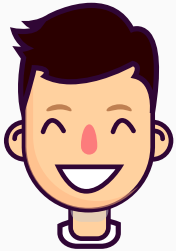
Hardware-Software Co-Design to Eliminate Cache Leakage

Lukas Giner, Daniel Gruss

June 11, 2019

Graz University of Technology

Who am I?



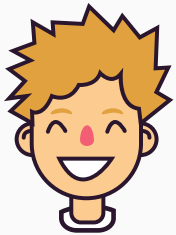
Lukas Giner

PhD student @ Graz University of Technology

@redrabbyte

lukas.giner@iaik.tugraz.at

Who am I?



Daniel Gruss

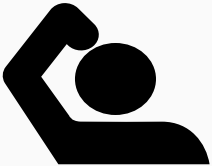
Asst. Prof. @ Graz University of Technology

@lavados

daniel.gruss@iaik.tugraz.at

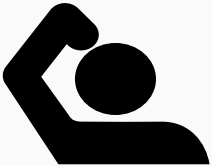

side channel
= obtaining meta-data and
deriving secrets from it
CHANGE MY MIND

Side Channel or not?



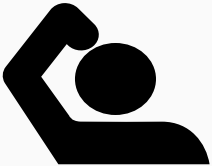
Side Channel or not?

- Profiling cache utilization with performance counters?

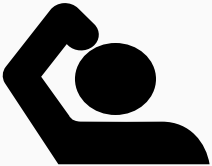


Side Channel or not?

- Profiling cache utilization with performance counters? → No



Side Channel or not?

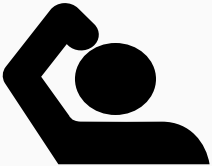


- Profiling cache utilization with performance counters? $\rightarrow$ No
- Observing cache utilization with performance counters and using it to infer a crypto key?

Side Channel or not?

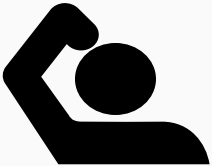
- Profiling cache utilization with performance counters? $\rightarrow$ No
- Observing cache utilization with performance counters and using it to infer a crypto key? $\rightarrow$ Yes

Side Channel or not?



- Profiling cache utilization with performance counters? $\rightarrow$ No
- Observing cache utilization with performance counters and using it to infer a crypto key? $\rightarrow$ Yes
- Measuring memory access latency with Flush+Reload?

Side Channel or not?



- Profiling cache utilization with performance counters? → No
- Observing cache utilization with performance counters and using it to infer a crypto key? → Yes
- Measuring memory access latency with Flush+Reload? → No

Side Channel or not?

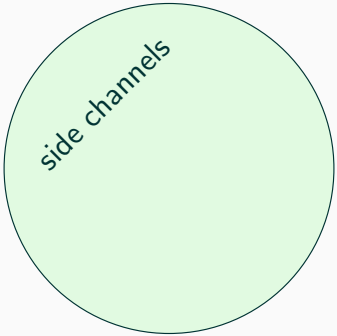
- Profiling cache utilization with performance counters? → No
- Observing cache utilization with performance counters and using it to infer a crypto key? → Yes
- Measuring memory access latency with Flush+Reload? → No
- Measuring memory access latency with Flush+Reload and using it to infer keystroke timings?

Side Channel or not?

- Profiling cache utilization with performance counters? → No
- Observing cache utilization with performance counters and using it to infer a crypto key? → Yes
- Measuring memory access latency with Flush+Reload? → No
- Measuring memory access latency with Flush+Reload and using it to infer keystroke timings? → Yes

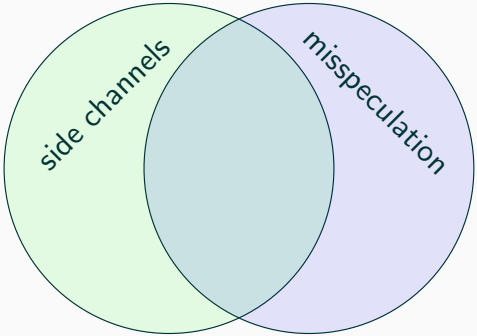
Speculative Side-Channel Attacks?

Speculative Side-Channel Attacks?



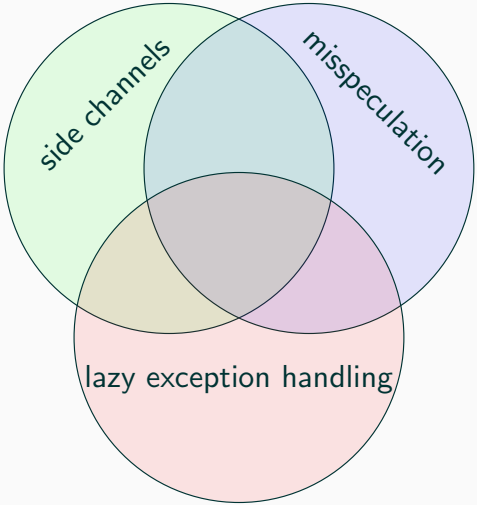
- traditional cache attacks (crypto, keys, etc)

Speculative Side-Channel Attacks?



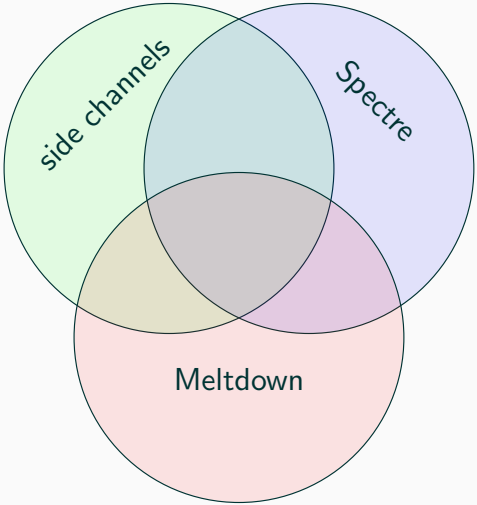
- traditional cache attacks (crypto, keys, etc)
- actual misspeculation (e.g., branch misprediction)

Speculative Side-Channel Attacks?



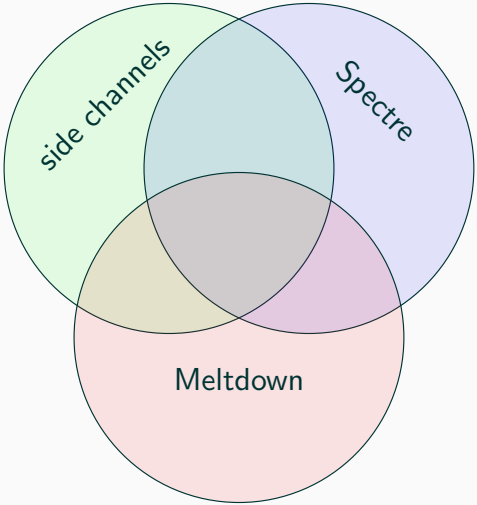
- traditional cache attacks (crypto, keys, etc)
- actual misspeculation (e.g., branch misprediction)
- Meltdown, Foreshadow, ZombieLoad, etc

Speculative Side-Channel Attacks?



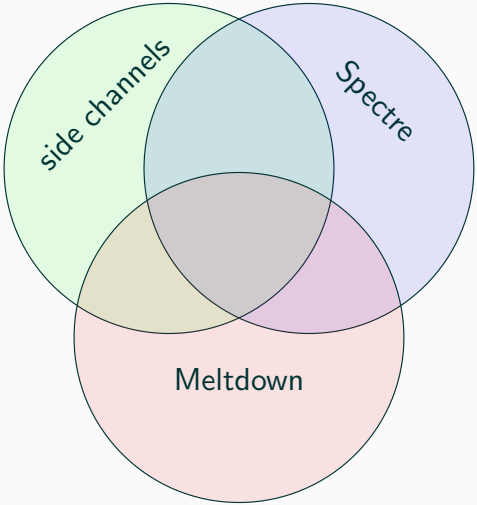
- traditional cache attacks (crypto, keys, etc)
- actual misspeculation (e.g., branch misprediction)
- Meltdown, Foreshadow, ZombieLoad, etc

Speculative Side-Channel Attacks?



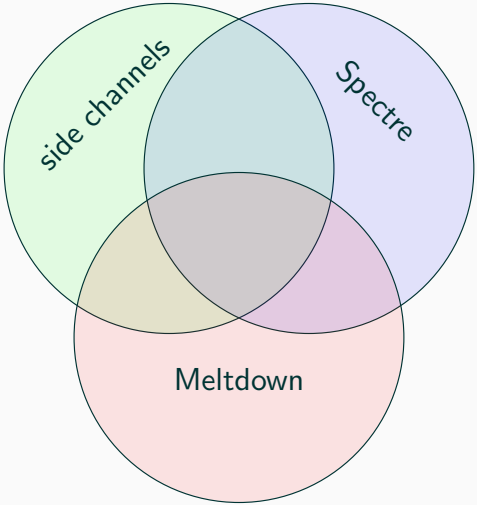
- traditional cache attacks (crypto, keys, etc)
- actual misspeculation (e.g., branch misprediction)
- Meltdown, Foreshadow, ZombieLoad, etc
- **Let's avoid the term Speculative Side-Channel Attacks**

Speculative Side-Channel Attacks?



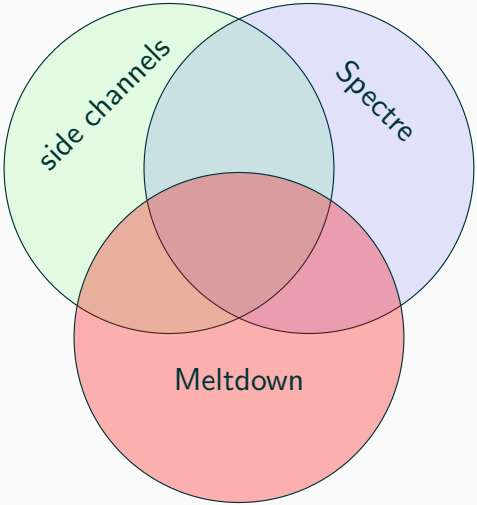
- traditional cache attacks (crypto, keys, etc)
- actual misspeculation (e.g., branch misprediction)
- Meltdown, Foreshadow, ZombieLoad, etc
- **Let's avoid the term Speculative Side-Channel Attacks**
- Let's be more precise

Speculative Side-Channel Attacks?

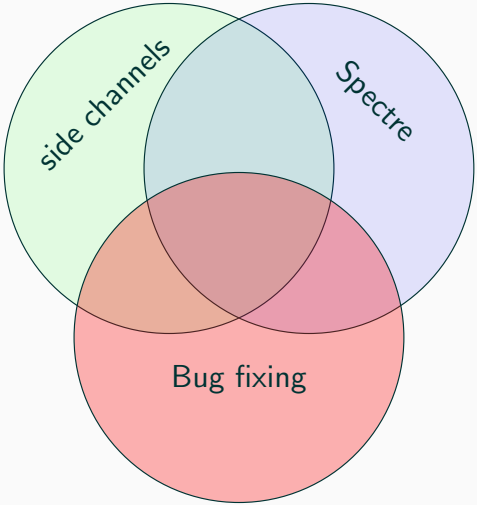


- traditional cache attacks (crypto, keys, etc)
- actual misspeculation (e.g., branch misprediction)
- Meltdown, Foreshadow, ZombieLoad, etc
- **Let's avoid the term Speculative Side-Channel Attacks**
- Let's be more precise
- $\rightarrow$ then we can think about actual mitigations

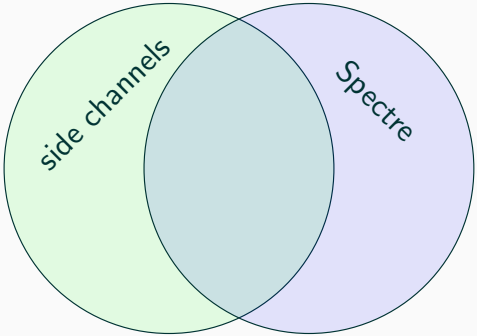
Speculative Side-Channel Attacks?



Speculative Side-Channel Attacks?



Speculative Side-Channel Attacks?



But what about these two?

- CPU Caches are a shared resource

The Problem with Cache Side Channels

- CPU Caches are a shared resource
  $\rightarrow$ there's probably a side-channel

The Problem with Cache Side Channels

- CPU Caches are a shared resource
  $\rightarrow$ there's probably a side-channel
- Cache side-channels are currently very powerful

- CPU Caches are a shared resource
  → there's probably a side-channel
- Cache side-channels are currently very powerful
- At the moment: simple, fast & reliable

The Problem with Cache Side Channels

- CPU Caches are a shared resource
  $\rightarrow$ there's probably a side-channel
- Cache side-channels are currently very powerful
- At the moment: simple, fast & reliable
- They enable a large number of attacks, e.g.:

The Problem with Cache Side Channels

- CPU Caches are a shared resource
  $\rightarrow$ there's probably a side-channel
- Cache side-channels are currently very powerful
- At the moment: simple, fast & reliable
- They enable a large number of attacks, e.g.:
  "direct" cache attacks,

The Problem with Cache Side Channels

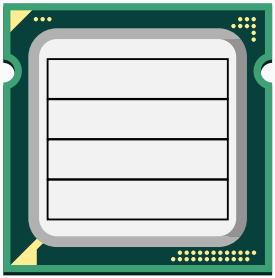
- CPU Caches are a shared resource
  → there's probably a side-channel
- Cache side-channels are currently very powerful
- At the moment: simple, fast & reliable
- They enable a large number of attacks, e.g.:
  "direct" cache attacks, Spectre & Meltdown,

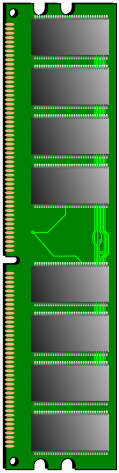
The Problem with Cache Side Channels

- CPU Caches are a shared resource
  $\rightarrow$ there's probably a side-channel
- Cache side-channels are currently very powerful
- At the moment: simple, fast & reliable
- They enable a large number of attacks, e.g.:
  "direct" cache attacks, Spectre & Meltdown, Zombieload, ...

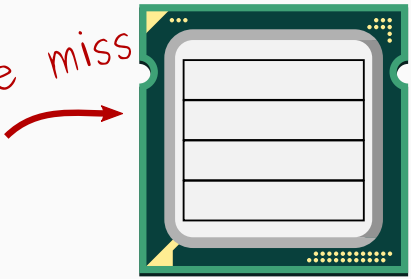
How do caches work?

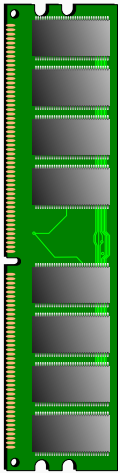
```
printf("%d", i);
printf("%d", i);
```



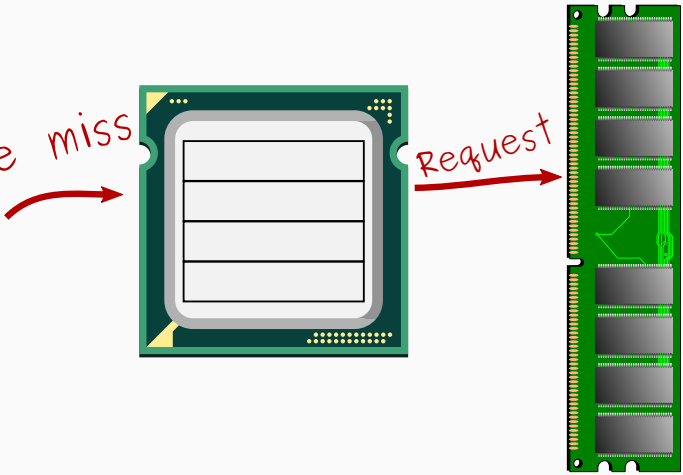


```
printf("%d", i);
printf("%d", i);
```

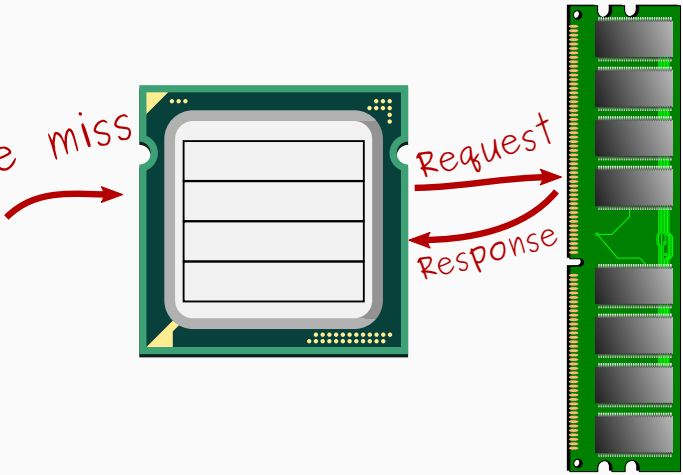




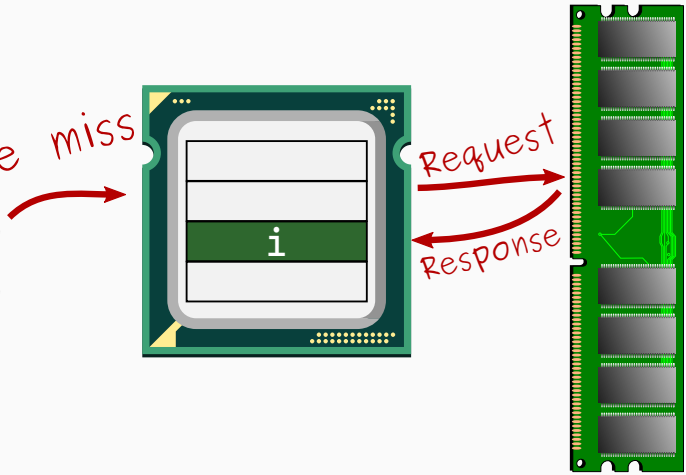
```
printf("%d", i);
printf("%d", i);
```

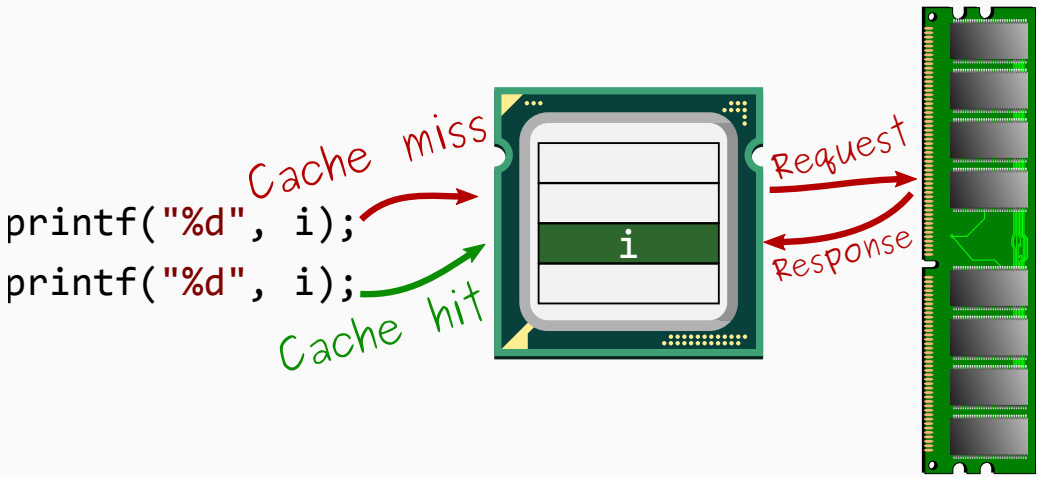


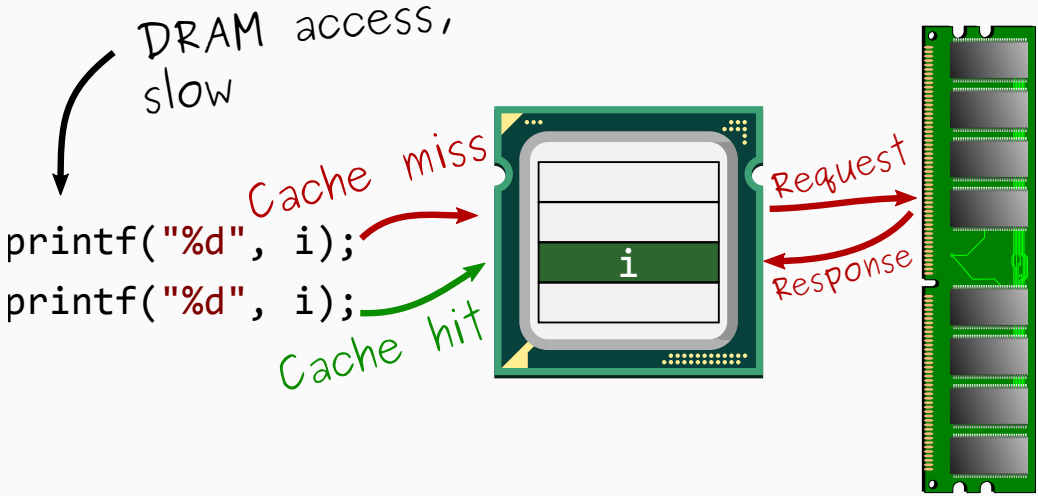
```
printf("%d", i);
printf("%d", i);
```

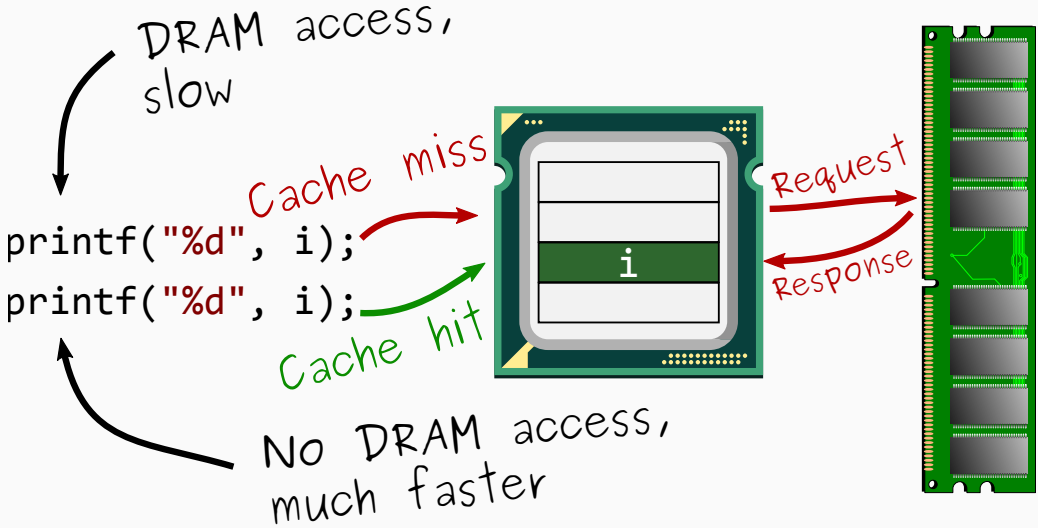


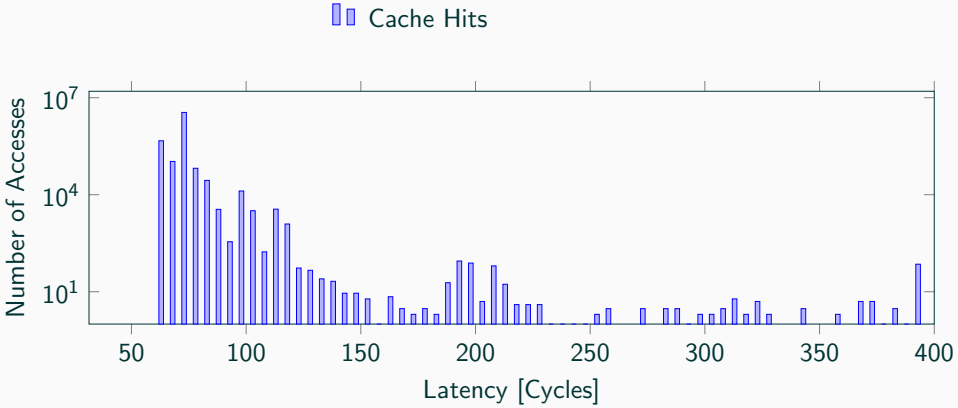
```
printf("%d", i);
printf("%d", i);
```

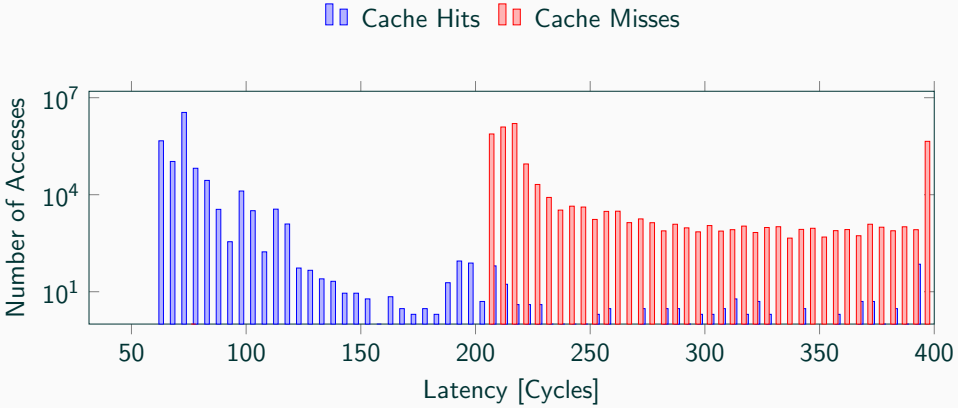



printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response

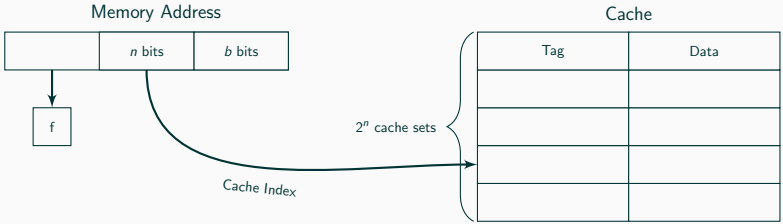

DRAM access, slow
printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response


DRAM access, slow
printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response
No DRAM access, much faster

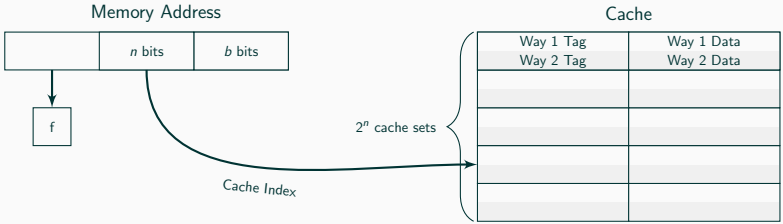

Cache Hits
Number of Accesses
10^7
10^4
10^1
50
100
150
200
250
300
350
400
Latency [Cycles]


Cache Hits
Cache Misses
Number of Accesses
10^7
10^4
10^1
50
100
150
200
250
300
350
400
Latency [Cycles]

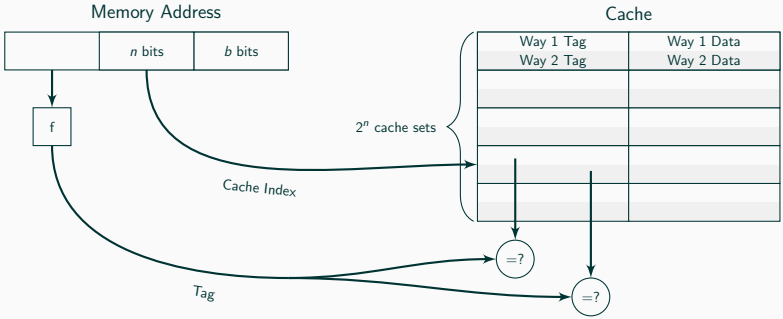
2-way set associativity



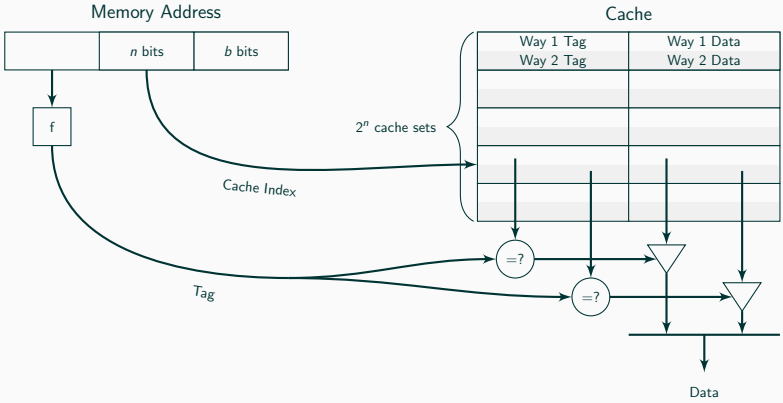
2-way set associativity



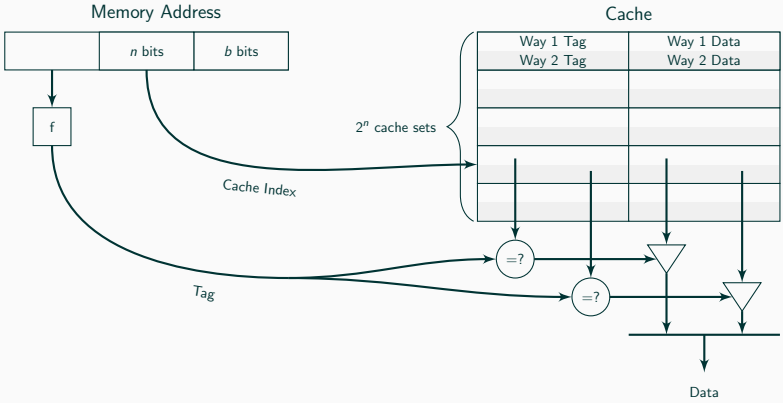
2-way set associativity



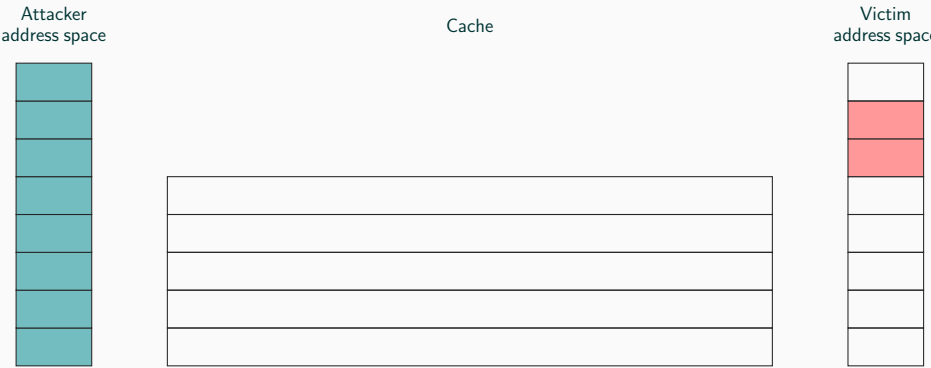
2-way set associativity

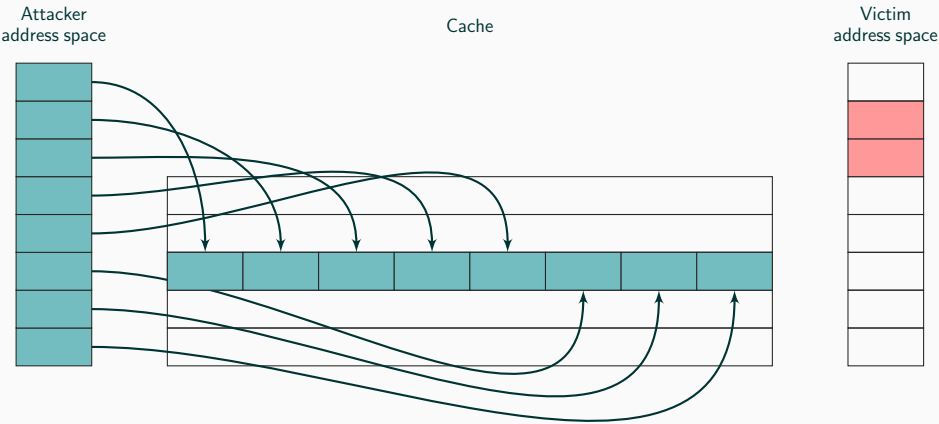


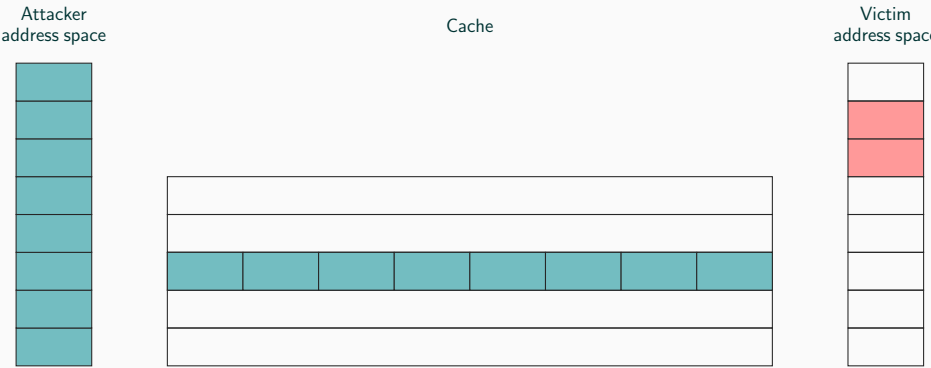
2-way set associativity

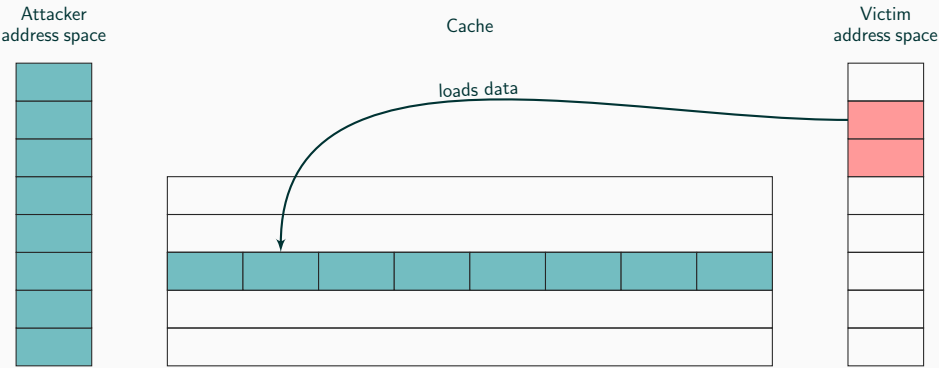


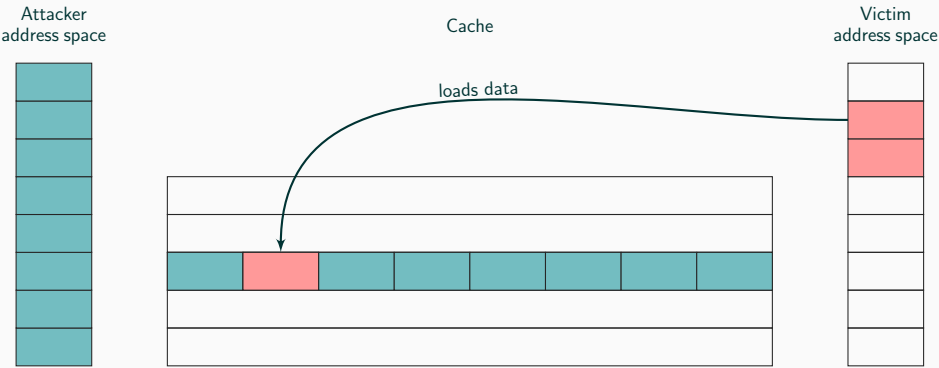
→ replacement policy

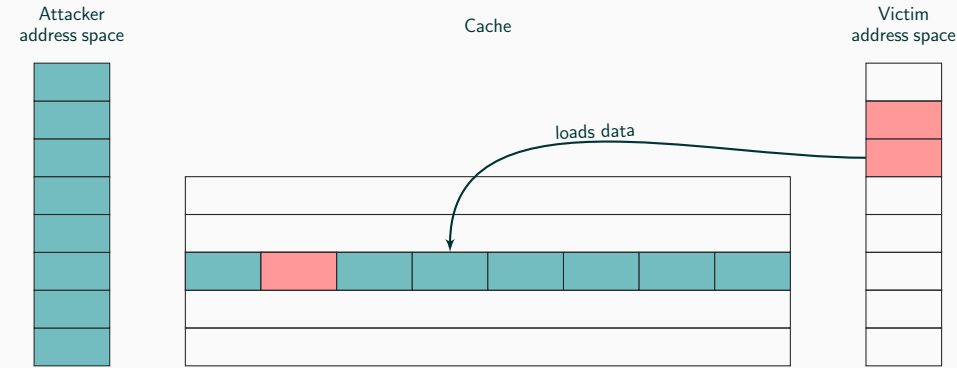

Attacker
address space
Cache
Victim
address space

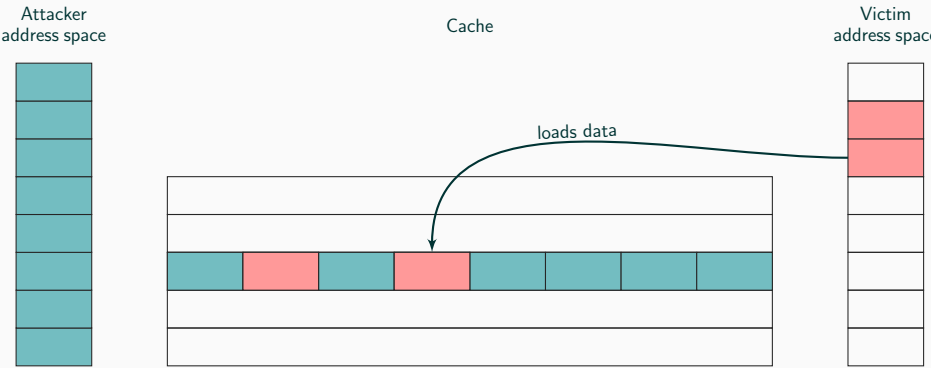

Attacker
address space
Cache
Victim
address space

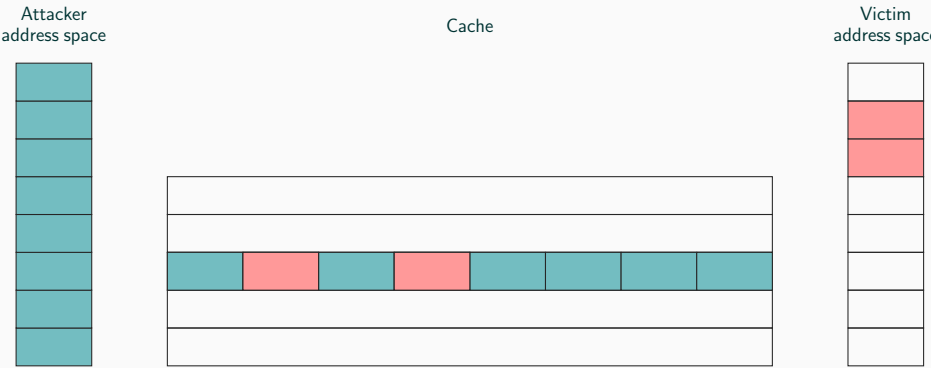

Attacker
address space
Cache
Victim
address space

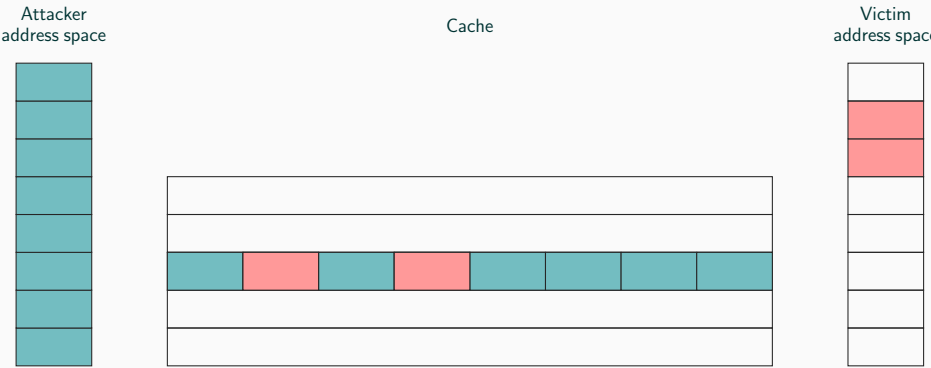

Attacker
address space
Cache
Victim
address space
loads data

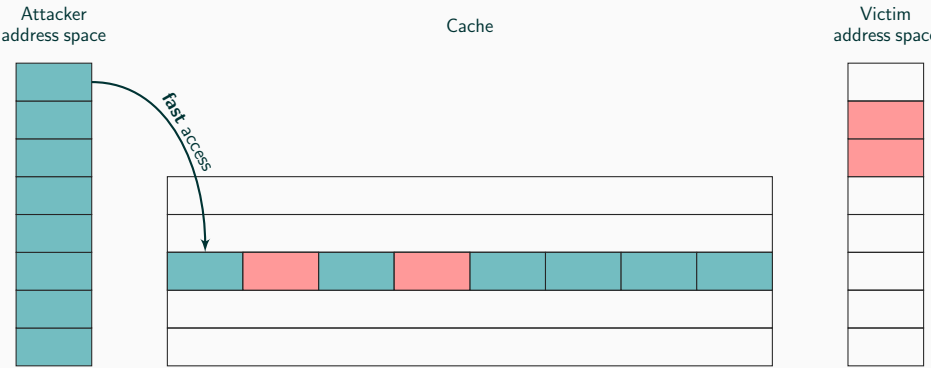

Attacker
address space
Cache
Victim
address space
loads data

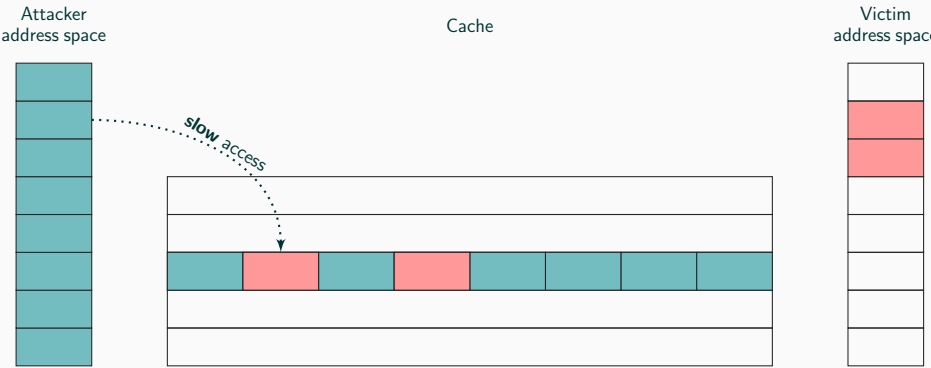

Attacker
address space
Cache
Victim
address space
loads data


Attacker
address space
Cache
Victim
address space
loads data


Attacker
address space
Cache
Victim
address space


Attacker
address space
Cache
Victim
address space


Attacker
address space
Cache
Victim
address space
fast access

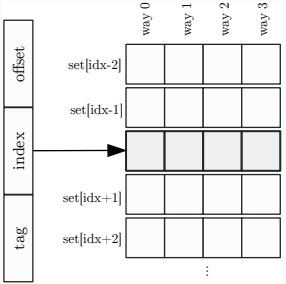

Attacker
address space
Cache
Victim
address space
slow access

ScatterCache

if cache attacks are simple because the mapping to sets is simple ..

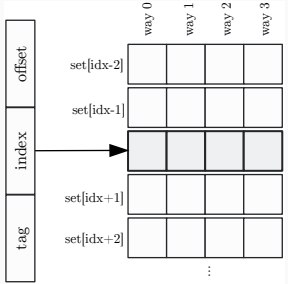
if cache attacks are simple because the mapping to sets is simple ..

instead of this:

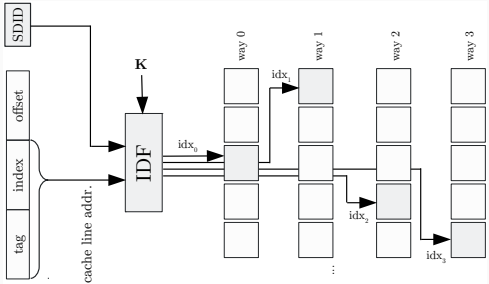


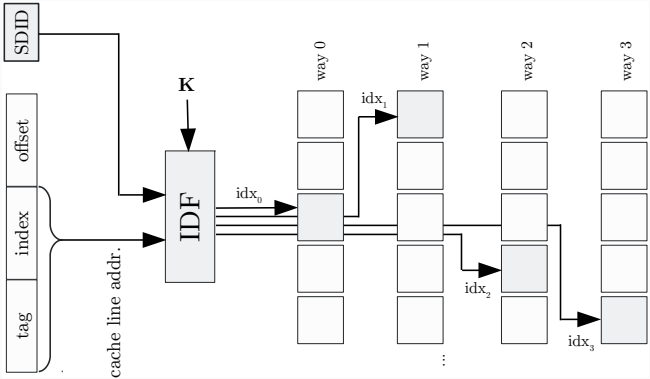
if cache attacks are simple because the mapping to sets is simple ..

instead of this:

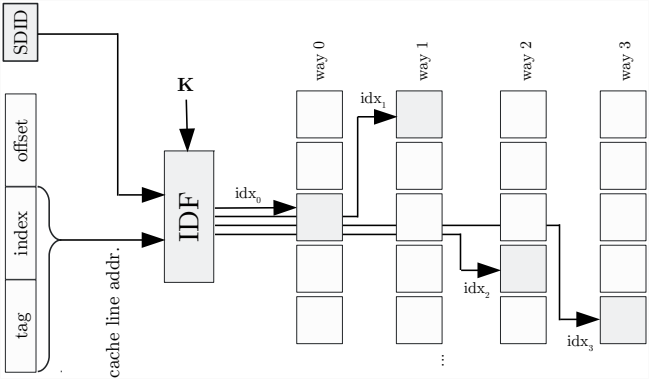


let's do this:

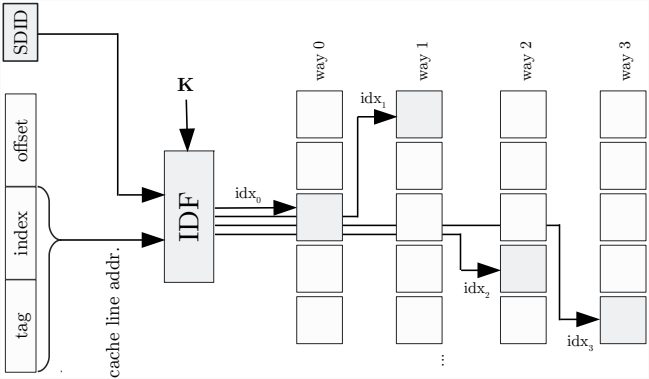




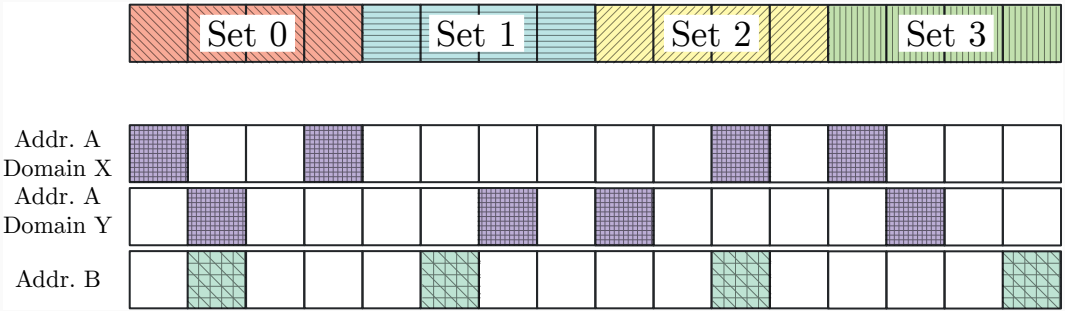
- Index Derivation Function (IDF) takes an address and returns a cache set



- Index Derivation Function (IDF) takes an address and returns a cache set
- Depends on hardware key K and optional Security Domain ID (SDID)



- Index Derivation Function (IDF) takes an address and returns a cache set
- Depends on hardware key K and optional Security Domain ID (SDID)
- → unique combination of cache lines for each address


Set 0
Set 1
Set 2
Set 3
Addr. A
Domain X
Addr. A
Domain Y
Addr. B

- ScatterCache requires no software support, default SDID = 0

- ScatterCache requires no software support, default SDID = 0
- But - OS support enables security domains

- ScatterCache requires no software support, default SDID = 0
- But - OS support enables security domains
  → shared read-only pages can be private in the cache!

- ScatterCache requires no software support, default SDID = 0
- But - OS support enables security domains
  $\rightarrow$ shared read-only pages can be private in the cache!
- OS can define SDID per process and separate user space and kernel space

- ScatterCache requires no software support, default SDID = 0
- But - OS support enables security domains
  → shared read-only pages can be private in the cache!
- OS can define SDID per process and separate user space and kernel space
- Process can request distinct SDIDs for memory ranges

- Unshared memory has no shared cache lines

- Unshared memory has no shared cache lines
  → Flush+Reload, Flush+Flush and Evict+Reload are not possible

- Unshared memory has no shared cache lines
  → Flush+Reload, Flush+Flush and Evict+Reload are not possible
- Shared, read-only memory is like unshared memory, given OS support. Without OS support, eviction-based attacks are hindered

- Unshared memory has no shared cache lines
  → Flush+Reload, Flush+Flush and Evict+Reload are not possible
- Shared, read-only memory is like unshared memory, given OS support. Without OS support, eviction-based attacks are hindered
- Shared, writable memory can't be separated, eviction-based attacks are hindered

- Specialized Prime+Probe variants are still possible

- Specialized Prime+Probe variants are still possible
- But, overlap in more than 1 cache line is very unlikely
  $\rightarrow$ Eviction is now probabilistic, $p = \frac{1}{{n_{ways}}^2}$ to evict

- Specialized Prime+Probe variants are still possible
- But, overlap in more than 1 cache line is very unlikely
  $\rightarrow$ Eviction is now probabilistic, $p = \frac{1}{{n_{ways}}^2}$ to evict
- Evicting an address with 99% certainty needs 275 addresses for 8-way cache, instead of ≈ 8 for standard Prime+Probe

- Specialized Prime+Probe variants are still possible
- But, overlap in more than 1 cache line is very unlikely
  $\rightarrow$ Eviction is now probabilistic, $p = \frac{1}{{n_{ways}}^2}$ to evict
- Evicting an address with 99% certainty needs 275 addresses for 8-way cache, instead of ≈ 8 for standard Prime+Probe
- Constructing this set requires $\approx 2^{25}$ profiled victim accesses, compared to less than 100 accesses for standard, noise-free Prime+Probe

- Micro benchmarks GAP, MiBench, Imbench, scimark2 on gem5 full system simulator

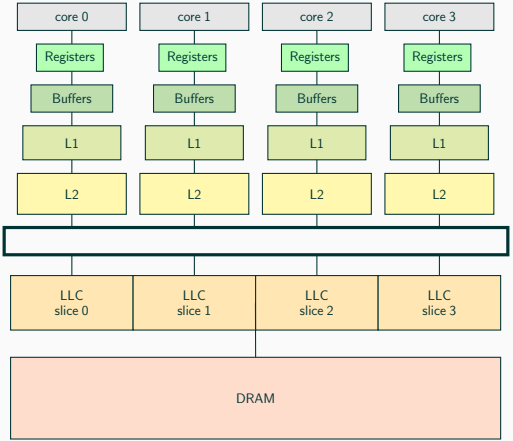
- Micro benchmarks GAP, MiBench, Imbench, scimark2 on gem5 full system simulator
- Macro benchmarks from SPEC CPU 2017 on custom cache simulator

- Micro benchmarks GAP, MiBench, Imbench, scimark2 on gem5 full system simulator
- Macro benchmarks from SPEC CPU 2017 on custom cache simulator
- Cache hit rate always at or above levels of set-associative cache with random replacement

- Micro benchmarks GAP, MiBench, Imbench, scimark2 on gem5 full system simulator
- Macro benchmarks from SPEC CPU 2017 on custom cache simulator
- Cache hit rate always at or above levels of set-associative cache with random replacement
- Typically $2\% - 4\%$ below LRU on micro benchmarks, $0\% - 2\%$ for SPEC

ConTExT

Where can data be?



ConTExT Idea

ConTExT Idea

- Mark secrets in source code

ConTExT Idea

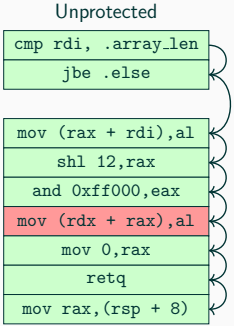
- Mark secrets in source code
- Propagate taint through memory hierarchy:

- Mark secrets in source code
- Propagate taint through memory hierarchy:
  - Pages

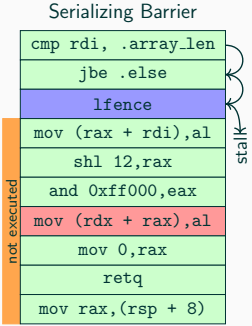
ConTExT Idea

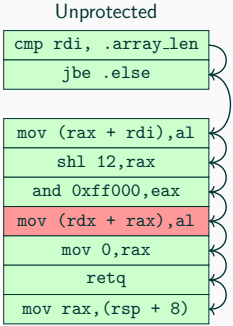
- Mark secrets in source code
- Propagate taint through memory hierarchy:
  - Pages
  - Cache Lines (in caches and buffers)

- Mark secrets in source code
- Propagate taint through memory hierarchy:
  - Pages
  - Cache Lines (in caches and buffers)
  - Registers

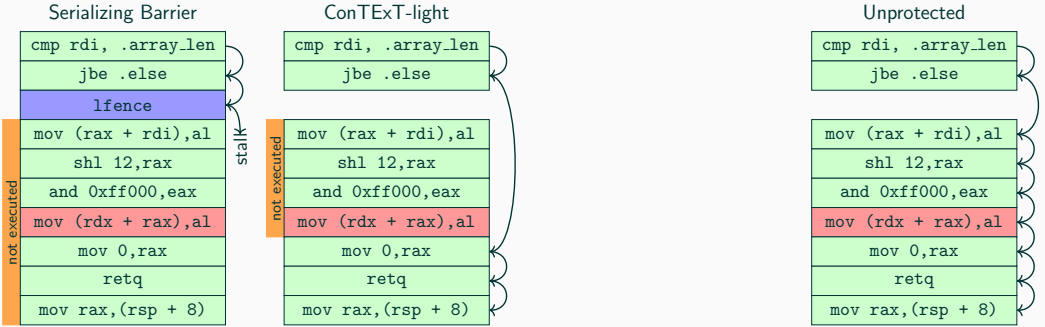

Unprotected
cmp rdi, .array_len
jbe .else
mov (rax + rdi),al
shl 12,rax
and 0xff000,eax
mov (rdx + rax),al
mov 0,rax
retq
mov rax,(rsp + 8)

ConTExT Comparison

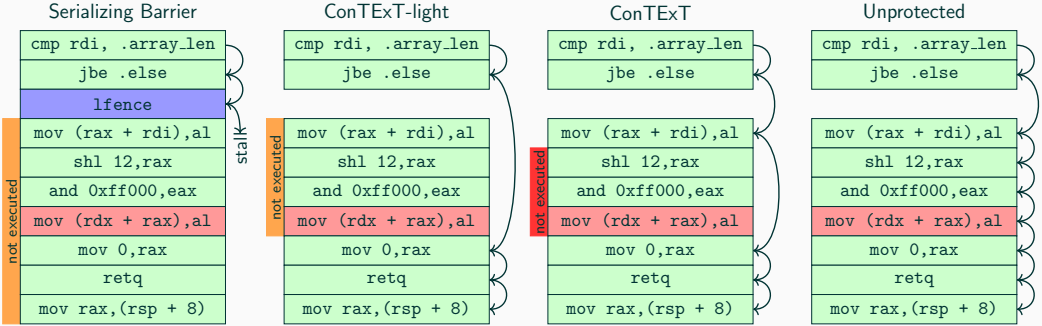




ConTExT Comparison



ConTExT Comparison



Avoiding Taint Explosion

Avoiding Taint Explosion

Avoiding Taint Explosion

- Writing to unprotected memory exposes value to attackers

- Writing to unprotected memory exposes value to attackers
  $\rightarrow$ Untaint register

- Writing to unprotected memory exposes value to attackers
  $\rightarrow$ Untaint register
- Split stack into protected and unprotected half

- Writing to unprotected memory exposes value to attackers
  $\rightarrow$ Untaint register
- Split stack into protected and unprotected half
- Stack spills of unprotected data $\rightarrow$ stay unprotected as long as they stay in the cache

Papers co-funded by Intel SCAP

C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium. 2019.

D. Gruss, E. Kraft, T. Tiwari, M. Schwarz, A. Trachtenberg, J. Hennessey, A. Ionescu, and A. Fogh. Page Cache Attacks. In: CCS. 2019.

M. Schwarz, C. Canella, L. Giner, and D. Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. In: arXiv:1905.05725 (2019).

M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: arXiv:1905.05726 (2019).

M. Schwarz, R. Schilling, F. Kargl, M. Lipp, C. Canella, and D. Gruss. ConTExT: Leakage-Free Transient Execution. In: arXiv:1905.09100 (2019).

M. Schwarz, S. Weiser, and D. Gruss. Practical Enclave Malware with Intel SGX. In: DIMVA. 2019.

M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In: USENIX Security Symposium. 2019.

Conclusions

- many attacks out there

- many attacks out there
- thorough defenses can defeat entire classes of attacks

Conclusions

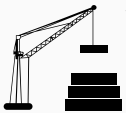
- many attacks out there
- thorough defenses can defeat entire classes of attacks
- important to distinguish between different attacks

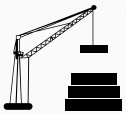
Hardware-Software Co-Design to Eliminate Cache Leakage

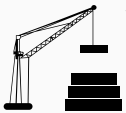
Lukas Giner, Daniel Gruss

June 11, 2019

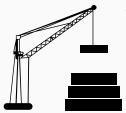
Graz University of Technology



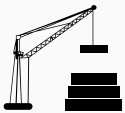




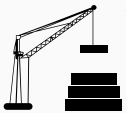
- Compiler Extension



- Compiler Extension
- Linux Patch



- Compiler Extension
- Linux Patch
- CPU Emulation in Bochs



- Compiler Extension
- Linux Patch
- CPU Emulation in Bochs
- Native via uncacheable memory (ConTExT-light)

Benchmark	SPEC Score		Overhead
	Baseline	ConTExT	[%]
600.perlbench_s	7.03	6.86	+2.42
602.gcc_s	11.90	11.80	+0.84
605.mcf_s	9.06	9.16	−1.10
620.omnetpp_s	5.07	4.81	+5.13
623.xalancbmk_s	6.06	5.95	+1.82
625.x264_s	9.25	9.25	0.00
631.deepsjeng_s	5.26	5.22	+0.76
641.leela_s	4.71	4.64	+1.48
648.exchange2_s	*would require Fortran runtime*		
657.xz_s	12.10	12.10	0.00
Average			+1.26

Table 1: Performance of the ConTExT split stack using the SPECspeed 2017 integer benchmark.