Security: Can we afford to have it? Can we afford not to have it?

Daniel Gruss
2023-10-09
Graz University of Technology
side channel
= obtaining meta-data and deriving secrets from it

CHANGE MY MIND
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
Profiling cache utilization with performance counters?
- Profiling cache utilization with performance counters? → No
• Profiling cache utilization with performance counters? → No
• Observing cache utilization with performance counters and using it to infer a crypto key?
Profiling cache utilization with performance counters? → No

Observing cache utilization with performance counters and using it to infer a crypto key? → Yes
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?
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
THERE IS NO NOISE

NOISE IS JUST SOMEONE ELSE'S DATA
Revolutionary concept!
Store your food at home, never go to the grocery store during cooking.
Can store ALL kinds of food.

ONLY TODAY INSTEAD OF $1,300
ORDER VIA PHONE: +555 12345
printf("%d", i);
printf("%d", i);
printf("%d", i);
Cache miss
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);

CPU Cache

Request

Response

Cache miss

Daniel Gruss — Graz University of Technology
printf("%d", i);
printf("%d", i);
printf("%d", i);
Cache miss

printf("%d", i);
Cache hit
CPU Cache

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

DRAM access, slow

Daniel Gruss — Graz University of Technology
CPU Cache

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

- **Cache miss**
  - DRAM access, slow

- **Cache hit**
  - No DRAM access, much faster

- **Request**
- **Response**
Flush + Reload

ATTACKER

Shared Memory

VICTIM

flush
access
access
Flush+Reload

Shared Memory

ATTACKER

flush
access

cached

VICTIM

cached
access

Daniel Gruss — Graz University of Technology
Flush+Reload

ATTACKER

\texttt{flush}

\texttt{access}

Shared Memory

\texttt{access}

VICTIM

Daniel Gruss — Graz University of Technology
Flush+Reload

ATTACKER

flush
access

Shared Memory

VICTIM

access
Flush+Reload

ATTACKER
flush
access

Shared Memory

VICTIM
access
Flush+Reload

ATTACKER

flush
access

Shared Memory

VICTIM

access

Daniel Gruss — Graz University of Technology
Flush+Reload

ATTACKER

flush

access

fast if victim accessed data,
slow otherwise

Shared Memory

VICTIM

access

Daniel Gruss — Graz University of Technology
Memory Access Latency

Access time [CPU cycles]

Number of accesses

Cache Hits
Cache Template

Address

Key

0x7c680
0x7c6c0
0x7c700
0x7c740
0x7c780
0x7c7c0
0x7c800
0x7c840
0x7c880
0x7c900
0x7c940
0x7c980
0x7cc0
0x7cb80
0x7cc40
0x7cc80
0x7ccc0
0x7cd00
• Add a layer of indirection to test

```c
char data = *(char*) 0xffffffff81a000e0;
array[data * 4096] = 0;
```
• Add a layer of indirection to test

```c
char data = *(char*) 0xffffffffffff81a000e0;
array[data * 4096] = 0;
```

• Then check whether any part of `array` is cached
• **Flush+Reload over all pages of the array**

![Graph showing access time per page]

• **Index of cache hit reveals data**

Daniel Gruss — Graz University of Technology
• Flush+Reload over all pages of the array

• Index of cache hit reveals data

• Permission check is in some cases not fast enough
Kernel Address Isolation to have Side channels Efficiently Removed
Kernel Address Isolation to have Side channels Efficiently Removed
Without KAISER:

Shared address space

User memory

Kernel memory

context switch

With KAISER:

User address space

Not mapped

Kernel address space

SMAP + SMEP

context switch
Without KAISER:

Shared address space

User memory ➔ Kernel memory

0 ➔ −1

context switch

With KAISER:

User address space

User memory ➔ Not mapped

SMAP + SMEP ➔ Kernel memory

0 ➔ −1

context switch

kernel address space

Interrupt dispatcher
A table for 6 please
Speculative Cooking
»A table for 6 please«
index = 0;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

else

0
LUT

index = 0;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

Prediction

0

else
index = 0;

char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
Speculate
else
0
```c
index = 0;
char* data = "textKEY";
if (index < 4)
  LUT[data[index] * 4096]
else
  0
```
index = 1;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096] 0
index = 1;

char* data = "textKEY";

if (index < 4)
    then
        Prediction
        LUT[data[index] * 4096]
    else
        0
index = 1;

char* data = "textKEY";

if (index < 4)
    LUT[data[index] * 4096]
else
    0
index = 1;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0
index = 2;

char* data = "textKEY";

if (index < 4) then

LUT[data[index] * 4096]

else

0
index = 2;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096] 0
```c
index = 2;
char* data = "textKEY";

if (index < 4)
    LUT[data[index] * 4096]
else
    0
```
index = 2;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

else

0
index = 3;

char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
0

Prediction
index = 3;

```c
char* data = "textKEY";
```

```
if (index < 4)
```

```
LUT[data[index] * 4096]
```

```
0
```

Prediction
index = 3;

c char* data = "textKEY";

if (index < 4)
    LUT[data[index] * 4096]
else
    0
index = 3;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0
index = 4;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

Prediction

0

Daniel Gruss — Graz University of Technology
index = 4;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

else

0

Prediction
index = 4;

char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
0
index = 4;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

Prediction

Execute

0
index = 5;

char* data = "textKEY";

if (index < 4)
    LUT[data[index] * 4096]
else
    0
index = 5;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] \times 4096]

else

Prediction

0
index = 5;

char* data = "textKEY";

if (index < 4)
then
Speculate
LUT[data[index] * 4096]
else
Prediction
0
index = 5;

char* data = "textKEY";

if (index < 4)
    then
        LUT[data[index] * 4096]
    else
        Prediction

Execute

0
index = 6;

char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
Prediction

0
index = 6;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0

Prediction
index = 6;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

else

0
index = 6;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

else

0
operation #n

prediction

operation #n+2

possibly architectural transient execution

flush pipeline on wrong prediction

daniel gruss — graz university of technology
Meltdown

operation #n

exception

raise

data

data dependency

possibly architectural

operation #n+2

transient execution

time

15
Mistraining Location

out-of-place/same-address-space

Congruent branch

Address collision

Victim branch

Shared Branch Prediction State

Attacker

Congruent branch

Address collision

Shadow branch

in-place/same-address-space

out-of-place/cross-address-space

in-place/cross-address-space

Victim

Victim branch
## Table 1: Reported performance impacts of countermeasures

<table>
<thead>
<tr>
<th>Defense</th>
<th>Impact</th>
<th>Performance Loss</th>
<th>Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>InvisiSpec</td>
<td>22%</td>
<td></td>
<td>SPEC</td>
</tr>
<tr>
<td>SafeSpec</td>
<td>3% (improvement)</td>
<td></td>
<td>SPEC2017 on MARSSx86</td>
</tr>
<tr>
<td>DAWG</td>
<td>2–12%, 1–15%</td>
<td></td>
<td>PARSEC, GAPBS</td>
</tr>
<tr>
<td>RSB Stuffing</td>
<td>no reports</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Retpoline</td>
<td>5–10%</td>
<td></td>
<td>real-world workload servers</td>
</tr>
<tr>
<td>Site Isolation</td>
<td>only memory overhead</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLH</td>
<td>36.4%, 29%</td>
<td></td>
<td>Google microbenchmark suite</td>
</tr>
<tr>
<td>YSNB</td>
<td>60%</td>
<td></td>
<td>Phoenix</td>
</tr>
<tr>
<td>IBRS</td>
<td>20–30%</td>
<td></td>
<td>two sysbench 1.0.11 benchmarks</td>
</tr>
<tr>
<td>STIPB</td>
<td>30–50%</td>
<td></td>
<td>Rodinia OpenMP, DaCapo</td>
</tr>
<tr>
<td>IBPB</td>
<td>no individual reports</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Serialization</td>
<td>62%, 74.8%</td>
<td></td>
<td>Google microbenchmark suite</td>
</tr>
<tr>
<td>SSBD/SSBB</td>
<td>2–8%</td>
<td></td>
<td>SYSmark\textsuperscript{®}2014 SE &amp; SPEC integer</td>
</tr>
<tr>
<td>KAISER/KPTI</td>
<td>0–2.6%</td>
<td></td>
<td>system call rates</td>
</tr>
<tr>
<td>L1TF mitigations</td>
<td>-3–31%</td>
<td></td>
<td>various SPEC</td>
</tr>
</tbody>
</table>
320: 12
330: 9
340: 1
350: 0
360: 1
370: 2
380: 199
390: 76
400: 72
410: 231
420: 572
1250

[!] Found flip (254 != 255) at array index 340021386 when hammering indices 339881984 and 340156416
[!] Found flip (239 != 255) at array index 340022176 when hammering indices 339881984 and 340156416
[!] Found flip (191 != 255) at array index 340023138 when hammering indices 339881984 and 340156416
[!] Found flip (254 != 255) at array index 340025146 when hammering indices 339881984 and 340156416
Rowhammer

Cells leak faster upon proximate accesses → Rowhammer

Daniel Gruss — Graz University of Technology
Rowhammer

Cells leak faster upon proximate accesses → Rowhammer

Daniel Gruss — Graz University of Technology
Cells leak faster upon proximate accesses → Rowhammer
Cells leak faster upon proximate accesses → Rowhammer
Cells leak faster upon proximate accesses → Rowhammer
Rowhammer

Cells leak faster upon proximate accesses → Rowhammer
#1 - Single-sided hammering

![Diagram of DRAM bank with activate signal]
#1 - Single-sided hammering

![Diagram of DRAM bank with activate signal]
#1 - Single-sided hammering

 activates DRAM bank
#1 - Single-sided hammering

DRAM bank

activate
#1 - Single-sided hammering

![Diagram of a DRAM bank with bits activating specific rows.](Diagram)
#1 - Single-sided hammering

DRAM bank

bit flips
#2 - Double-sided hammering

![DRAM bank diagram with binary values and activate flag highlighted]
#2 - Double-sided hammering

![DRAM bank diagram]

activate

20

Daniel Gruss — Graz University of Technology
#2 - Double-sided hammering

DRAM bank

- 1111111111111111
- 1111111111111111
- 1111111111111111
- 1111111111111111
- 1111111111111111
- 1111111111111111
- 1111111111111111
- 1111111111111111

activate
#2 - Double-sided hammering

![Diagram of DRAM bank with activate marker]

Daniel Gruss — Graz University of Technology
#2 - Double-sided hammering

DRAM bank

activate
#2 - Double-sided hammering

![Diagram of DRAM bank with bit flips and activation]

Daniel Gruss — Graz University of Technology
HAMMERING TWO ROWS
#3 - One-location hammering
#3 - One-location hammering

![Diagram of DRAM bank]
#3 - One-location hammering

![Diagram of DRAM bank with activate indicated]
#3 - One-location hammering

![DRAM bank diagram]
#3 - One-location hammering

![Diagram of a DRAM bank with bit flips indicated]
activate

Half-Double

DRAM bank

1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
Half-Double

DRAM bank

activate
Half-Double

DRAM bank

activate
Half-Double

activate

DRAM bank
Half-Double

DRAM bank

activate
Half-Double

DRAM bank

activate

bit flips
Opcode Flipping - Conditional Jump

JE
01110100

HLT
11110100

Daniel Gruss — Graz University of Technology
Opcode Flipping - Conditional Jump

JE

0 1 1 1 0 1 0 0

XORB

0 0 1 1 0 1 0 0

Daniel Gruss — Graz University of Technology
Opcode Flipping - Conditional Jump

JE

PUSHQ

0 1 1 1 1 0 1 0 0

0 1 0 1 0 1 0 1 0 0
Opcode Flipping - Conditional Jump

JE

01110100

<prefix>

011100100

Daniel Gruss — Graz University of Technology
Opcode Flipping - Conditional Jump

JE

\[
\begin{array}{cccccc}
0 & 1 & 1 & 1 & 0 & 1 & 0 & 0
\end{array}
\]

JO

\[
\begin{array}{cccccc}
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0
\end{array}
\]
Opcode Flipping - Conditional Jump

JE

0 1 1 1 0 1 0 0

↑

JBE

0 1 1 1 0 1 1 0
Opcode Flipping - Conditional Jump

JE

JNE

0 1 1 1 0 1 0 0

0 1 1 1 0 1 0 1

Daniel Gruss — Graz University of Technology
uint64_t multiplier = 0x1122334455667788;
uint64_t correct = 0xdeadbeef * multiplier;
uint64_t var = 0xdeadbeef * multiplier;

while (var == correct) {
    var = 0xdeadbeef * multiplier;
}

uint64_t flipped_bits = var ^ correct;
Safe voltages

- Base voltage
- Voltage for first fault

Voltage (V) vs. Frequency (GHz)
do
{
    i++;
    plaintext = <randomly generated>

    result1 = aes128_enc(plaintext);
    result2 = aes128_enc(plaintext);
} while (vec_equal_128(result1,result2) && i<iterations);
• Should be related to undervolting
• Should be related to *undervolting*
• From protected TEE *vaults*
• Should be related to undervolting
• From protected TEE vaults
• Steal
• Should be related to undervolting
• From protected TEE vaults
• Steal, corrupt
• Should be related to **undervolting**
• From protected TEE **vaults**
• Steal, corrupt, **plunder**, ...
Security Costs
Security Costs
Security Costs

- 8.20%
- 5%
0.09%
0.40%
There are alternatives
There are alternatives to security!
How expensive is security?
RACE FOR A VACCINE

WE WANT TO BE FIRST
WE WANT TO BE FIRST!

WE WANT TO BE FIRST!
WE WANT TO BE FIRST!

RACE TO ACT ON CLIMATE

YOU GO FIRST!
Why should we be first?

NO, YOU GO FIRST
They should go first!
FUNCTIONALITY

SECURITY
Moore's Law: The number of transistors on microchips has doubled every two years.

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing—such as processing speed or the price of computers.
THE END IS NEAR OF MOORE'S LAW
Before

CPU Undervolting

After

I7-9750H

Huge difference!!!
IF MY SYSTEMS RAN UNDERSIZED
TOTALLY FINE FOR 10 YEARS

WHY DO WE
WASTE 40% ENERGY?
Why are problems like Rowhammer not solved already?
Common misunderstandings...

- A "bit" more reliability
- Why not higher or dynamic refresh rates everywhere (e.g. TRR, PARA, ...)?
- "just a few more targeted refreshes"
- Why not ECC everywhere?

→ What incentives does it create?
Common misunderstandings...

• A “bit” more reliability

• Why not higher or dynamic refresh rates everywhere (e.g. TRR, PARA, ...)

• “just a few more targeted refreshes”

• Why not ECC everywhere?

→ What incentives does it create?
Common misunderstandings...

... create bad incentives.
Common misunderstandings...

... create bad incentives.

- A “bit” more reliability
Common misunderstandings...

... create bad incentives.

- A “bit” more reliability
- Why not higher or dynamic refresh rates everywhere (e.g. TRR, PARA, ...)?
... create bad incentives.

- A “bit” more reliability
- Why not higher or dynamic refresh rates everywhere (e.g. TRR, PARA, …)?
  - “just a few more targeted refreshes”
Common misunderstandings...

... create bad incentives.

- A “bit” more reliability
- Why not higher or dynamic refresh rates everywhere (e.g. TRR, PARA, ...)?
  - “just a few more targeted refreshes”
- Why not ECC everywhere?
Common misunderstandings...

... create bad incentives.

- A “bit” more reliability
- Why not higher or dynamic refresh rates everywhere (e.g. TRR, PARA, ...)?
  - “just a few more targeted refreshes”
- Why not ECC everywhere?

→ What incentives does it create?
Optimizing systems...

Fundamental problem: we assume what is still reliable

- Refreshing $x$ times per second is fine
- Normal usage, no adversary
- Assume there won’t be more than $n$ bit errors

$\Rightarrow$ How far can we go with $x$ while staying below $n$ bit errors?
Fundamental problem: we assume what is still reliable

- Refreshing $x$ times per second is fine
- Normal usage, no adversary
- Assume there won’t be more than $n$ bit errors
  → How far can we go with $x$ while staying below $n$ bit errors?
Fundamental problem: we assume what is still reliable
Fundamental problem: we assume what is still reliable

- Refreshing $x$ times per second is fine
Fundamental problem: we assume what is still reliable

- Refreshing $x$ times per second is fine
- Normal usage, no adversary
Fundamental problem: we assume what is still reliable

- Refreshing $x$ times per second is fine
- Normal usage, no adversary
- Assume there won’t be more than $n$ bit errors
Fundamental problem: we assume what is still reliable

- Refreshing $x$ times per second is fine
- Normal usage, no adversary
- Assume there won’t be more than $n$ bit errors

→ How far can we go with $x$ while staying below $n$ bit errors?
AFTER ALL THESE REFRESHES, WHY NOT

WHY SHOULDN'T I OPTIMIZE PERFORMANCE TO THE ABSOLUTE LIMIT?
Mobile vendors since 2018: let’s add ECC by default
Mobile vendors since 2018: let’s add ECC by default

- ECC memory $\rightarrow$ fewer bit flips + more security
Mobile vendors since 2018: let’s add ECC by default

- ECC memory $\rightarrow$ fewer bit flips + more security

Also vendors:
Real-world example

Mobile vendors since 2018: let’s add ECC by default
- ECC memory $\rightarrow$ fewer bit flips $+$ more security

Also vendors:
- Let’s squeeze out the last bit of efficiency for battery runtime until just before bit flips occur
It’s an optimization problem

• You never know how far is still safe
• “safe”/“reliable” changes over time
• Adversary is intelligent and improves attacks over time
It’s an optimization problem
It’s an optimization problem
It's an optimization problem

- You never know how far is still safe
It's an optimization problem

• You never know how far is still safe
• “safe” / “reliable” changes over time
It’s an optimization problem

- You never know how far is still safe
- “safe” / “reliable” changes over time
- Adversary is intelligent and improves attacks over time
Security vs Reliability
Security vs Reliability
YOU CAN'T DO THAT!
THAT'S AGAINST THE RULES!
Security for Efficiency?
Make bit flips degrade performance **without** impacting security
Make bit flips degrade performance \textit{without} impacting security

- Cryptographic MAC
Make bit flips degrade performance \textbf{without} impacting security

- Cryptographic MAC
- Detect \textit{any} number of bit flips
Make bit flips degrade performance \textbf{without} impacting security

- Cryptographic MAC
- Detect \textbf{any} number of bit flips
- Correction by \textbf{brute-force} search for correct data
<table>
<thead>
<tr>
<th># Errors</th>
<th># MAC Comp.</th>
<th>Avg Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17</td>
<td>11 ns</td>
</tr>
<tr>
<td>2</td>
<td>771</td>
<td>3.68 µs</td>
</tr>
<tr>
<td>3</td>
<td>33,800</td>
<td>124 µs</td>
</tr>
<tr>
<td>4</td>
<td>$1.51 \times 10^6$</td>
<td>6.65 ms</td>
</tr>
<tr>
<td>5</td>
<td>$6.91 \times 10^7$</td>
<td>261 ms</td>
</tr>
<tr>
<td>6</td>
<td>$3.07 \times 10^9$</td>
<td>12.8 s</td>
</tr>
<tr>
<td>7</td>
<td>$1.21 \times 10^{11}$</td>
<td>9.11 min</td>
</tr>
<tr>
<td>8</td>
<td>$5.72 \times 10^{12}$</td>
<td>6.11 h</td>
</tr>
</tbody>
</table>
CSI: Rowhammer Security Guarantees

- Silent data corruption less than once per $10^{9}$ billion years
- Second preimage after hammering for one year: $9.75 \times 10^{-5}$
- Erroneous correction of 8-bit errors: 0.0161 %
CSI:Rowhammer Security Guarantees

- Silent data corruption less than once per $10^{9}$ billion years
- Second preimage after hammering for one year: $9.75 \cdot 10^{-5}$
- Erroneous correction of 8-bit errors: 0.0161%
• Silent data corruption less than once per $10^9$ billion years
• Silent data corruption less than once per $10^9$ billion years
• Second preimage after hammering for one year: $9.75 \cdot 10^{-5} \%$
CSI: Rowhammer Security Guarantees

- Silent data corruption less than once per $10^9$ billion years
- Second preimage after hammering for one year: $9.75 \cdot 10^{-5} \%$
- Erroneous correction of 8-bit errors: $0.0161 \%$
Security: Can we afford to have it? Can we afford not to have it?

Daniel Gruss
2023-10-09

Graz University of Technology