Cache Side-Channel Attacks and the case of Rowhammer

Daniel Gruss
IAIK, Graz University of Technology
April 28, 2016
1. Caches

2. Cache attacks

3. Cache Template Attacks

4. Rowhammer
1. Caches

2. Cache attacks

3. Cache Template Attacks

4. Rowhammer
CPU Caches

- CPU speed increases
- Latency of DRAM is too high
- Problem: DRAM is a bottleneck
CPU Caches

- CPU speed increases
- Latency of DRAM is too high
- Problem: DRAM is a bottle neck
- Solution: Caches - fast/small memory buffers
Memory accesses are cached

- Every memory reference goes through the cache
- Transparent to OS and programs
Memory Access Latency

Latency in Cycles

Number of Accesses

0 50 100 150 200 250 300 350 400 450 500+

Cached
Not Cached
Memory Access Latency

![Graph showing memory access latency with bars for cached and not cached accesses.]
Directly mapped cache

Memory Address
Directly mapped cache

Memory Address

Cache
Directly mapped cache

Memory Address

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Directly mapped cache

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tag</td>
</tr>
<tr>
<td></td>
<td>Data</td>
</tr>
</tbody>
</table>
Directly mapped cache

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $b$ bits

- Tag
- Data

$2^b$ bytes
Directly mapped cache

Memory Address

\[
\begin{array}{c|c}
\text{Tag} & \text{Data} \\
\hline
\end{array}
\]

Cache

\[2^b\text{ bytes}\]
Directly mapped cache

Memory Address

\[ \text{Tag} \quad \text{Data} \]

\[ 2^n \text{ cache lines} \]

\[ 2^b \text{ bytes} \]
Directly mapped cache

Memory Address

<table>
<thead>
<tr>
<th>$n$ bits</th>
<th>$b$ bits</th>
</tr>
</thead>
</table>

Cache

- Tag
- Data

Cache Index

$n$ cache lines

Tag

$2^n$ cache lines

Hit/Miss

$2^b$ bytes
Directly mapped cache

Problem: working on congruent addresses
2-way set associativity

Memory Address

\[
\begin{array}{cc}
\text{Tag} & \text{Data} \\
\end{array}
\]

Cache

\[\begin{array}{cc}
f & 2^n \text{ cache lines} \\
\end{array}\]

\[\begin{array}{cc}
\text{Cache Index} \\
\end{array}\]

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
2-way set associativity

Memory Address

\[ \begin{array}{c|c|c}
\text{n bits} & \text{b bits} \\
\hline
\end{array} \]

\[ f \]

Cache Index

\[ 2^n \text{ cache sets} \]

Cache

\[ \begin{array}{c|c|c}
\text{Way 1 Tag} & \text{Way 1 Data} \\
\hline
\text{Way 2 Tag} & \text{Way 2 Data} \\
\hline
\hline
\end{array} \]
2-way set associativity

Memory Address

$n$ bits  $b$ bits

$\rightarrow f$

Cache Index

$2^n$ cache sets

Tag

Cache

Way 1 Tag  Way 1 Data
Way 2 Tag  Way 2 Data

$=?$

$=?$
2-way set associativity

Memory Address

<table>
<thead>
<tr>
<th></th>
<th>( n ) bits</th>
<th>( b ) bits</th>
</tr>
</thead>
</table>

Cache Index

\( f \)

2\(^n\) cache sets

Way 2 Tag

Way 2 Data

Way 1 Tag

Way 1 Data

Tag

=?

=?

Data
2-way set associativity

Memory Address

\[
\begin{array}{c|c|c}
\text{f} & n \text{ bits} & b \text{ bits} \\
\end{array}
\]

Cache

\[
\begin{array}{c|c|c}
\text{Way 1 Tag} & \text{Way 1 Data} \\
\hline
\text{Way 2 Tag} & \text{Way 2 Data} \\
\end{array}
\]

\[2^n \text{ cache sets}\]

Tag

\[=?\]

Data

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]

\[=?!\]
Caches and Addresses

- Often use physical addresses
- Virtual-to-physical translation necessary
TLB and Paging

- Paging: memory translated page-wise from virtual to physical
- TLB (translation lookaside buffer) caches virtual to physical mapping
- TLB has some latency
TLB and Paging

- Paging: memory translated page-wise from virtual to physical
- TLB (translation lookaside buffer) caches virtual to physical mapping
- TLB has some latency
- Worst case for Cache: mapping not in TLB, need to load mapping from RAM
- Solution: Use virtual addresses instead of physical addresses
Cache indexing methods

- VIVT: Virtually indexed, virtually tagged
- PIPT: Physically indexed, physically tagged
- PIVT: Physically indexed, virtually tagged
- VIPT: Virtually indexed, physically tagged
VIVT

Virtual Address

VPN  \( n \) bits  \( b \) bits

Cache Index

\( 2^n \) cache sets

Tag

Cache

Way 1 Tag  Way 1 Data
Way 2 Tag  Way 2 Data

Data

Fast Virtual Tag is not unique (Context switches)

Shared memory more than once in cache

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
- Fast
- Virtual Tag is not unique (Context switches)
- Shared memory more than once in cache
PIPT

Virtual Address

\[ \text{Virtual Address} \]

\[ \begin{array}{c}
\text{Tag} \\
\text{Cache Index} \\
\text{TLB} \\
\end{array} \]

\[ \begin{array}{c}
\text{n bits} \\
\text{b bits} \\
\end{array} \]

Cache

\[ \text{Cache} \]

\[ \begin{array}{c}
\text{Way 1 Tag} \\
\text{Way 2 Tag} \\
\text{Way 1 Data} \\
\text{Way 2 Data} \\
\end{array} \]

\[ 2^n \text{ cache sets} \]

Tag

Data

Slow (TLB lookup for index)

Shared memory only once in cache!

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
PIPT

- Slow (TLB lookup for index)
- Shared memory only once in cache!
PIVT

Virtual Address

Cache

$2^n$ cache sets

$\text{Cache Index}$

$\text{Tag}$

$\text{Data}$

Way 1 Tag

Way 2 Tag

Way 1 Data

Way 2 Data

TLB

$n$ bits

$b$ bits

$n$ bits

$b$ bits

$f$

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
- Slow (TLB lookup for index)
- Virtual Tag is not unique (Context switches)
- Shared memory more than once in cache
VIPT

Virtual Address

VPN  \( n \) bits  \( b \) bits

TLB

PPN

f

Cache

2\(^n\) cache sets

Cache Index

Way 1 Tag  Way 2 Tag

Way 1 Data  Way 2 Data

Tag

=?

=?

Data

Fast

4 KiB pages: last 12 bits of VA and PA are equal

Using more bits has disadvantages (like VIVT)

→ Cache size \( \leq \) # ways · page size

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
### VIPT

**Virtual Address**

- **VPN**: \( n \) bits
- **Tag**: \( b \) bits

**Cache**

- **Way 1 Tag**
- **Way 2 Tag**
- **Way 1 Data**
- **Way 2 Data**

- \( 2^n \) cache sets

**Flow**

- **f**: Cache Index
- **Tag**: =?
- **Data**: =?

### Key Points

- **Fast**
  - 4 KiB pages: last 12 bits of VA and PA are equal
  - Using more bits has disadvantages (like VIVT)

- **Cache size** \( \leq \) # ways \( \cdot \) page size
Remarks

- L1 caches: VIVT or VIPT
- L2/L3 caches: PIPT
Remarks

- L1 caches: VIVT or VIPT
- L2/L3 caches: PIPT
  → shared memory is shared in cache!
Caches today

- L1 and L2 are private
- last-level cache:
  - divided in slices
  - shared across cores
  - inclusive
  → shared memory shared is in cache, across cores!
Cache mapping

- Function $H$ that maps slices is undocumented
- Reverse-engineered by [Hund et al., 2013, Maurice et al., 2015a, Inci et al., 2015, Yarom et al., 2015]

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
Unprivileged cache maintenance

User programs can optimize cache usage:

- **prefetch**: suggest CPU to load data into cache
- **clflush**: throw out data from all caches

... based on virtual addresses
1. Caches

2. Cache attacks

3. Cache Template Attacks

4. Rowhammer
Cache attacks

- cache-based keylogging
- crypto key recovery
  - various implementations (AES, RSA, ECC, ...)
  - up to 97% of RSA key bits recovered after 1 encryption
- cross-VM, cross-core, even cross-CPU
- any architecture (Intel, AMD, ARM)
Cross-core attacks?

- exploiting the **inclusive** property
Cross-core attacks?

- exploiting the inclusive property
- last-level cache is a superset of L1 and L2
Cross-core attacks?

- exploiting the **inclusive** property
- last-level cache is a superset of L1 and L2
- data evicted from last-level cache $\rightarrow$ evicted from L1 and L2
Cross-core attacks?

- exploiting the inclusive property
- last-level cache is a superset of L1 and L2
- data evicted from last-level cache $\rightarrow$ evicted from L1 and L2
- a core can evict lines in the private L1 cache of another core
Access-driven attacks

Attacker monitors its own activity to determine cache lines or sets accessed by victim.

Flush+Reload
[Gullasch et al., 2011]
[Yarom and Falkner, 2014]
[Gruss et al., 2015b]

Prime+Probe
[Percival, 2005]
[Liu et al., 2015]
[Maurice et al., 2015b]

Same techniques for covert and side channels
Flush+Reload

step 0: attacker maps shared library → shared memory, shared in cache
Flush+Reload

**step 0**: attacker maps shared library $\rightarrow$ shared memory, shared in cache
Flush+Reload

**step 0**: attacker maps shared library → shared memory, shared in cache

**step 1**: attacker flushes the shared line
Flush+Reload

**step 0**: attacker maps shared library $\rightarrow$ shared memory, shared in cache

**step 1**: attacker flushes the shared line

**step 2**: victim loads data while performing encryption
Flush+Reload

**Attacker address space**

**Cache**

**Victim address space**

**step 0**: attacker maps shared library → shared memory, shared in cache

**step 1**: attacker flushes the shared line

**step 2**: victim loads data while performing encryption

**step 3**: attacker reloads data → fast access if the victim loaded the line

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
Prime+Probe

Attacker address space

Cache

Victim address space

**step 0**: attacker fills the cache (prime)
Prime+Probe

**step 0**: attacker fills the cache (prime)
Prime+Probe

step 0: attacker fills the cache (prime)
step 0: attacker fills the cache (prime)
step 1: victim evicts cache lines while performing encryption
Prime+Probe

**step 0**: attacker fills the cache (prime)
**step 1**: victim evicts cache lines while performing encryption
**Prime+Probe**

**Attacker address space**

**Cache**

**Victim address space**

**step 0**: attacker fills the cache (prime)

**step 1**: victim evicts cache lines while performing encryption
Prime+Probe

**step 0**: attacker fills the cache (prime)

**step 1**: victim evicts cache lines while performing encryption
Prime+Probe

**Attacker address space**

**Cache**

**Victim address space**

**step 0**: attacker fills the cache (prime)

**step 1**: victim evicts cache lines while performing encryption
Prime+Probe

Step 0: Attacker fills the cache (prime)
Step 1: Victim evicts cache lines while performing encryption
Step 2: Attacker probes data to determine if the set was accessed
Prime+Probe

**Step 0:** attacker fills the cache (prime)
**Step 1:** victim evicts cache lines while performing encryption
**Step 2:** attacker probes data to determine if the set was accessed
Prime+Probe

**step 0**: attacker fills the cache (prime)
**step 1**: victim evicts cache lines while performing encryption
**step 2**: attacker probes data to determine if the set was accessed
1. Caches

2. Cache attacks

3. Cache Template Attacks

4. Rowhammer
Can we spy on gedit?

gedit source code tells us `gedit_window_get_active_view();` is used when a key is pressed, etc.
Can we spy on gedit?

`objdump -S /usr/bin/gedit`
Can we spy on gedit?

`objdump -S /usr/bin/gedit`

```
0000000000449ab0 <gedit_window_get_active_view>:
   449ab0: 53 : push %rbx
   449ab1: 48 89 fb : mov %rdi,%rbx
   449ab4: e8 87 df ff ff : callq 447a40 <gedit_window_get_type>
   449ab9: 48 85 db : test %rbx,%rbx
   449abc: 74 1c : je 449ada <gedit_window_get_active_view+0x2a>
   449abe: 48 85 db : test %rbx,%rbx
   449ac1: 48 85 d2 : test %rdx,%rdx
   449ac4: 74 05 : je 449acb <gedit_window_get_active_view+0x1b>
```

Round 449ab0 up to be cache-line-aligned: 449ac0
Can we spy on gedit?
objcdump -S /usr/bin/gedit

0000000000449ab0 <gedit_window_get_active_view>:
  449ab0:  53 : push %rbx
  449ab1: 48 89 fb : mov %rdi,%rbx
  449ab4: e8 87 df ff ff : callq 447a40 <gedit_window_get_type>
  449ab9: 48 85 db : test %rbx,%rbx
  449abc: 74 1c : je 449ada <gedit_window_get_active_view>
  449abe: 48 8b 13 : mov (%rbx),%rdx
  449ac1: 48 85 d2 : test %rdx,%rdx
  449ac4: 74 05 : je 449acb <gedit_window_get_active_view>

Round 449ab0 up to be cache-line-aligned: 449ac0

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
Where are my functions!?

449ac0 is a virtual address! What now?

Where in the binary is this function?
Where are my functions!?

readelf -a /usr/bin/gedit
readelf -a /usr/bin/gedit

Program Headers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Offset</th>
<th>VirtAddr</th>
<th>PhysAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FileSiz</td>
<td>MemSiz</td>
</tr>
<tr>
<td>LOAD</td>
<td>0x000000</td>
<td>0x400000</td>
<td>0x400000</td>
</tr>
<tr>
<td></td>
<td>0x089314</td>
<td>0x089314</td>
<td>R E</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Where are my functions!?

readelf -a /usr/bin/gedit

Program Headers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Offset</th>
<th>VirtAddr</th>
<th>PhysAddr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FileSiz</td>
<td>MemSiz</td>
</tr>
<tr>
<td>LOAD</td>
<td>0x000000</td>
<td>0x400000</td>
<td>0x400000</td>
</tr>
<tr>
<td></td>
<td>0x089314</td>
<td>0x089314</td>
<td>R E</td>
</tr>
</tbody>
</table>

0x400000–0x489314 is 0x0–0x89314 in the binary
Where are my functions!?

`readelf -a /usr/bin/gedit`

Program Headers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Offset</th>
<th>VirtAddr</th>
<th>PhysAddr</th>
<th>Flags</th>
<th>Align</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...  

LOAD 0x000000 0x400000 0x400000 0x089314 0x089314 R E 200000 0x49ac0

0x400000–0x489314 is 0x0–0x89314 in the binary

→ we attack 0x49ac0
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?

File Edit View Search Terminal Help

dgruss@t420dg:~/cache/sc/generic (git)-[master] % /usr/lib/x86_64-linux-gnu/libgdk-3.so 0x2d0f0
key down (after 3358683 cycles), t= 668402717 ns
Key up (after 151936 cycles), t= 659474876 ns
key down (after 428999 cycles), t= 7893566978 ns
key up (after 132857 cycles), t= 8652147911 ns
key down (after 346702 cycles), t= 8716639893 ns

File Edit View Search Terminal Help
dgruss@t420dg: ~ % sudo
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?

```
key up (after 133242 cycles), t= 9814428341 ns
key down (after 322659 cycles), t= 10415918868 ns
key up (after 165316 cycles), t= 10726619974 ns
key down (after 224248 cycles), t= 1118366573 ns
key up (after 132854 cycles), t= 11473800443 ns
key down (after 251339 cycles), t= 11978998613 ns
key up (after 131647 cycles), t= 12219441046 ns
key down (after 366253 cycles), t= 12993870546 ns
key up (after 142718 cycles), t= 13173811741 ns
```
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?

![Image of terminal output showing keystroke logging and sudo password entry.]

- Key down (after 224248 cycles), t= 11183656573 ns
- Key up (after 132054 cycles), t= 11473000443 ns
- Key down (after 251339 cycles), t= 11970898613 ns
- Key up (after 131647 cycles), t= 12215441046 ns
- Key down (after 366253 cycles), t= 12993870546 ns
- Key up (after 142718 cycles), t= 13173611741 ns
- Key down (after 1310221 cycles), t= 15681968205 ns
- Key up (after 194053 cycles), t= 15888892301 ns
- Key down (after 2668233 cycles), t= 19849303484 ns

```bash
dgruss@t420dg:~ % sudo zsh
[sudo] password for dgruss: 
```
A cache-based keystroke logger?
A cache-based keystroke logger?

<table>
<thead>
<tr>
<th>Event</th>
<th>Duration</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key down</td>
<td>251,339</td>
<td>1,197,089,061.3 ns</td>
</tr>
<tr>
<td>Key up</td>
<td>131,647</td>
<td>1,221,944,104.6 ns</td>
</tr>
<tr>
<td>Key down</td>
<td>366,253</td>
<td>1,293,870,546.9 ns</td>
</tr>
<tr>
<td>Key up</td>
<td>142,718</td>
<td>1,317,301,174.1 ns</td>
</tr>
<tr>
<td>Key down</td>
<td>131,022</td>
<td>1,568,196,820.5 ns</td>
</tr>
<tr>
<td>Key up</td>
<td>184,853</td>
<td>1,588,862,230.1 ns</td>
</tr>
<tr>
<td>Key down</td>
<td>266,823</td>
<td>1,984,930,348.4 ns</td>
</tr>
<tr>
<td>Key up</td>
<td>132,367</td>
<td>2,069,701,486.4 ns</td>
</tr>
<tr>
<td>Key down</td>
<td>189,908</td>
<td>2,945,048,359.5 ns</td>
</tr>
</tbody>
</table>

```bash
dgruss@t420dg:~ % sudo zsh
[sudo] password for dgruss:
```
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?

![Keystroke logger screenshot]

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
A cache-based keystroke logger?
Challenges

- How to locate key-dependent memory accesses?
Challenges

- How to locate key-dependent memory accesses?
- It’s complicated:
  - Large binaries and libraries (third-party code)
  - Many libraries (gedit: 60MB)
  - Closed-source / unknown binaries
  - Self-compiled binaries
Challenges

- How to locate key-dependent memory accesses?
- It’s complicated:
  - Large binaries and libraries (third-party code)
  - Many libraries (gedit: 60MB)
  - Closed-source / unknown binaries
  - Self-compiled binaries
- Difficult to find all exploitable addresses
Lessons learnt

- Staring at objdump’s for weeks will give you headache
Lessons learnt

- Staring at objdump’s for weeks will give you headache
- Automate everything right from the start
Cache Template Attacks

- Automatically find any secret-dependent cache access
- Can be used for attacks and to improve software
Cache Template Attacks

- Automatically find any secret-dependent cache access
- Can be used for attacks and to improve software
- Examples:
  - Cache-based keylogger
  - Automatic attacks on crypto algorithms
Cache Template Attacks

Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)
  - called “Cache Template”
Cache Template Attacks

Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)
  - called “Cache Template”

Exploitation Phase

- Monitor exploitable addresses
Profiling Phase

Attacker address space

Victim address space

Cache is empty
Profilng Phase

Attacker address space

Cache

Victim address space

Attacker triggers an event
Profiling Phase

Attacker checks one address for cache hits ("Reload")
Profiling Phase

Update cache hit ratio (per event and address)
Profiling Phase

Attacker address space

Cache

Victim address space

Attacker flushes shared memory
Profiling Phase

Repeat for higher accuracy
Profiling Phase

Attacker address space

Shared 0x0

Cache

Victim address space

Shared 0x0

Repeat for all events
Profiling Phase

Repeat for all events
Profiling Phase

Attacker address space

Cache

Victim address space

Shared 0x40

Continue with next address

Shared 0x40
Profiling Phase

Attacker address space

Cache

Victim address space

Shared 0x80

Continue with next address
Profiling a Single Event
Profiling a Single Event

![Image of profiling output]

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event

![Image of profiling output showing memory addresses and values]
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event

![Image of profiling and code output]

File: /usr/bin/gedit, 0x1ddc0, 0
   /usr/bin/gedit, 0x1de80, 0
   /usr/bin/gedit, 0x1de40, 0
   /usr/bin/gedit, 0x1de80, 0
   /usr/bin/gedit, 0x1dec0, 0
   /usr/bin/gedit, 0x1df00, 0
   /usr/bin/gedit, 0x1df40, 0
   /usr/bin/gedit, 0x1df80, 0
   /usr/bin/gedit, 0x1dfc0, 0
   /usr/bin/gedit, 0xe0000, 0
   /usr/bin/gedit, 0xe0040, 0
   /usr/bin/gedit, 0xe0080, 0
   /usr/bin/gedit, 0xe00c0, 0
   /usr/bin/gedit, 0xe0100, 0
   /usr/bin/gedit, 0xe0140, 0
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event

![Image of profiling output and graphical user interface]

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
Profiling a Single Event
Profiling a Single Event

![Screenshot of a text editor showing a list of file paths and a terminal window with text output.](image-url)
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling a Single Event
Profiling Phase: 1 Event, 1 Address

ADDRESS

0x7c800

KEY

n
Profiling Phase: 1 Event, 1 Address

Example: Cache Hit Ratio for \((0x7c800, n)\): 200 / 200
Profiling Phase: All Events, 1 Address

ADDRESS

0x7c800

KEY
g h i j k l m n o p q r s t u v w x y z
Profiling Phase: All Events, 1 Address

Example: Cache Hit Ratio for \((0x7c800, u)\): 13 / 200
Profiling Phase: All Events, 1 Address

Distinguish \texttt{n} from other keys by monitoring \texttt{0x7c800}
Profiling Phase: All Events, All Addresses

![Key diagram]

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
Exploitation Phase

- Monitor addresses from Cache Template
Exploitation Phase

- Monitor addresses from Cache Template
- Report to log file / attacker
Exploitation Phase

- Monitor addresses from Cache Template
- Report to log file / attacker
- Manual analysis of log file
  - Find password in keypress log, etc.

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
Example Attack: Keylogging

- Linux with GTK: monitor keystrokes of specific keys
- Detect groups of keys
- Some keys distinct
Example Attack: AES T-Tables

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Events: encryption with only one fixed key byte
Example Attack: AES T-Tables

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Events: encryption with only one fixed key byte
- Profile each event
Example Attack: AES T-Tables

AES T-Table implementation from OpenSSL 1.0.2

- Known-plaintext attack
- Events: encryption with only one fixed key byte
- Profile each event
- Exploitation phase: an auto-generated first-round attack
Example Attack: AES T-Tables Template

![ADDRESS](transposed)

\[
k_0 = 0x00
\]

\[
k_0 = 0x55
\]
Cache Template Attacks: How hard is it?

- We let students attack different programs
- Leakage typically found within 30 minutes
- Wide range of programs
Cache Template Attacks: Example Applications

- Keystrokes in nano, quasselclient, xterm, Pidgin, gksu, Firefox, GNU Emacs (on OSX), mousepad, user32.dll on Windows
- Keystrokes specifically in the password field in KeePassX
- Font rendering in libharfbuzz (used by many Linux applications)
- Action buttons in VLC (play, mute, ...)
- Receiving characters in netcat
- Distinguishing keygroups in gksu
- Monitoring checkbox clicking in gksu
Cache Template Attacks: Conclusion

- Technique to find any cache side-channel leakage
Cache Template Attacks: Conclusion

- Technique to find any cache side-channel leakage
- Works on Intel, ARM, and some AMD
- Works even with unknown binaries
Cache Template Attacks: Conclusion

- Technique to find any cache side-channel leakage
- Works on Intel, ARM, and some AMD
- Works even with unknown binaries
- Large scale automated cache attacks possible!
Cache Template Attacks: Conclusion

- Technique to find any cache side-channel leakage
- Works on Intel, ARM, and some AMD
- Works even with unknown binaries
- Large scale automated cache attacks possible!
- But: still requires shared memory
What about Prime+Probe?

- Good: no shared memory
What about Prime+Probe?

- Good: no shared memory
- Bad: much more difficult due to noise
What about Prime+Probe?

- Good: no shared memory
- Bad: much more difficult due to noise

→ Let’s do something more fun with Prime+Probe!
1. Caches

2. Cache attacks

3. Cache Template Attacks

4. Rowhammer
DRAM organization example
DRAM organization example

channel 0

channel 1
DRAM organization example

back of DIMM: rank 1

front of DIMM: rank 0

channel 0

channel 1
DRAM organization example

back of DIMM: rank 1

front of DIMM: rank 0

chip

channel 0

channel 1
DRAM organization example

- bits in cells in rows
- access: activate row, copy to row buffer
- cells leak $\rightarrow$ refresh necessary
- cells leak faster upon proximate accesses

chip

<table>
<thead>
<tr>
<th>bank 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>row 0</td>
</tr>
<tr>
<td>row 1</td>
</tr>
<tr>
<td>row 2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>row 32767</td>
</tr>
<tr>
<td>row buffer</td>
</tr>
</tbody>
</table>
Rowhammer

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice
Rowhammer

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice

![Diagram of DRAM bank and row buffer showing the activation and copying process.](image-url)
Rowhammer

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice
Rowhammer

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice
Rowhammer

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice
Rowhammer

“It’s like breaking into an apartment by repeatedly slamming a neighbor’s door until the vibrations open the door you were after” – Motherboard Vice

<table>
<thead>
<tr>
<th>DRAM bank</th>
<th>bit flips in row 2!</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>1 0 1 1 1 1 1 0 1 0 1 1 1</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1 1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1 1 1 1</td>
<td></td>
</tr>
</tbody>
</table>

row buffer
Impact of the CPU cache

- only **non-cached accesses** reach DRAM
- original attacks use `clflush` instruction
  → flush line from cache
  → next access will be served from DRAM
Rowhammer

- Rowhammer: DRAM bug that causes bit flips [Kim et al., 2014]
- Bug used in security exploits [Seaborn, 2015]
- Need non-cached accesses to reach DRAM
- Very similar to Flush+Reload
Rowhammer (with clflush)

- Cache set 1
- Cache set 2
- DRAM bank
Rowhammer (with clflush)

cache set 1

cache set 2

DRAM bank

clflush

clflush
Rowhammer (with clflush)
Rowhammer (with clflush)
Rowhammer (with `clflush`)

- Cache set 1
- Cache set 2
- DRAM bank

Reload
Rowhammer (with clflush)

DRAM bank

cache set 1

cache set 2

reload

reload

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
Rowhammer (with clflush)

- cache set 1
- cache set 2
- DRAM bank

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
Rowhammer (with clflush)
Rowhammer (with clflush)

cache set 1

cache set 2

DRAM bank

clflush

clflush
Rowhammer (with clflush)

cache set 1

cache set 2

DRAM bank

reload

reload
Rowhammer (with clflush)

cache set 1

clflush

cache set 2

clflush

DRAM bank
Rowhammer (with clflush)

- DRAM bank
- cache set 1
  - Reload
- cache set 2
  - Reload
Rowhammer (with clflush)

wait for it...
Rowhammer (with clflush)

cache set 1

DRAM bank

cache set 2

reload

bit flip!
Rowhammer without clflush?

Idea: use Prime+Probe for Rowhammer instead of Flush+Reload
Rowhammer without clflush

cache set 1

DRAM bank

 cache set 2
Rowhammer without \texttt{clflush}


cache set 1

\begin{center}
\begin{tikzpicture}
\node at (0,0) {\text{load}};
\draw[fill=gray] (0,0) rectangle (1,1);
\draw[fill=green] (1,0) rectangle (2,1);
\node at (2,0) {\text{load}};
\end{tikzpicture}
\end{center}

cache set 2

\begin{center}
\begin{tikzpicture}
\node at (0,0) {\text{load}};
\draw[fill=green] (0,0) rectangle (1,1);
\draw[fill=gray] (1,0) rectangle (2,1);
\node at (2,0) {\text{load}};
\end{tikzpicture}
\end{center}

\textbf{DRAM bank}
Rowhammer without clflush

![Diagram showing cache sets and DRAM bank](image-url)
Rowhammer without `clflush`

cache set 1

cache set 2

DRAM bank
Rowhammer without clflush

Cache set 1

Cache set 2

DRAM bank

load

load
Rowhammer without clflush

- cache set 1
- cache set 2
- load

DRAM bank
Rowhammer without `clflush`

(cache set 1)  
(cache set 2)  

DRAM bank
Rowhammer without clflush

cache set 1

DRAM bank

load

cache set 2

load
Rowhammer without clflush

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
Rowhammer without `clflush`

Cache set 1

Cache set 2

DRAM bank

reload

reload
Rowhammer without clflush

repeat!

cache set 1

cache set 2

DRAM bank
Rowhammer without clflush

cache set 1

cache set 2

DRAM bank

wait for it...
Rowhammer without clflush

cache set 1

cache set 2

DRAM bank

bit flip!
Rowhammer without clflush

Challenges:

1. How to get accurate timing (in JS)?
2. How to get physical addresses (in JS)?
3. Which physical addresses to access?
4. In which order to access them?
Rowhammer without clflush

Challenges:

1. How to get accurate timing (in JS)? → easy
2. How to get physical addresses (in JS)? → easy
3. Which physical addresses to access? → already solved
4. In which order to access them? → details today!
Accurate timing in JS (easy)

- native code: `rdtsc`
- JavaScript: `window.performance.now()`
Physical addresses in JS (easy)

- fixed map between physical addresses and DRAM
- we reverse-engineered it [Pessl et al., 2015]
- OS optimization: use 2MB pages
- last 21 bits equal for physical address, virtual address, JS array index [Gruss et al., 2015a]
- several DRAM rows per 2MB page
Which addresses to access? (already solved)

- use eviction → congruent addresses
- slice mapping is known [Maurice et al., 2015a]
Which addresses to access? (already solved)

- use eviction → congruent addresses
- slice mapping is known [Maurice et al., 2015a]

Remaining challenge: “in which order to access them?”
Replacement policy on older CPUs

“LRU eviction” memory accesses

cache set

[Diagram showing a cache set with one entry highlighted]
Replacement policy on older CPUs

“LRU eviction” memory accesses

- LRU replacement policy: oldest entry first
Replacement policy on older CPUs

“LRU eviction” memory accesses

- LRU replacement policy: oldest entry first
- timestamps for every cache line

<table>
<thead>
<tr>
<th>cache set</th>
<th>2</th>
<th>5</th>
<th>8</th>
<th>1</th>
<th>7</th>
<th>6</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
</table>
Replacement policy on older CPUs

“LRU eviction” memory accesses

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp
Replacement policy on older CPUs

“LRU eviction” memory accesses

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp
Replacement policy on older CPUs

“LRU eviction” memory accesses

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp
Replacement policy on older CPUs

“LRU eviction” memory accesses

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp
Replacement policy on older CPUs

“LRU eviction” memory accesses

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp

Daniel Gruss, IAIK, Graz University of Technology
April 28, 2016
Replacement policy on older CPUs

“LRU eviction” memory accesses

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp
Replacement policy on older CPUs

“LRU eviction” memory accesses

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp
Replacement policy on older CPUs

“LRU eviction” memory accesses

- LRU replacement policy: oldest entry first
- timestamps for every cache line
- access updates timestamp
Replacement policy on recent CPUs

“LRU eviction” memory accesses

- no LRU replacement

```plaintext
<table>
<thead>
<tr>
<th>Cache Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 5 8 1 7 6 3 4</td>
</tr>
</tbody>
</table>
```
Replacement policy on recent CPUs

“LRU eviction” memory accesses

- no LRU replacement

![Diagram of cache set with numbers 2, 5, 8, 9, 7, 6, 3, 4]
Replacement policy on recent CPUs

“LRU eviction” memory accesses

- no LRU replacement
Replacement policy on recent CPUs

“LRU eviction” memory accesses

- no LRU replacement
Replacement policy on recent CPUs

“LRU eviction” memory accesses

- no LRU replacement
Replacement policy on recent CPUs

“LRU eviction” memory accesses

- no LRU replacement
Replacement policy on recent CPUs

“LRU eviction” memory accesses

- no LRU replacement
Replacement policy on recent CPUs

“LRU eviction” memory accesses

- no LRU replacement
Replacement policy on recent CPUs

“LRU eviction” memory accesses

- no LRU replacement
Replacement policy on recent CPUs

“LRU eviction” memory accesses

- no LRU replacement
- only 75% success rate on Haswell
Replacement policy on recent CPUs

“LRU eviction” memory accesses

- no LRU replacement
- only 75% success rate on Haswell
- more accesses → higher success rate, but too slow
Cache eviction strategy (the beginning)
Cache eviction strategy (new representation)

Represent accesses as a sequence of numbers:

- 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4, ...
- can be a long sequence
Cache eviction strategy properties

- all addresses are congruent
  \[ \rightarrow \text{indistinguishable in terms of the eviction strategy} \]
Cache eviction strategy properties

- all addresses are congruent
  → indistinguishable in terms of the eviction strategy
    - 2, 5, 4, 6, 3, 1, 2, 5, 6, 4, 1, 3, ... all equivalent!
  → 1, 2, 3, 4, 5, 6
Cache eviction strategy properties

- all addresses are congruent
  → indistinguishable in terms of the eviction strategy
    - 2, 5, 4, 6, 3, 1, 2, 5, 6, 4, 1, 3, ... all equivalent!
    → 1, 2, 3, 4, 5, 6

- adding more **different** addresses can increase eviction rate
- **multiple** accesses to one address can increase the eviction rate
Cache eviction strategy properties (2)

- multiple accesses, low number of different addresses
  → sequences often have repeating sub-sequences
Cache eviction strategy properties (2)

- multiple accesses, low number of different addresses
  → sequences often have repeating sub-sequences
- idea: repetitive sequences can be compressed to loops over sub-sequences
Cache eviction strategy (compression)

for (s = 0; s <= S-D; s += L)
    for (c = 0; c <= C; c += 1)
        for (d = 0; d <= D; d += 1)
            *a[s+d] ;
Cache eviction strategy (compression)

```
for (s = 0; s <= S-D; s += L)
    for (c = 0; c <= C; c += 1)
        for (d = 0; d <= D; d += 1)
            *a[s+d];
```

*D*: different addresses per inner access loop
*C*: number of repetitions of the inner access loop
*S*: total number of different addresses (=set size)
*L*: overlap parameter for the inner access loops (=step size)
Cache eviction strategy (compression)

```
for (s = 0; s <= S-D; s += L)
  for (c = 0; c <= C; c += 1)
    for (d = 0; d <= D; d += 1)
      *a[s+d];
```

- **D**: different addresses per inner access loop
- **C**: number of repetitions of the inner access loop
- **S**: total number of different addresses (=set size)
- **L**: overlap parameter for the inner access loops (=step size)

Write eviction strategies as: \( \mathcal{P}-C-D-L-S \)
Cache eviction strategy example

```c
for (s = 0; s <= S-D; s += L)
  for (c = 0; c < C; c += 1)
    for (d = 0; d < D; d += 1)
      *a[s+d];
```
Cache eviction strategy example

for (s = 0; s <= S-D; s += L)
    for (c = 0; c < C; c += 1)
        for (d = 0; d < D; d += 1)
            *a[s+d];

- $\mathcal{P}$-1-1-1-4
- $\mathcal{P}$-2-2-1-4
Cache eviction strategy example

\[
\begin{align*}
&\text{for (s = 0; s <= S-D; s += L)} \\
&\quad \text{for (c = 0; c < C; c += 1)} \\
&\quad \quad \text{for (d = 0; d < D; d += 1)} \\
&\quad \quad \quad \star a[s+d];
\end{align*}
\]

- $P$-1-1-1-4 $\rightarrow$ 1, 2, 3, 4
- $P$-2-2-1-4 $\rightarrow$ 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4
Cache eviction strategy example

for (s = 0; s <= S-D; s += L)
    for (c = 0; c < C; c += 1)
        for (d = 0; d < D; d += 1)
            *a[s+d];

- $P_{1-1-1-4} \rightarrow 1, 2, 3, 4$
- $P_{2-2-1-4} \rightarrow 1, 2, 1, 2, 2, 3, 2, 3, 3, 4, 3, 4$

We evaluated more than 10000 strategies...
Cache eviction strategies (simple example)

Which one has the better eviction rate?

- \( P-1-1-1-17 \) (as used in previous work)
- \( P-1-1-1-20 \)

(executed in a loop, on Haswell, 16-way cache)
Cache eviction strategies (simple example)

Which one has the better eviction rate?

- $P-1-1-1-17$ (as used in previous work)
  - Average eviction rate: 74.46%
  - Not high enough for Rowhammer!

- $P-1-1-1-20$
  - Average eviction rate: 99.82%
  - High enough for Rowhammer!

(executed in a loop, on Haswell, 16-way cache)
Cache eviction strategies (simple example)

Which one is faster?

- \( \mathcal{P}-1-1-1-17 \) (as used in previous work)

- \( \mathcal{P}-1-1-1-20 \) (=20 accesses)

(exeucted in a loop, on Haswell, 16-way cache)
Cache eviction strategies (simple example)

Which one is faster?

- \( P-1-1-1-17 \) (as used in previous work)
  - Average execution time: 307 ns
  - Fast enough for Rowhammer!

- \( P-1-1-1-20 \) (=20 accesses)
  - Average execution time: 934 ns
  - Not fast enough for Rowhammer!

(executed in a loop, on Haswell, 16-way cache)
Cache eviction strategies (another example)

Which one has the better eviction rate?

- \( P-1-1-1-17 \) (=17 accesses)
  
  \[ \rightarrow \text{Average eviction rate: 74.46\%} \]

- \( P-2-1-1-17 \) (=34 accesses)

(executed in a loop, on Haswell, 16-way cache)
Cache eviction strategies (another example)

Which one has the better eviction rate?

- $\mathcal{P}-1-1-1-17$ (=17 accesses)
  - Average eviction rate: 74.46%
- $\mathcal{P}-2-1-1-17$ (=34 accesses)
  - Average eviction rate: 99.86%
  - High enough for Rowhammer!

(executed in a loop, on Haswell, 16-way cache)
Cache eviction strategies (another example)

Which one is faster?

- $P_{1-1-1-17}$ (=17 accesses)
  - $\rightarrow$ Average execution time: 307 ns
- $P_{2-1-1-17}$ (=34 accesses)

(executed in a loop, on Haswell, 16-way cache)
Cache eviction strategies (another example)

Which one is faster?

- \( P-1-1-1-17 \) (=17 accesses)
  - Average execution time: 307 ns

- \( P-2-1-1-17 \) (=34 accesses)
  - Average execution time: 191 ns
  - Fast enough for Rowhammer!

(executed in a loop, on Haswell, 16-way cache)
Cache eviction strategies (another example)

Which one has the better eviction rate?

- $\mathcal{P}-2-1-1-17$ (=34 accesses)
  - Average eviction rate: 99.86%
  - Average execution time: 191 ns

- $\mathcal{P}-2-2-1-17$ (=64 accesses)

(executed in a loop, on Haswell, 16-way cache)
Cache eviction strategies (another example)

Which one has the better eviction rate?

- $P-2-1-1-17$ (=34 accesses)
  - Average eviction rate: 99.86%
  - Average execution time: 191 ns

- $P-2-2-1-17$ (=64 accesses)
  - Average eviction rate: 99.98%
  - Average execution time: 180 ns

(executed in a loop, on Haswell, 16-way cache)
Rowhammer without clflush

Challenges:

1. How to get accurate timing (in JS)? → easy
2. How to get physical addresses (in JS)? → easy
3. Which physical addresses to access? → solved
4. In which order to access them?
Rowhammer without clflush

Challenges:

1. How to get accurate timing (in JS)? → easy
2. How to get physical addresses (in JS)? → easy
3. Which physical addresses to access? → solved
4. In which order to access them? → solved
Evaluation on Haswell

![Graph showing the number of bit flips within 15 minutes.](image)

**Figure:** Number of bit flips within 15 minutes. [Gruss et al., 2016]
Conclusions

- cache eviction good enough to replace clflush
- independent of programming language and available instructions
- hardware-fault attack induced in JavaScript
  → remote attacks through websites are possible
At least on this laptop (default settings)

ROOT privileges for web apps!
Cache Side-Channel Attacks and the case of Rowhammer

Daniel Gruss
IAIK, Graz University of Technology
April 28, 2016
Practical Memory Deduplication Attacks in Sandboxed JavaScript.

Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript.
In DIMVA’16.

Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches.
In USENIX Security Symposium (USENIX Security’15).

Cache Games – Bringing Access-Based Cache Attacks on AES to Practice.
In IEEE Symposium on Security and Privacy (S&P’11).
Practical Timing Side Channel Attacks against Kernel Space ASLR.

Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud.
*Cryptology ePrint Archive, Report 2015/898*.

Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors.
In *ACM/IEEE International Symposium on Computer Architecture (ISCA’14)*.

Last-Level Cache Side-Channel Attacks are Practical.
In *IEEE Symposium on Security and Privacy (S&P’15)*.
Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters.
In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

C5: Cross-Cores Cache Covert Channel.
In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA’15)*.

Cache Missing for Fun and Profit.
URL: [http://daemonology.net/hyperthreading-considered-harmful/](http://daemonology.net/hyperthreading-considered-harmful/).

Reverse Engineering Intel DRAM Addressing and Exploitation.
*arXiv:1511.08756.*
Exploiting the DRAM rowhammer bug to gain kernel privileges.

In USENIX Security Symposium (USENIX Security’14).

Mapping the Intel Last-Level Cache.