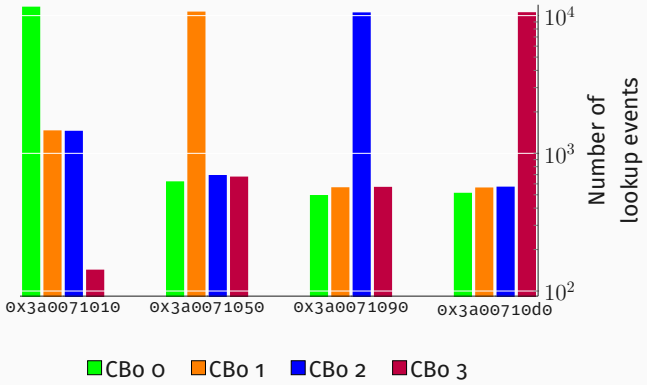
Mapping a physical addr. to a slice [Mau+15]

1. translate virtual to physical address with `/proc/pid/pagemap`
2. set up monitoring session
3. *repeat access* to a single address
   → hint: `clflush` is already counted as an access
4. read UNC_CBO_CACHE_LOOKUP event for each CBo

Mapping a physical addr. to a slice [Mau+15]

1. translate virtual to physical address with `/proc/pid/pagemap`
2. set up monitoring session
3. *repeat access* to a single address
   → hint: `clflush` is already counted as an access
4. read UNC_CBO_CACHE_LOOKUP event for each CBo
5. slice is the one that has the maximum lookup events

Mapping physical addresses to slices



The gory details (1)

Accessing the performance counters

- program *Model Specific Registers* (MSRs) manually
- documentation in Intel Manual vol 3 (only 1914 pages!)
- code provided at `https://github.com/clementine-m/msr-uncore-cbo`

The gory details (2)

`monitoring.c`

1.	disable counters	write to `MSR_UNC_PERF_GLOBAL_CTRL`
2.	reset counters	write to `MSR_UNC_CBO_i_PER_CTR0` for each CBo i
3.	select event to monitor	write to `MSR_UNC_CBO_i_PERFEVTSEL0` for each CBo i
4.	enable counting	write to `MSR_UNC_PERF_GLOBAL_CTRL`
5.	function to monitor	cause access to address
6.	read counters	read `MSR_UNC_CBO_i_PER_CTR0` for each CBo i

The gory details (3)

`scan.c`

- outputs performance counters for UNC_CBO_CACHE_LOOKUP event for each CBo, every 0.1s

The gory details (3)

`scan.c`

- outputs performance counters for UNC_CBO_CACHE_LOOKUP event for each CBo, every 0.1s
- create a *mapping table* for each *physical address* (with a 64B stride) to a *slice*

A few tips

Performance counters via MSRs monitor *every* event on your CPU $\rightarrow$ *noise*

- for one address, make a lot of accesses

A few tips

Performance counters via MSRs monitor *every* event on your CPU $\rightarrow$ *noise*

- for one address, make a lot of accesses
- try to minimize background noise

A few tips

Performance counters via MSRs monitor *every* event on your CPU $\rightarrow$ *noise*

- for one address, make a lot of accesses
- try to minimize background noise
- verify your table matches known functions [Mau+15]

A few tips

Performance counters via MSRs monitor *every* event on your CPU $\rightarrow$ *noise*

- for one address, make a lot of accesses
- try to minimize background noise
- verify your table matches known functions [Mau+15]
- functions from [Mau+15] are valid for Sandy Bridge, Ivy Bridge, Haswell and Broadwell

Method #2: Flush+Flush [Gru+16]

- side channel similar to Flush+Reload, using only `clflush` execution time differences

Method #2: Flush+Flush [Gru+16]

- side channel similar to Flush+Reload, using only `clflush` execution time differences
- time difference between cached and not cached

Method #2: Flush+Flush [Gru+16]

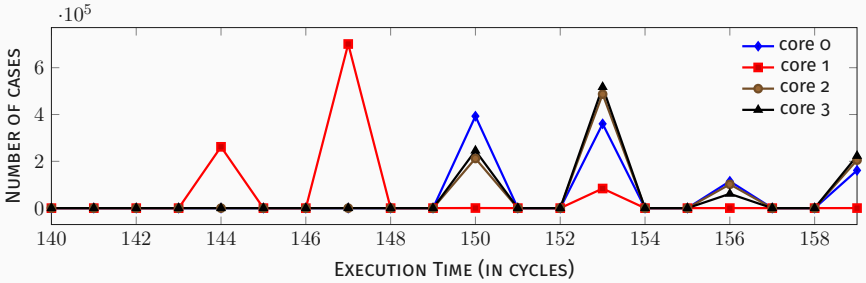
- side channel similar to Flush+Reload, using only `clflush` execution time differences
- time difference between cached and not cached
- time difference between *slices*

Last-level cache and Ring interconnect

Intel Optimization Manual:

“The LLC hit latency [...] depends on the core location relative to the LLC block, and *how far the request needs to travel on the ring*.”

clflush time difference between slices



clflush histogram for an address in slice 1 on different cores

Learning phase

- timing attack: first phase is…?

Learning phase

- timing attack: first phase is…?
- building a *histogram* of “flush hits”!

Learning phase

- timing attack: first phase is…?
- building a *histogram* of “flush hits”!
  - for each slice: local and remote slices

Learning phase

- timing attack: first phase is…?
- building a *histogram* of “flush hits”!
  - for each slice: local and remote slices
- find a threshold to distinguish a hit on *remote* slice vs *local* slice

Building the histogram

repeat:

1. measure time
2. flush a line (“flush hit”)
3. measure time
4. update histogram with delta
5. access the line twice

Mapping one physical address to a slice

1. translate virtual to physical address with `/proc/pid/pagemap`

Mapping one physical address to a slice

1. translate virtual to physical address with `/proc/pid/pagemap`
2. for each core ($\neq$ thread):
    - build a histogram for this address

Mapping one physical address to a slice

1. translate virtual to physical address with `/proc/pid/pagemap`
2. for each core ($\neq$ thread):
   - build a histogram for this address
3. decide which core corresponds to the local slice using threshold

$\rightarrow$ *local* slice = slice with *fastest* hits

Method #3: Using memory accesses

- used in [Yar+15]
- similar to the method using `clflush`
- measure the time difference between hits in different slices

Method #3: Using memory accesses

- used in [Yar+15]
- similar to the method using `clflush`
- measure the time difference between hits in different slices
- less convenient:

Method #3: Using memory accesses

- used in [Yar+15]
- similar to the method using `clflush`
- measure the time difference between hits in different slices
- less convenient:
  - must evict data from L1 and L2 exactly (not LLC)

  $\rightarrow$ more susceptible to noise

Method #3: Using memory accesses

- used in [Yar+15]
- similar to the method using `clflush`
- measure the time difference between hits in different slices
- less convenient:
  - must evict data from L1 and L2 exactly (not LLC)

  $\rightarrow$ more susceptible to noise
  - longer

Inferring the function

Two cases:

1. linear function: 2^n number of cores
2. non-linear function: the rest

Methods

- brute-force
- faster method

Perspective: both methods assume that we know what the functions look like (XORs of address bits)

Brute force

For each bit of output $o_0, \ldots, o_{k-1}$

1. try one function

$\rightarrow$ e.g., $b_6 \oplus b_7 \oplus \ldots \oplus b_{32}$

2. test if the function corresponds to the mapping you already have

Brute force

For each bit of output $o_0, \ldots, o_{k-1}$

1. try one function

$\rightarrow$ e.g., $b_6 \oplus b_7 \oplus \ldots \oplus b_{32}$

2. test if the function corresponds to the mapping you already have
3. if not $\rightarrow$ try another function until it works!

Brute force

For each bit of output $o_0, \ldots, o_{k-1}$

1. try one function

$\rightarrow$ e.g., $b_6 \oplus b_7 \oplus \ldots \oplus b_{32}$

2. test if the function corresponds to the mapping you already have
3. if not $\rightarrow$ try another function until it works!

Fail fast, but still tolerate some errors

Finding linear function (1/2) [Mau+15]

- XOR is commutative: $A \oplus B \Leftrightarrow B \oplus A$

Finding linear function (1/2) [Mau+15]

- XOR is commutative: $A \oplus B \Leftrightarrow B \oplus A$
- XOR is associative: $A \oplus (B \oplus C) \Leftrightarrow (A \oplus B) \oplus C$

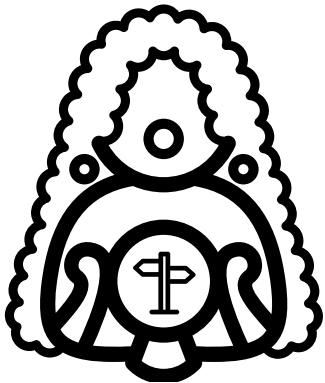
Finding linear function (1/2) [Mau+15]

- XOR is commutative: $A \oplus B \Leftrightarrow B \oplus A$
- XOR is associative: $A \oplus (B \oplus C) \Leftrightarrow (A \oplus B) \oplus C$
- you can treat all the input bits independently

Finding linear function (2/2) [Mau+15]

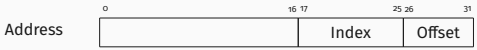
For each bit of output $o_0, \ldots, o_{k-1}$

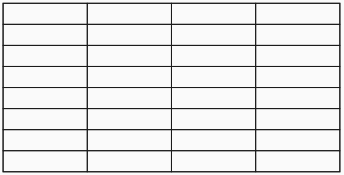
- find two addresses that differ by a single bit b_i
- compare output
  - if different: b_i is part of the function
  - if equal: b_i is not part of the function



Way Prediction

Set-Associative Caches



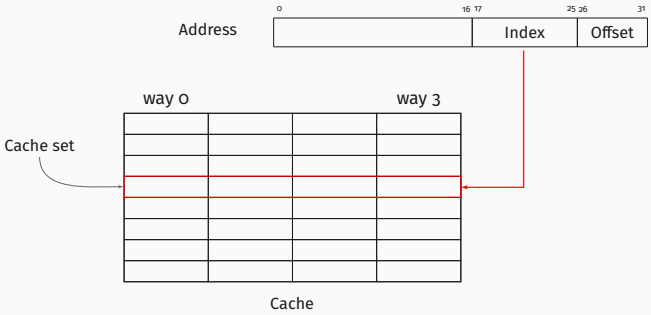


Cache

Set-Associative Caches

Data loaded in a specific set depending on its address

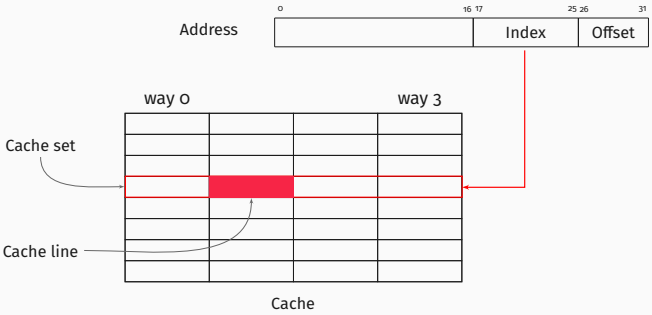
Set-Associative Caches



Data loaded in a specific set depending on its address

Several ways per set

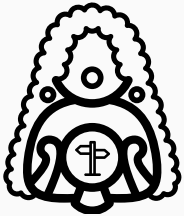
Set-Associative Caches



Data loaded in a specific set depending on its address

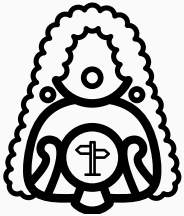
Several ways per set

Way Prediction



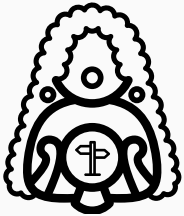
- In a set-associative cache, bits in the address determine in which set the cache line is located.

Way Prediction



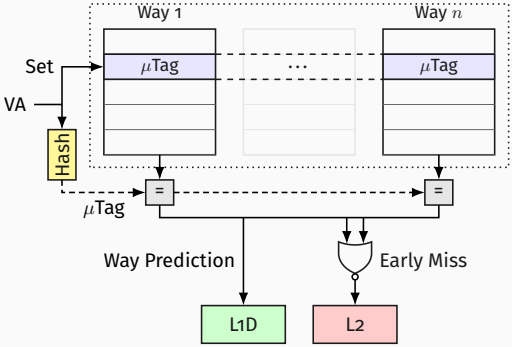
- In a set-associative cache, bits in the address determine in which set the cache line is located.
- With an n-way cache, n possible entries need to be checked.

Way Prediction



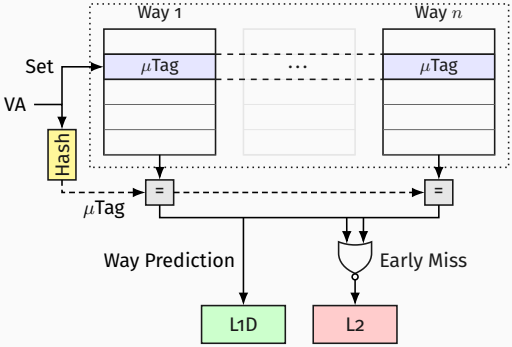
- In a set-associative cache, bits in the address determine in which set the cache line is located.
- With an n-way cache, n possible entries need to be checked.
- Using **way prediction** [IIM99], one entry is predicted
  - Correct prediction: Access completed
  - Incorrect prediction: Perform associate check

AMD Way Predictor



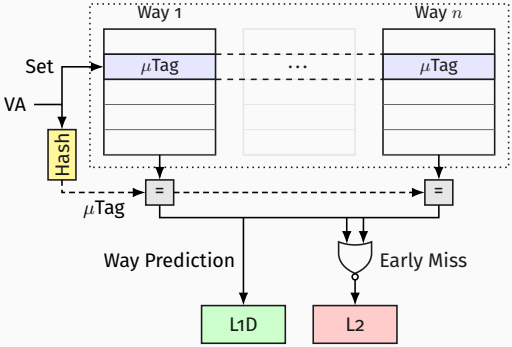
- Introduced with the AMD Bulldozer microarchitecture

AMD Way Predictor



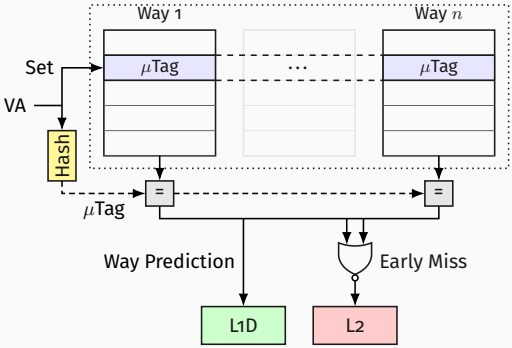
- Introduced with the AMD Bulldozer microarchitecture
- Every cache line in the L1D is tagged with a μTag

AMD Way Predictor



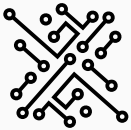
- Introduced with the AMD Bulldozer microarchitecture
- Every cache line in the L1D is tagged with a μTag
- Predicts the cache way based on this μTag
  - Saving power and reduces bank conflicts

AMD Way Predictor



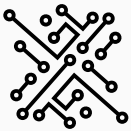
- Introduced with the AMD Bulldozer microarchitecture
- Every cache line in the L1D is tagged with a μTag
- Predicts the cache way based on this μTag
  - Saving power and reduces bank conflicts
- No match for μTag, detect early miss and issue L2 request

AMD Way Predictor



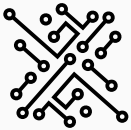
- Aliased cache lines can induce performance penalties
  - Two different virtual addresses that map to the same physical address

AMD Way Predictor



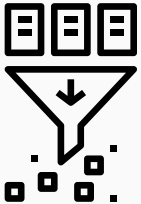
- Aliased cache lines can induce performance penalties
  - Two different virtual addresses that map to the same physical address
- Loads to an address that is valid in the L1D under a different alias will see an L1D miss

AMD Way Predictor



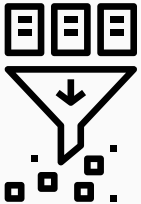
- Aliased cache lines can induce performance penalties
  - Two different virtual addresses that map to the same physical address
- Loads to an address that is valid in the L1D under a different alias will see an L1D miss
- Two different virtual addresses with the same μTag but different physical addresses will conflict

Hash Function



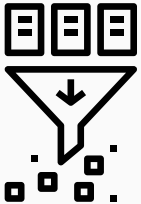
- L1D way predictor computes a hash (μTag) from the virtual address

Hash Function



- L1D way predictor computes a hash (μTag) from the virtual address
- This hash function is **not documented**

Hash Function



- L1D way predictor computes a hash (μTag) from the virtual address
- This hash function is **not documented**
  - Assume it is linear-based like other hash functions (cache slices, DRAM mapping, …)
  - Expect the size of the μTag to be a power of 2

Recovering the Hash Function

- Rely on μTag collisions to reverse-engineer the hash function

Recovering the Hash Function

- Rely on μTag collisions to reverse-engineer the hash function
- Pick two random virtual addresses mapping to the same cache set

Recovering the Hash Function

- Rely on μTag collisions to reverse-engineer the hash function
- Pick two random virtual addresses mapping to the same cache set
- Access them repeatedly

Recovering the Hash Function

- Rely on μTag collisions to reverse-engineer the hash function
- Pick two random virtual addresses mapping to the same cache set
- Access them repeatedly
- If they have the **same μTag**:
  - Increased access time
  - Increased number of performance counter for L1 misses

Recovering the Hash Function

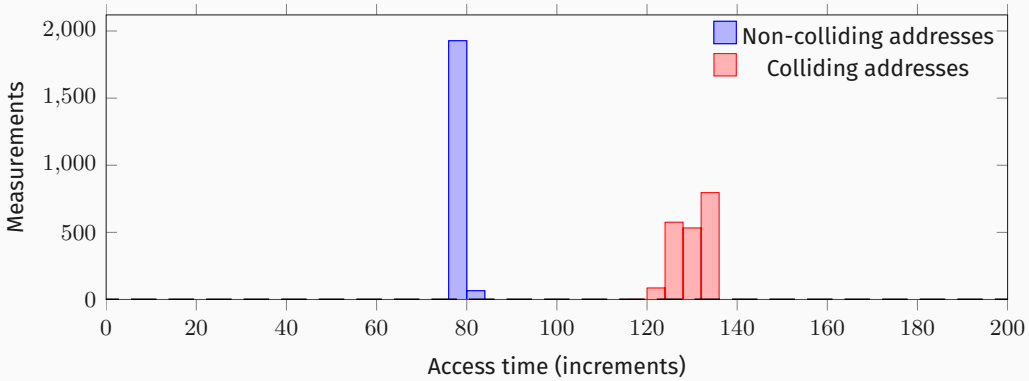
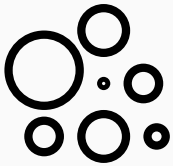


Figure 1: Measured duration of 250 alternating accesses to addresses with and without the same μTag.

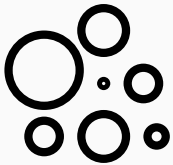
Creating sets



Build n sets where n is the number of entries in the μTag table

1. Create pool v of virtual addresses which all map to the same cache set

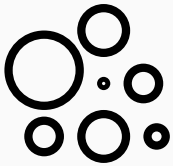
Creating sets



Build n sets where n is the number of entries in the μTag table

1. Create pool v of virtual addresses which all map to the same cache set
2. Start with one set S_0 containing one address from v

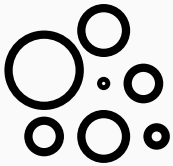
Creating sets



Build n sets where n is the number of entries in the μTag table

1. Create pool v of virtual addresses which all map to the same cache set
2. Start with one set S_0 containing one address from v
3. Repeat
   - Pick random address v_x

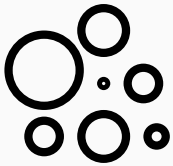
Creating sets



Build n sets where n is the number of entries in the μTag table

1. Create pool v of virtual addresses which all map to the same cache set
2. Start with one set S_0 containing one address from v
3. Repeat
   - Pick random address v_x
   - Measure access time of v_x and an address from each set $S_{0..n}$

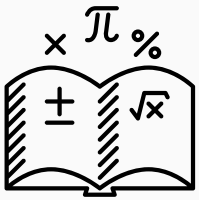
Creating sets



Build n sets where n is the number of entries in the μTag table

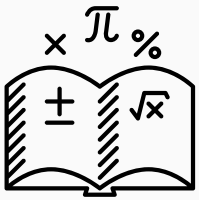
1. Create pool v of virtual addresses which all map to the same cache set
2. Start with one set S_0 containing one address from v
3. Repeat
   - Pick random address v_x
   - Measure access time of v_x and an address from each set $S_{0..n}$
   - If conflict, add to set. Otherwise, create new set containing v_x.

Recover hash function



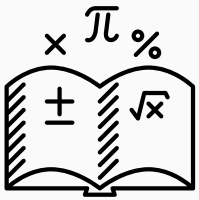
- Every virtual address within a set produces the **same 8-bit hash**

Recover hash function



- Every virtual address within a set produces the **same 8-bit hash**
- Each output bit can be expressed as a **series of XORs**

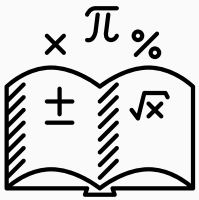
Recover hash function



- Every virtual address within a set produces the **same 8-bit hash**
- Each output bit can be expressed as a **series of XORs**
  - Over-determined linear equation system
  - b bits of virtual address a are coefficients, bits of the hash x are the unknown
  - For every address a in set s:

    $a_{b-1}x_{b-1} \oplus a_{b-2}x_{b-2} \oplus \cdots \oplus a_{12}x_{12} = y_s$

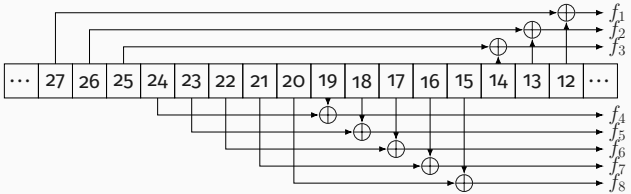
Recover hash function



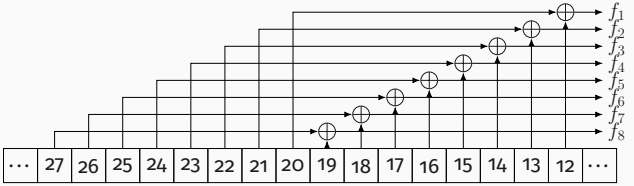
- Every virtual address within a set produces the **same 8-bit hash**
- Each output bit can be expressed as a **series of XORs**
  - Over-determined linear equation system
  - b bits of virtual address a are coefficients, bits of the hash x are the unknown
  - For every address a in set s:

    $a_{b-1}x_{b-1} \oplus a_{b-2}x_{b-2} \oplus \cdots \oplus a_{12}x_{12} = y_s$
- Solve equation system using Z3 SAT solver

Recovering the Hash Function



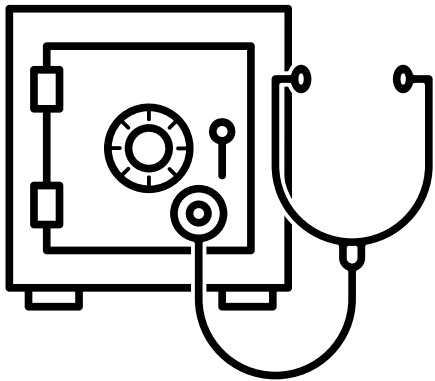
(a) Zen, Zen+, Zen 2



(b) Bulldozer, Piledriver, Steamroller

Microarchitectures

Setup	CPU	μ-arch.	WP
Lab	AMD Athlon 64 X2 3800+	K8	✗
Lab	AMD Turion II Neo N40L	K10	✗
Lab	AMD Phenom II X6 1055T	K10	✗
Lab	AMD E-450	Bobcat	✗
Lab	AMD Athlon 5350	Jaguar	✗
Lab	AMD FX-4100	Bulldozer	✓
Lab	AMD FX-8350	Piledriver	✓
Lab	AMD A10-7870K	Steamroller	✓
Lab	AMD Ryzen Threadripper 1920X	Zen	✓
Lab	AMD Ryzen Threadripper 1950X	Zen	✓
Lab	AMD Ryzen Threadripper 1700X	Zen	✓
Lab	AMD Ryzen Threadripper 2970WX	Zen+	✓
Lab	AMD Ryzen 7 3700X	Zen 2	✓
Cloud	AMD EPYC 7401p	Zen	✓
Cloud	AMD EPYC 7571	Zen	✓



Way Predictor for Side Channels

Collide+Probe

- Exploit μTag collisions in L1D cache way predictor

Collide+Probe

- Exploit μTag collisions in L1D cache way predictor
- Does not require physical address information like Prime+Probe or specific instructions like Flush+Reload

Collide+Probe

- Exploit μTag collisions in L1D cache way predictor
- Does not require physical address information like Prime+Probe or specific instructions like Flush+Reload
- Does not work cross core and requires a high resolution timer

Collide+Probe

Attacker selects v' with the same μTag as the target address

- **Phase 1: Collide**: Access v' to update the way predictor

Collide+Probe

Attacker selects v' with the same μTag as the target address

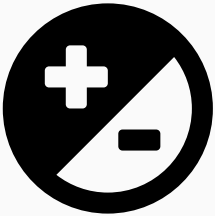
- **Phase 1: Collide**: Access v' to update the way predictor
- **Phase 2: Scheduling the victim**: The victim process is scheduled.
  - Access v: Updating the way predictor causing v' to be inaccessible from L1D

Collide+Probe

Attacker selects v' with the same μTag as the target address

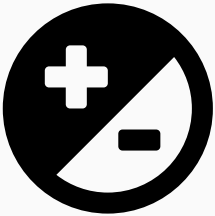
- **Phase 1: Collide**: Access v' to update the way predictor
- **Phase 2: Scheduling the victim**: The victim process is scheduled.
  - Access v: Updating the way predictor causing v' to be inaccessible from L1D
- **Phase 3: Probe**: Measure access time to v'. If v has been accessed, observe higher timing.

Collide+Probe: Pros and Cons



Pros:

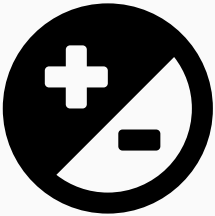
Collide+Probe: Pros and Cons



Pros:

- No knowledge of physical addresses required

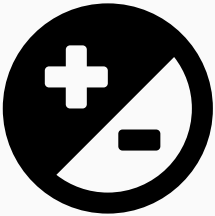
Collide+Probe: Pros and Cons



Pros:

- No knowledge of physical addresses required
- Single memory load is enough

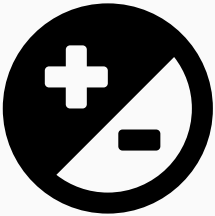
Collide+Probe: Pros and Cons



Pros:

- No knowledge of physical addresses required
- Single memory load is enough
- No clflush, no shared memory required

Collide+Probe: Pros and Cons

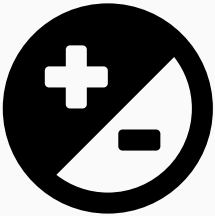


Pros:

- No knowledge of physical addresses required
- Single memory load is enough
- No clflush, no shared memory required

Cons:

Collide+Probe: Pros and Cons



Pros:

- No knowledge of physical addresses required
- Single memory load is enough
- No clflush, no shared memory required

Cons:

- Timing difference between L1D and L2 hit is small

Load+Reload

- Exploit behavior for aliased addresses, *i.e.*, virtual addresses mapping to the same physical addresses

Load+Reload

- Exploit behavior for aliased addresses, *i.e.*, virtual addresses mapping to the same physical addresses
- Evict shared data used by the sibling thread with **single load**

Load+Reload

- **Phase 1: Load**: Load address v' with the same physical tag as v.

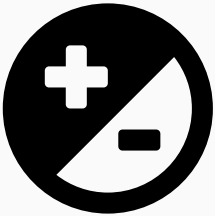
Load+Reload

- **Phase 1: Load**: Load address v' with the same physical tag as v.
- **Phase 2: Scheduling the victim**: The victim process is scheduled.
  - Access v: Sees an L1D cache miss and loads the data from L2

Load+Reload

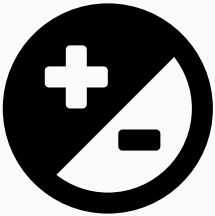
- **Phase 1: Load**: Load address v' with the same physical tag as v.
- **Phase 2: Scheduling the victim**: The victim process is scheduled.
  - Access v: Sees an L1D cache miss and loads the data from L2
- **Phase 3: Reload**: Measure access time to v'. If v has been accessed, observe higher timing.

Load+Reload: Pros and Cons



Pros:

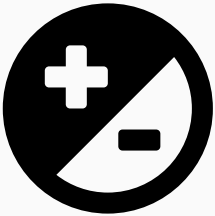
Load+Reload: Pros and Cons



Pros:

- Only evicts from L1D

Load+Reload: Pros and Cons

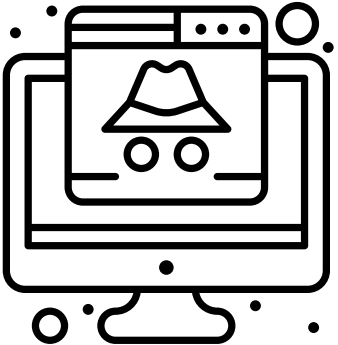


Pros:

- Only evicts from L1D

Cons:

- Only applicable in cross-thread scenario



Case Studies

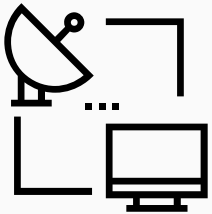
Covert Channel

- Communication Channel between two parties that are not allowed to communicate

Covert Channel

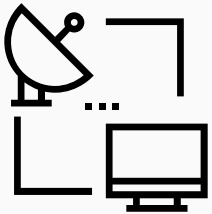
- Communication Channel between two parties that are not allowed to communicate
- Leveraging a side channel

Covert Channel



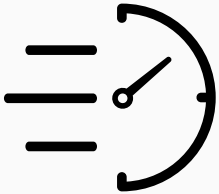
- 2 Processes accessing addresses with the same μTag and cache set
  - **Send a 1**: Access address
  - **Send a 0**: Don't access address

Covert Channel



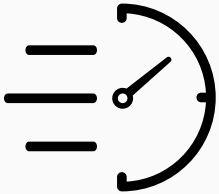
- 2 Processes accessing addresses with the same μTag and cache set
  - **Send a 1**: Access address
  - **Send a 0**: Don't access address
- Receiver measures timing difference and can deduce transmitted bit

Covert Channel



- Transmit multiple bits in parallel by utilizing multiple cache sets

Covert Channel



- Transmit multiple bits in parallel by utilizing multiple cache sets
- Maximum transmission rate of $588.9\,\mathrm{kB/s}$

Performance

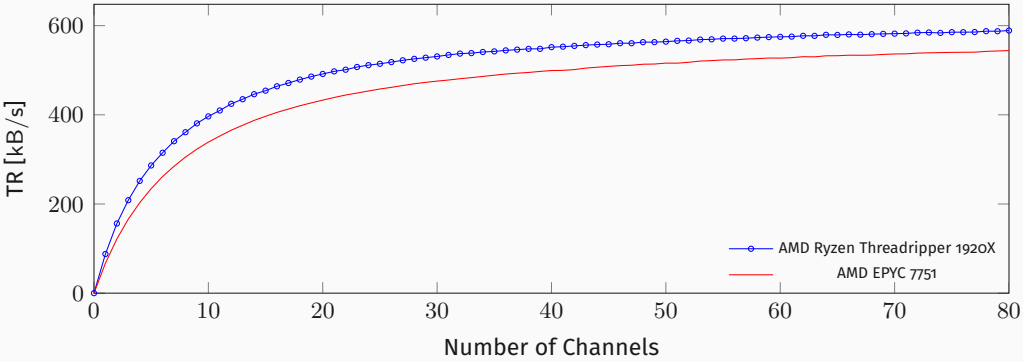


Figure 3: Mean transmission rate of the covert channels using multiple parallel channels on different CPUs.

Error Correction

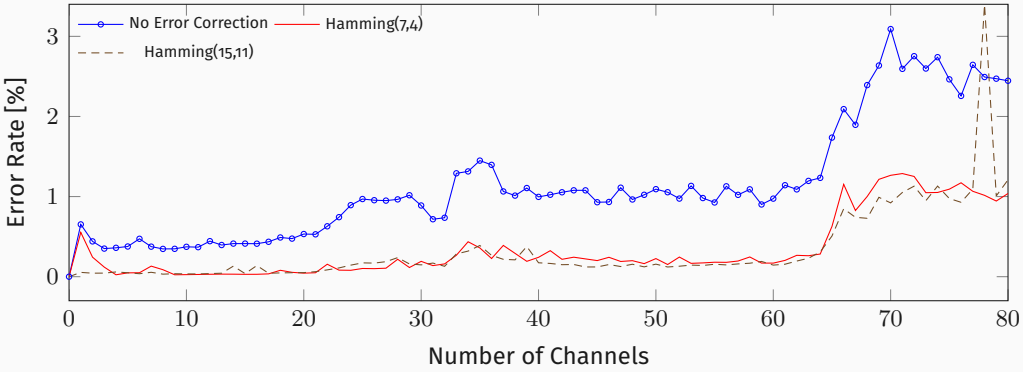
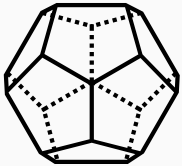


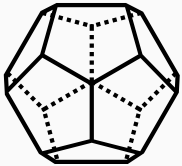
Figure 4: Error rate of the covert channel with and without error correction using different Hamming codes.

Breaking KASLR



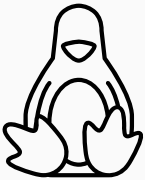
- Address Space Layout Randomization (ASLR) randomizes the location of memory sections to impede the exploitation of memory corruption vulnerabilities

Breaking KASLR



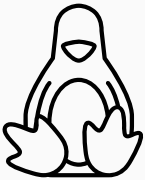
- Address Space Layout Randomization (ASLR) randomizes the location of memory sections to impede the exploitation of memory corruption vulnerabilities
- Exploit relation between μTags and virtual addresses to reduce entropy
  - Determine the μTag accessed by the victim
  - Use mapping function to infer bits of the address

Breaking KASLR



- Linux kernel is mapped using $2\,\mathrm{MB}$ pages
  - 512 different locations
  - 9 bits of entropy

Breaking KASLR



- Linux kernel is mapped using $2\,\mathrm{MB}$ pages
  - 512 different locations
  - 9 bits of entropy
- 21 lower bits of a **global variable** are identical to the offset within the kernel image

Breaking KASLR

- **Defeat KASLR**: Know offset of a global variable accessed by the kernel on a triggerable event

Breaking KASLR

- **Defeat KASLR**: Know offset of a global variable accessed by the kernel on a triggerable event
- `SYS_time` system call returns the global second counter

Breaking KASLR

- Access address v' with a specific μTag

Breaking KASLR

- Access address v' with a specific μTag
- Schedule system call accessing global variable

Breaking KASLR

- Access address v' with a specific μTag
- Schedule system call accessing global variable
- Probe μTag for address v'
  - **Conflict:** v' has the same μTag as the global variable

Breaking KASLR

- Access address v' with a specific μTag
- Schedule system call accessing global variable
- Probe μTag for address v'
  - **Conflict:** v' has the same μTag as the global variable
  - Otherwise: Repeat for different μTag

Breaking KASLR

$$v_{20} = v_{15} \oplus t_3 \qquad v_{21} = v_{16} \oplus t_4 \qquad v_{22} = v_{17} \oplus t_5$$
$$v_{23} = v_{18} \oplus t_6 \qquad v_{24} = v_{19} \oplus t_7 \qquad v_{25} = v_{14} \oplus t_2$$
$$v_{26} = v_{13} \oplus t_1 \qquad v_{27} = v_{12} \oplus t_0$$

- Compute **bits 21 to 27** using the reverse-engineered hash function

Breaking KASLR

$$v_{20} = v_{15} \oplus t_3 \qquad v_{21} = v_{16} \oplus t_4 \qquad v_{22} = v_{17} \oplus t_5$$
$$v_{23} = v_{18} \oplus t_6 \qquad v_{24} = v_{19} \oplus t_7 \qquad v_{25} = v_{14} \oplus t_2$$
$$v_{26} = v_{13} \oplus t_1 \qquad v_{27} = v_{12} \oplus t_0$$

- Compute **bits 21 to 27** using the reverse-engineered hash function
- Reduce to 4 different locations (as bits 28 and 29 remain unknown)

Breaking KASLR

$$v_{20} = v_{15} \oplus t_3 \qquad v_{21} = v_{16} \oplus t_4 \qquad v_{22} = v_{17} \oplus t_5$$
$$v_{23} = v_{18} \oplus t_6 \qquad v_{24} = v_{19} \oplus t_7 \qquad v_{25} = v_{14} \oplus t_2$$
$$v_{26} = v_{13} \oplus t_1 \qquad v_{27} = v_{12} \oplus t_0$$

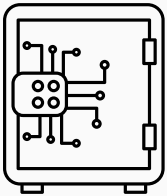
- Compute **bits 21 to 27** using the reverse-engineered hash function
- Reduce to 4 different locations (as bits 28 and 29 remain unknown)
- Average runtime of 0.51 ms and success rate of $98.5\,\%$

Other scenarios

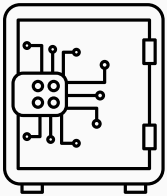
- **Hypervisor**: Leak base address of the KVM kernel module from a guest virtual machine
- **JavaScript**: Recover address bits of start of a typed array
- **User space**: Break heap ASLR using a high-speed API interface

Attacking AES T-Tables

- AES T-Tables: Fast software implementation

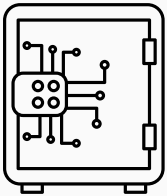


Attacking AES T-Tables



- AES T-Tables: Fast software implementation
- Uses pre-computed look-up tables

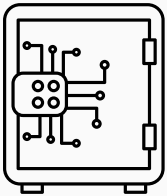
Attacking AES T-Tables



- AES T-Tables: Fast software implementation
- Uses pre-computed look-up tables
- One-round known-plaintext attack by Osvik et al. (2006) [OST06]
  - p plaintext and k secret key
  - Intermediate state $x^{(r)} = (x_0^{(r)}, \ldots, x_{15}^{(r)})$ at each round r
  - First round, accessed table indices are

$$x_i^{(0)} = p_i \oplus k_i \qquad \text{for all } i = 0, \ldots, 15$$

Attacking AES T-Tables

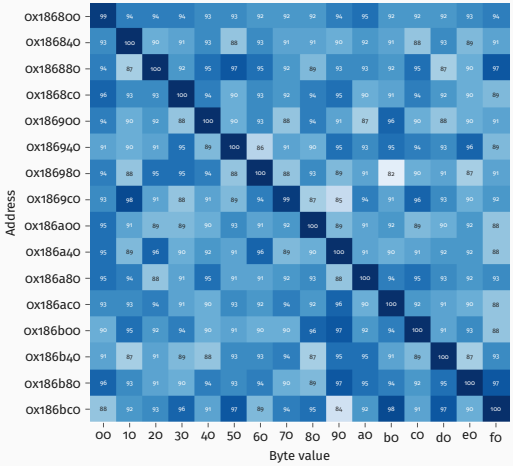


- AES T-Tables: Fast software implementation
- Uses pre-computed look-up tables
- One-round known-plaintext attack by Osvik et al. (2006) [OST06]
  - p plaintext and k secret key
  - Intermediate state $x^{(r)} = (x_0^{(r)}, \ldots, x_{15}^{(r)})$ at each round r
  - First round, accessed table indices are

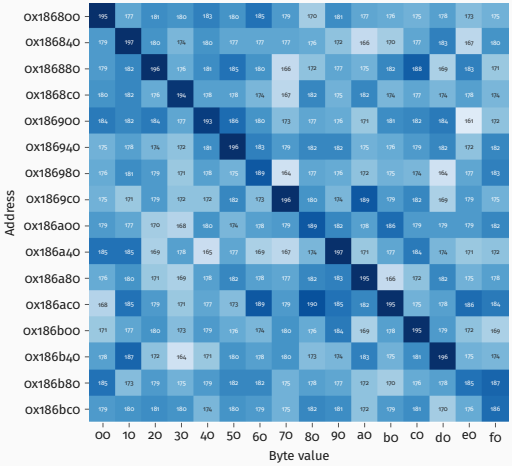
$$x_i^{(0)} = p_i \oplus k_i \qquad \text{for all } i = 0, \ldots, 15$$

$\rightarrow$ Recovering accessed table indices $\Rightarrow$ recovering the key

Attacking AES T-Tables

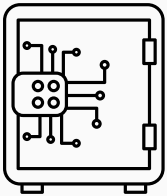


(a) Collide+Probe



(b) Load+Reload

Attacking AES T-Tables



- **Collide+Probe**: $168\,867$ encryptions per byte in $0.07\,\mathrm{s}$
- **Load+Reload**: $367\,731$ encryptions per byte in $0.53\,\mathrm{s}$
- **L1-Prime+Probe**: $450\,406$ encryptions per byte in $1.23\,\mathrm{s}$

Side-Channel Security

Chapter 3: Reverse Engineering Intel Last-Level Cache Mapping + AMD Way Predictor

Claudio Canella

March 16, 2021

Graz University of Technology

References i


Daniel Gruss et al. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016.


Koji Inoue et al. Way-predicting set-associative cache for high performance and low energy consumption. In: Symposium on Low Power Electronics and Design. 1999.


Clémentine Maurice et al. Reverse Engineering Intel Complex Addressing Using Performance Counters. In: RAID. 2015.


Dag Arne Osvik et al. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006.


Yuval Yarom et al. Mapping the Intel Last-Level Cache. In: Cryptology ePrint Archive, Report 2015/905 (2015).