Cache side channel attacks: CPU Design as a security problem

Agenda

1) Hardware Design has security implications

2) Cache side channel attacks are a real danger

3) Despite being hardware enabled we can defend against them

Introduction

- Cache side channel attacks are attacks enabled by the micro architecturual design of the CPU.
- Probably most important side channel because of bandwith, size and central position in computer.
- Because these side channels are part of hardware design they are notoriously difficult to defeat.

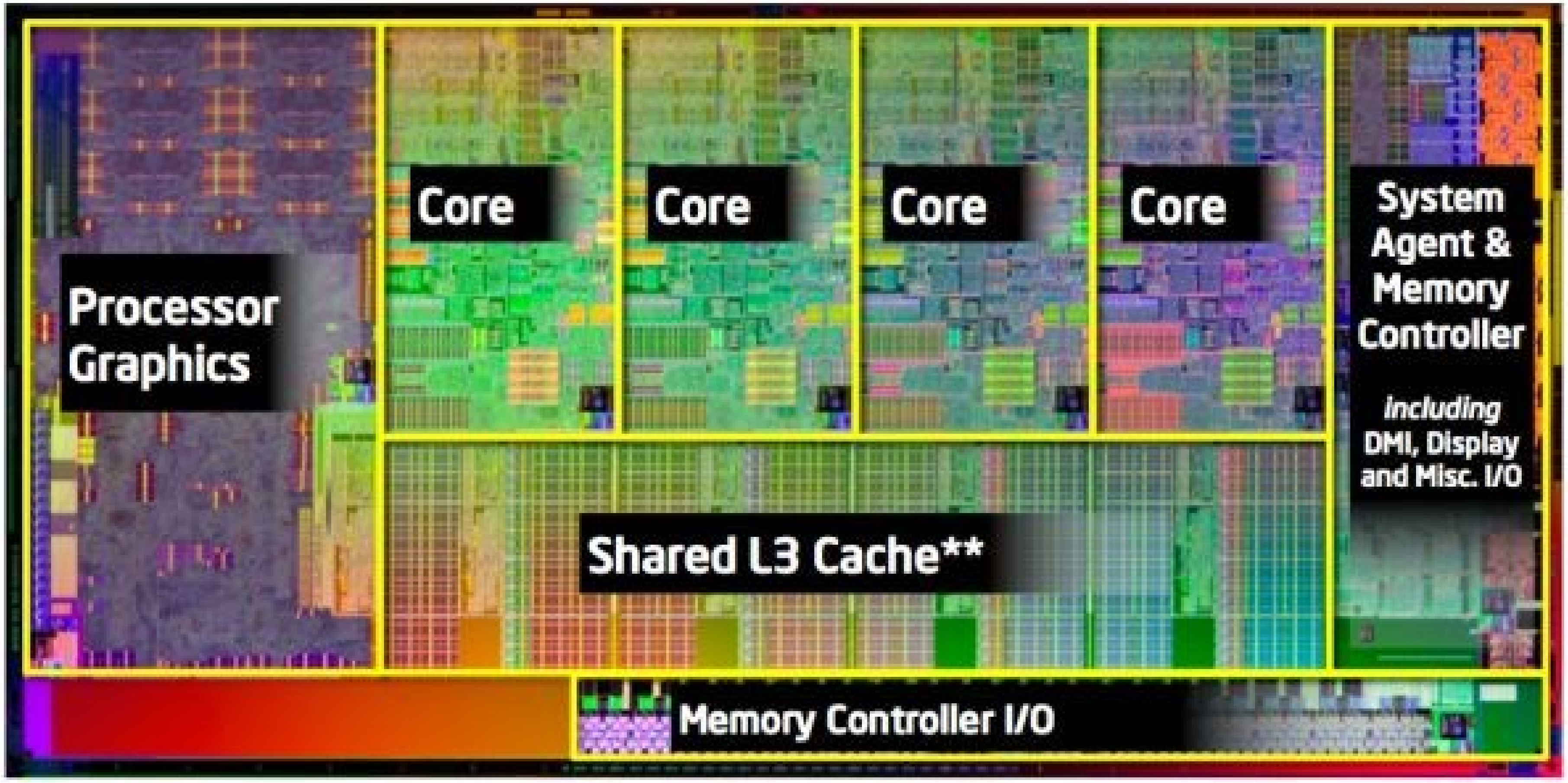
WHOAMI

- Anders Fogh
- VP Engineering and co founder Protect Software
- Been playing with malware since 1992
- Twitter: @anders_fogh
- Email: anders_fogh@hotmail.com
- Blog: http://dreamsofastone.blogspot.de/

Scope

- Modern Intel CPU's
  - Sandy bridge and newer
  - Will mention AMD cpu's from time to time
  - Cache side channel attacks work on other CPU's too
- Many caches in modern computers e.g.:
  - Translation Lookaside Buffers
  - Row buffer cache in D-Ram
  - **Data caches**
    - **Focus on shared L3 cache**

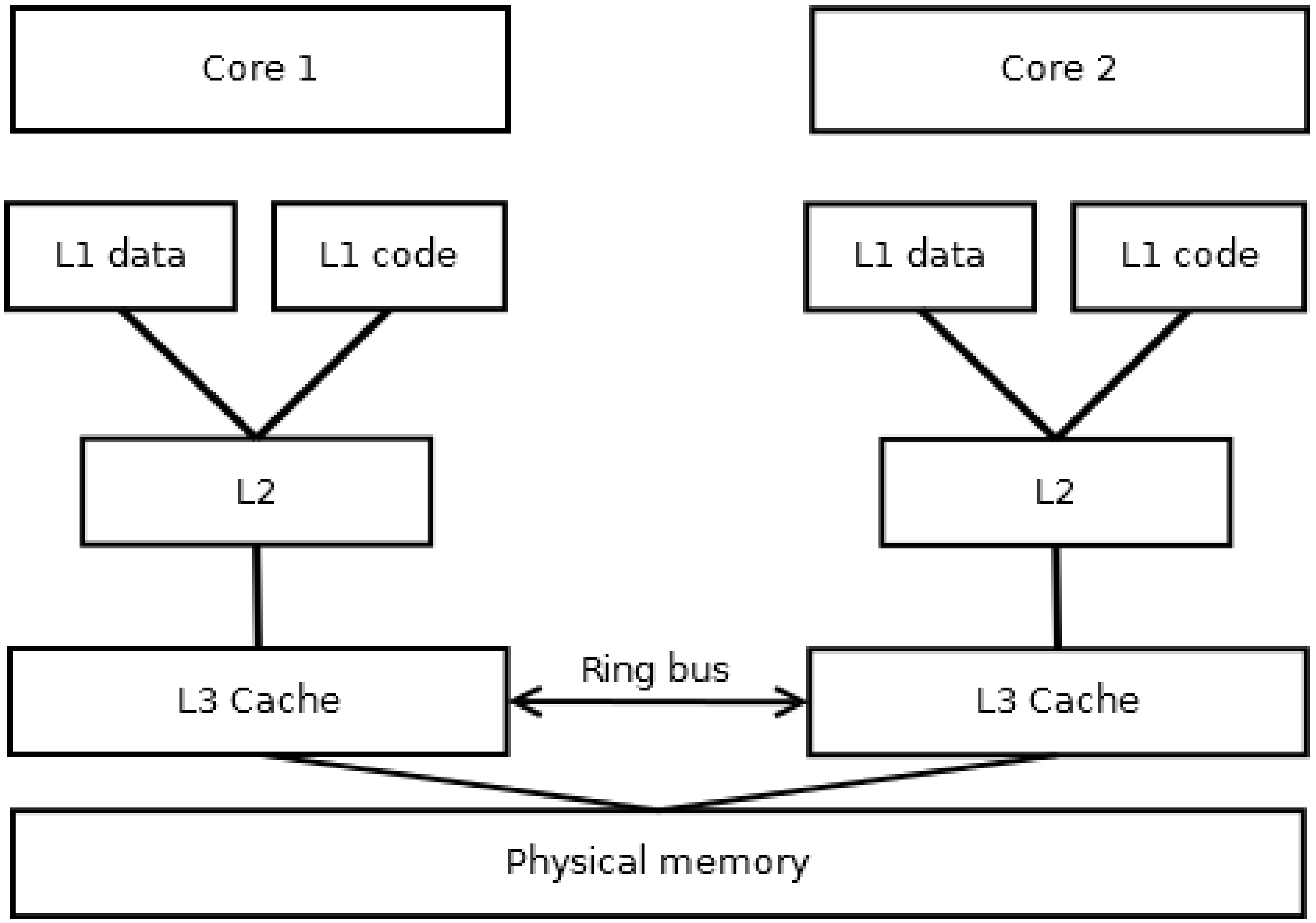
L3 is not that small



Why is this interesting

- Cache side channel attacks do not respect priveledges
- Cross CPU, Cross Core
- Cross VM, cross User, out of sandbox
- Many uses e.g.:
  - Covert channels
  - Stealing crypto keys: RSA, EDCSA, AES...
  - Spying on keyboard, mouse …
  - Breaking Kernel ASLR

How the data cache works on Intel



Important cache features

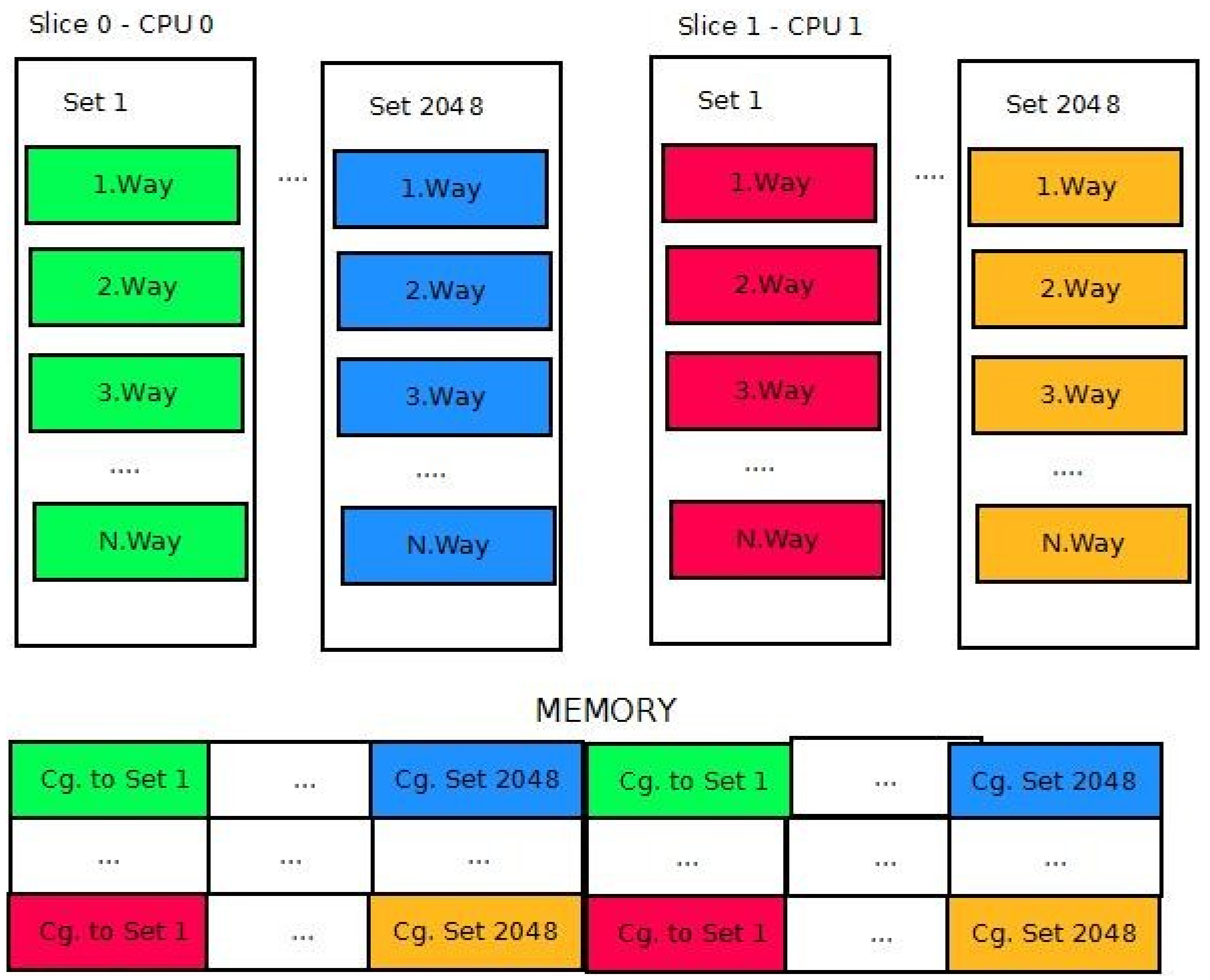
- Almost any memory read/write is placed in the cache: The cache is a mirror image of memory activity on the computer.
- L3 is shared globally across all users and priviledge levels
- Inclusive cache hierachy: If remove memory from L3, we remove it from all caches: We can manipulate the cache!

How memory is stored in L3

- Intels Problem: It takes a lot of infrastructure and time to keep track of the position of each byte of memory if it could be anywhere in the cache.

- Works on blocks called cache lines. Which typically is 64 consecutive bytes in memory.
- N-Way Set associative cache
  - Any cache line belongs to a so called cache set. Which is determined by the address.
  - There are 2048 cache sets per slice
  - Each set can store N (typically 12-20) cache lines, depending on total cache size. Each storage position is called a „way“

N-Way Set Associative L3



Example code to attack

```
WCHAR gdk_keysym_to_unicode(gkeysym Input) {
   if (IsUpper(Input)) {
     return gkeysym2unicode_UpperCase(Input);
   }
   else {
     return gkeysym2unicode_LowerCase(Input);
   }
}
```

Common for all CSC

- We can determine if something is cached
  - Speed difference: 80 CLK vs. 200 CLK
- We can manipulate the cache
  - Evict: Access memory until a given address is no longer cached
  - Flush: Remove a given address using clflush instruction
  - Prime: Place known addresses in the cache

Big 3 Cache side channel Attacks

- Evict + Time
- Prime + Probe
- Flush + Reload

They all work this way: Manipulate cache to known state, „wait“ for victim activity and examine what has changed.

Evict + Time

- Step 1. Execute a function to prime cache
- Step 2. Time the function
- Step 3. Evict a cache set
- Step 4. Time the function

If Step 2 was faster than step 4 the function probably used an address congruent to the cache set in step 3.

Prime + Probe

- Step 1. Prime a cache set to contain known attacker addresses
- Step 2. Wait for victim activity
- Step 3. Time accessing address from step 1.

If accessing memory in step 3 is slow(cache miss) victim used memory congruent with cache set in step 1.

Flush + Reload

- Step 1. Flush: Flush shared address from cache
- Step 2. Wait for victim
- Step 3. Reload: Time access for accessing the shared address:

If fast timing in 3 was fast it was placed in cache by victim. If slow victim did not use the address

Overview

- Evict + Time
  - Accuracy: Cache set congruence
  - Requires function call
  - „Post mortem“ analysis
  - Possible in java script
- Prime + Probe
  - Accuracy: cache set congruence
  - Live analysis
  - Works well in java script
- Flush + Reload
  - Accuracy: Cache line accuracy
  - Requires shared memory
  - Live analysis
  - Fastest attack → best resolution on spying
  - Does not work in JS.

Shared memory

- Shared memory not as rare as one would think.
  - Shared libraries
  - Dedublication - even on VM

Noise

- Cache side channels are inherently noisy
    - Other code than victim running
        - Use same cache lines / cache sets
        - Use other shared subsystems
    - Interupts
    - Hardware prefetcher

Solution: Repeat attack and rely on law of large numbers.

Detecting CSC

- I'll use flush+reload as an example. The method does work for the other CSC's as well.

Performance counters

- We have 18 performance counters we can program to count micro events
- Monitors a vast number of micro events.
  - For example, instructions decoded, interrupts received, **L3 cache misses**, **L3 cache references**, mis-predicted branches...
- Optionally interrupt on overflow and we can set the counter.
- We can use them to look for security related „wierdness“ (Not just CSC but also root kits, row hammer and ROP)

Flush + Reload code

```
Clflush [shared_address]
Wait() // optionally wait for victim activity
Mfence
StartTime = rdtsc()
Mov register, [shared_address]
Mfence
Information = rdtscp() - StartTime
```

Detecting flush+reload

Clflush [shared_address]

Wait() // optionally wait for victim activity

Mfence

StartTime = rdtsc()

Mov register, [shared_address]

When victim didn't use shared address we get a cache miss. They are rare in real applications. We can use this for detection with performance counters

Mfence

Information = rdtsc() - StartTime

Row hammer was flush + reload

```
Code1a:
  mov eax, [address1]
  mov ebx, [address2]
  clflush [address1]
  clflush [address2]
  jmp Code1a
```

Flush + Flush

- Idea: Flushing an address from the cache is slower when the address is actually in the cache

```
Clflush [shared_address]
Wait()
Mfence
StartTime = Rdtsc()
Clflush [shared_address]
Mfence
Information = Rdtsc()-StartTime
```

Why is flush+flush Stealth

- Flush+flush does not have a reload phase.
- No mov, [shared_address] => no cache miss
- Performance counters useless?

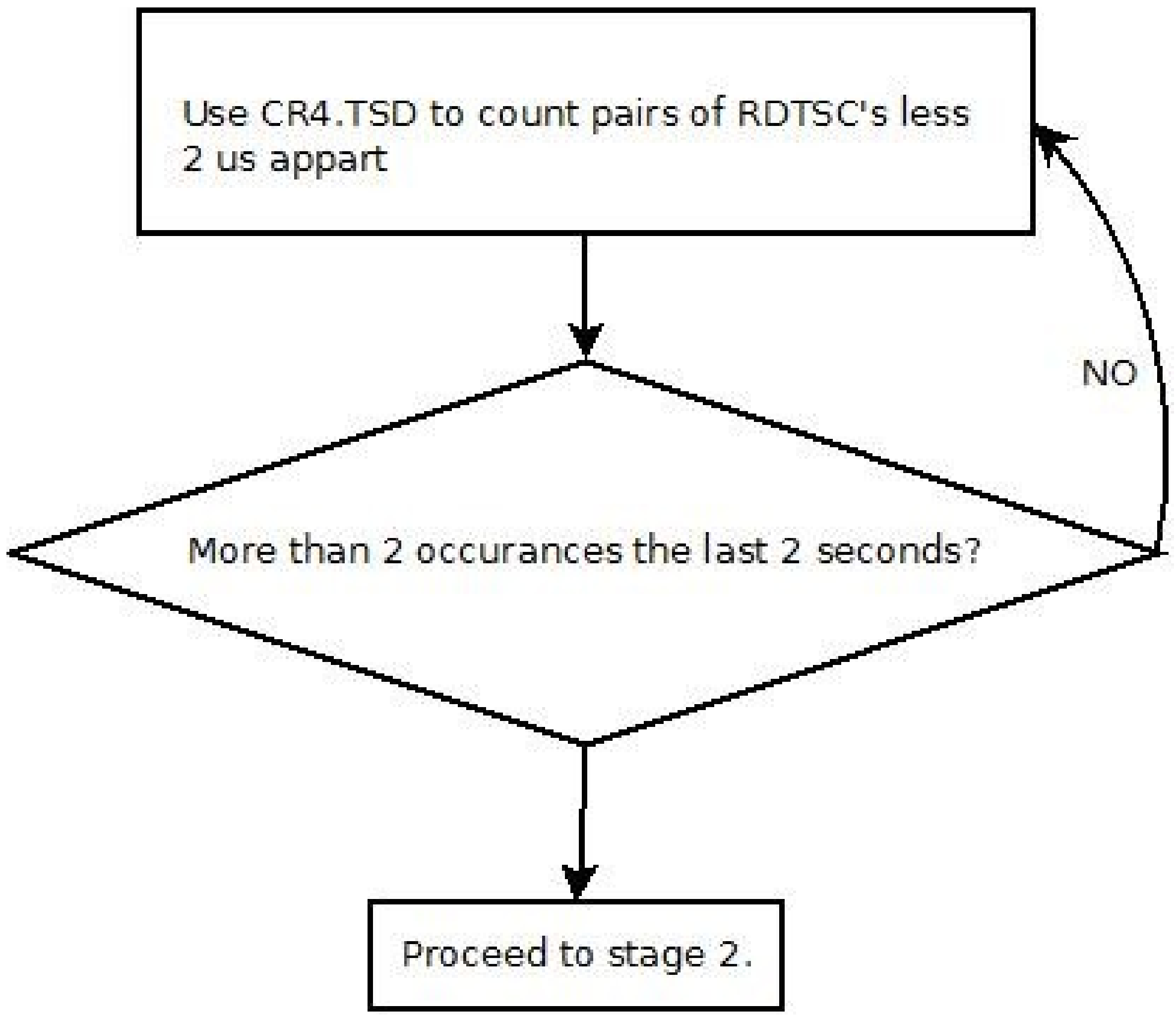
Detecting flush+flush

- Idea: flush+flush always have this structure: a clflush instruction bracketed by high resolution timers (Rdtsc) in short sequence.
- Can we detect this? Yes.

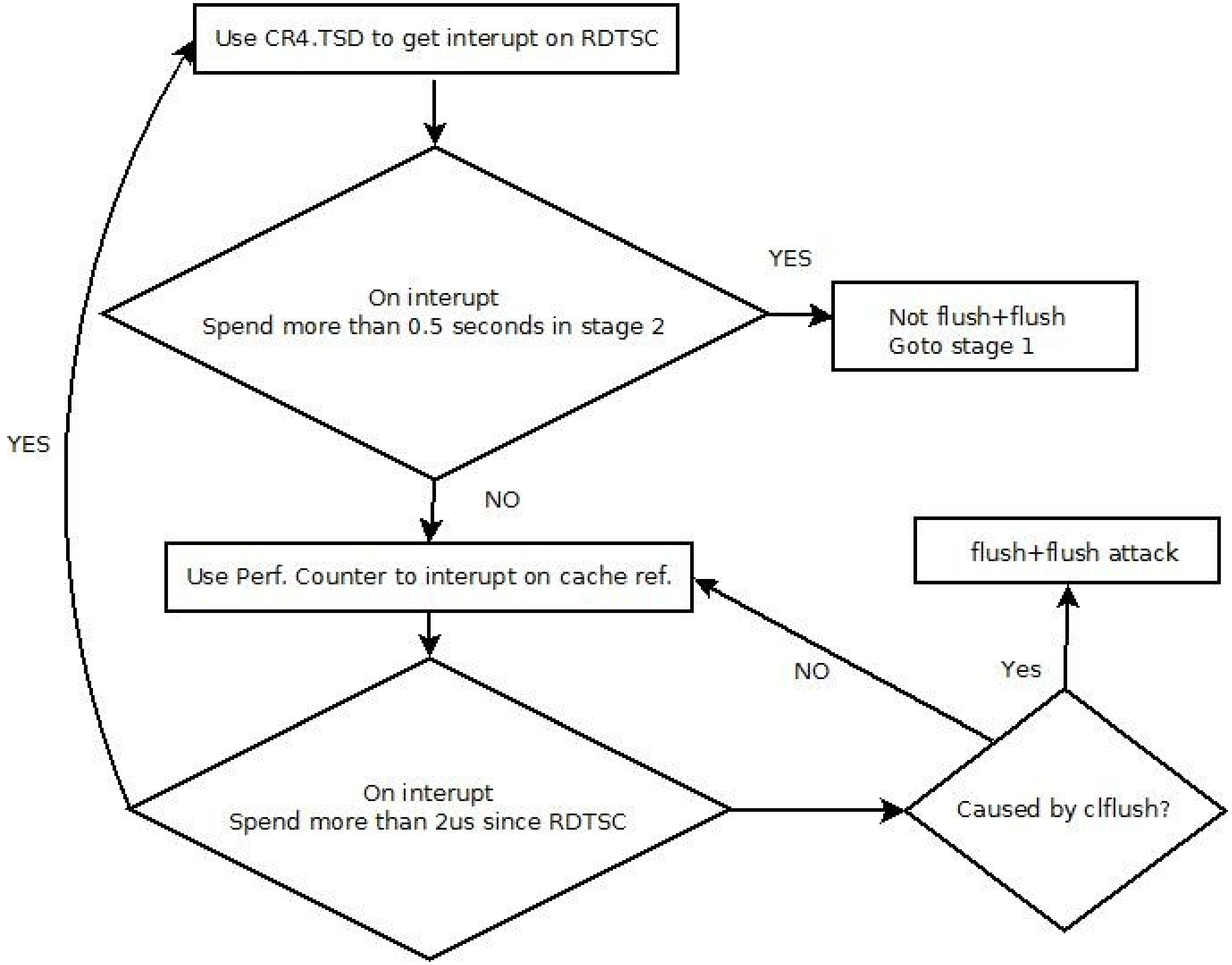
Detecting flush+flush

1) Attack is not cross VM-dedublication enabled
2) RDTSC can become priviledged by setting CR4.TSD flag: We can get an access violation if this instruction is used
3) Attacker cannot spend much time between RDTSC because execution time on multi core CPUs depend on things attacker cannot observe. => wait too long, too much noise to pick up signal.
4) When Clflush actually flushes an address a cache reference event is generated.

Stage 1


Use CR4.TSD to count pairs of RDTSC's less 2 us appart
More than 2 occurances the last 2 seconds?
NO
Proceed to stage 2.

Stage 2



Use CR4.TSD to get interupt on RDTSC
On interupt
Spend more than 0.5 seconds in stage 2
YES
Not flush+flush
Goto stage 1
NO
YES
Use Perf. Counter to interupt on cache ref.
flush+flush attack
NO
Yes
On interupt
Spend more than 2us since RDTSC
Caused by clflush?

Problems?

- Causing interrupts penalizes performance on benign programs using rdtsc.
- Other timers
- Leakage of other timer
- Implementation: problems with PatchGuard

- Can we do better? Possibly...

Ommisions

- All the gory details was intentionally left out for time
- There are mitigations for cache side channel attacks (as opposed to detection)
- CAT
- Branch-free cache set aware code

- Closing keynote: Sophie D'Atoine's might be worth a view if you like this kind of stuff.
- If you have questions don't hesitate to ask – I love talking about this stuff.

- Thank you

Q & A

My thanks goes to:

- G-DATA Advanced Analytics
- Marion Marschalek (G-DATA adan)
- Daniel Gruss (TU Graz)
- Nishat „Joey“ Herath (Qualys)

Literature

Evict+Time

- [1] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. In CT-RSA 2006, 2006.

Flush+Reload

- [2] Y. Yarom and K. Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In USENIX Security Symposium, 2014.

Prime + Reload

- [3] D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. In CT-RSA 2006, 2006.

RSA 2048 in public cloud

- [4] M. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, B. Sunar. "Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud" http://eprint.iacr.org/2015/898.pdf

ECDSA

- [5] T. Allan, B. Brumley, K. Falkner,J. Pol, Y. Yarom. "Amplifying Side Channels Through Performance Degradation". http://eprint.iacr.org/2015/1141.pdf

AES

- [6] D. J. Bernstein. Cache-Timing Attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2004.

Literature

Keyboard

- [7] D. Gruss, R. Spreitzer, and S. Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches." In USENIX Security Symposium, 2015.

Mouse

- [8] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications." In CCS'15, 2015.

Breaking Kernel ASLR

- [9] D.Gruss, C. Maurice, A. Fogh, M. Lipp, S. Mangard. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR". Yet to be published

Rowbuffer

- [10] A. Fogh: "Rowbuffer side channel attacks". http://dreamsofastone.blogspot.de/2015/11/rowbuffer-side-channel-attacks.html

ARM Side channels

- [11]M Lipp, D Gruss, R Spreitzer, S Mangard. "ARMageddon: Last-Level Cache Attacks on Mobile Devices" http://arxiv.org/pdf/1511.04897.pdf

Machine learning with perf. counters

- [12] Marco Chiappetta, Erkay Savas & Cemal Yilmaz (2015): “Real time detection of cache-based side-channel attacks using Hardware Performance Counters”: http://eprint.iacr.org/2015/1034.pdf

Flush+Flush & Modified detection of CSC

- [13] D. Gruss, C. Maurice & K.Wagner(2015): “Flush+Flush: A Stealthier Last-Level Cache Attack”, http://arxiv.org/pdf/1511.04594v1.pdf