The Story of Meltdown and Spectre

Jann Horn & Daniel Gruss

May 17, 2018

Who are we

- **Jann Horn**
- Google Project Zero
- ✉ jannh@google.com

Who are we

- **Daniel Gruss**
- Post-Doc @ Graz University Of Technology
- @lavados
- daniel.gruss@iaik.tugraz.at

1337 4242

FOOD CACHE

Revolutionary concept!

Store your food at home, never go to the grocery store during cooking.

Can store **ALL** kinds of food.

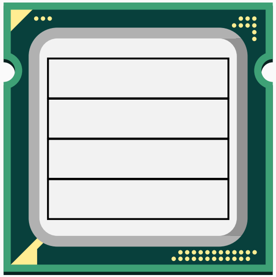
ONLY TODAY INSTEAD OF $~~1,300~~

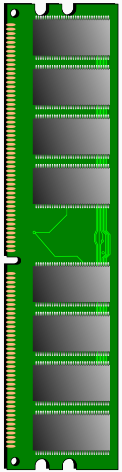
$1,299

ORDER VIA PHONE: +555 12345

CPU Cache

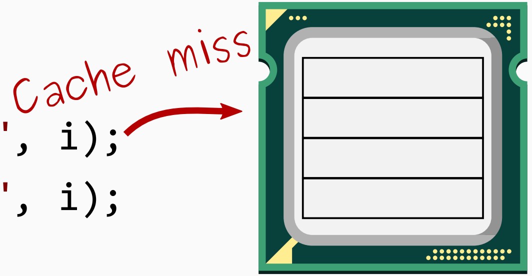
```
printf("%d", i);
printf("%d", i);
```

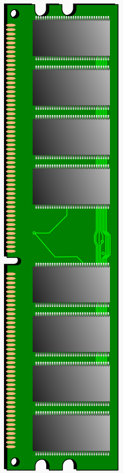




CPU Cache

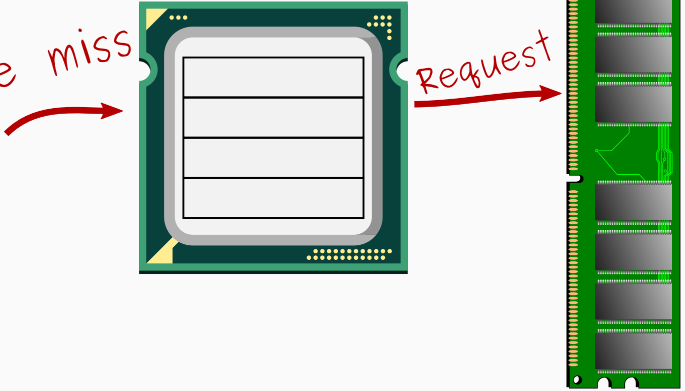
```
printf("%d", i);
printf("%d", i);
```





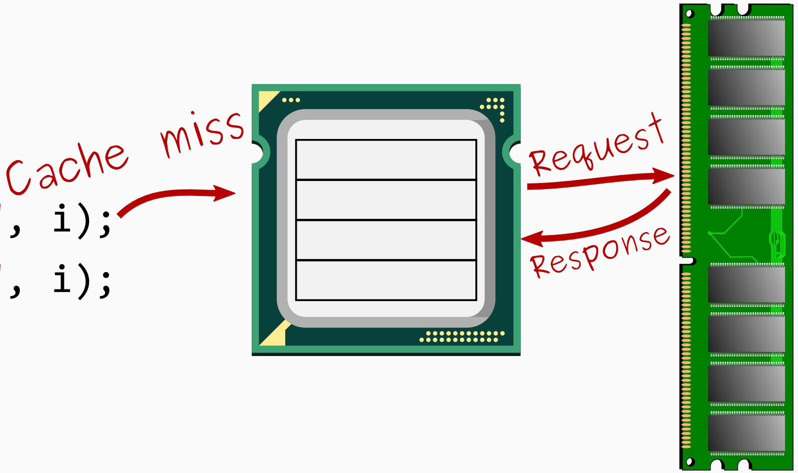
CPU Cache

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



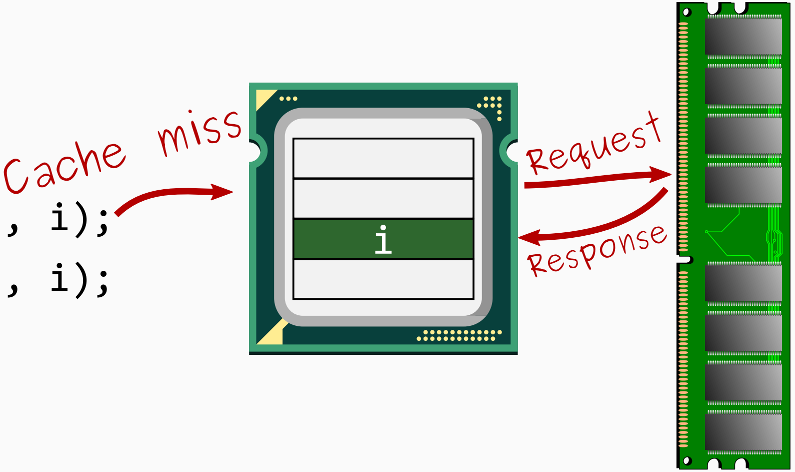
CPU Cache

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



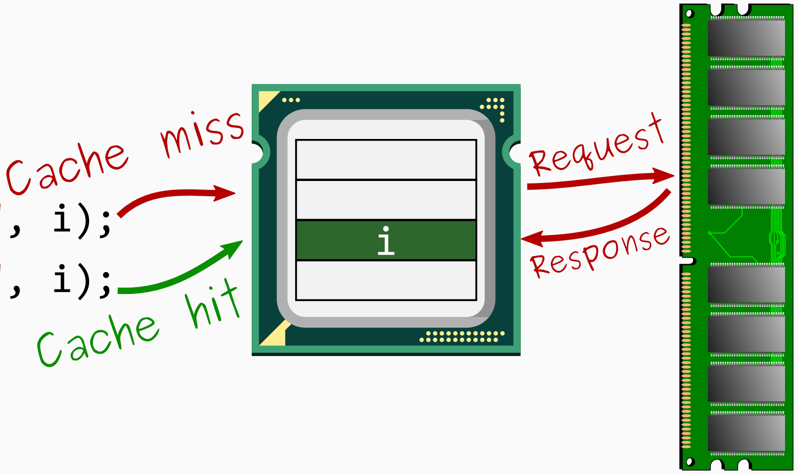
CPU Cache

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



CPU Cache

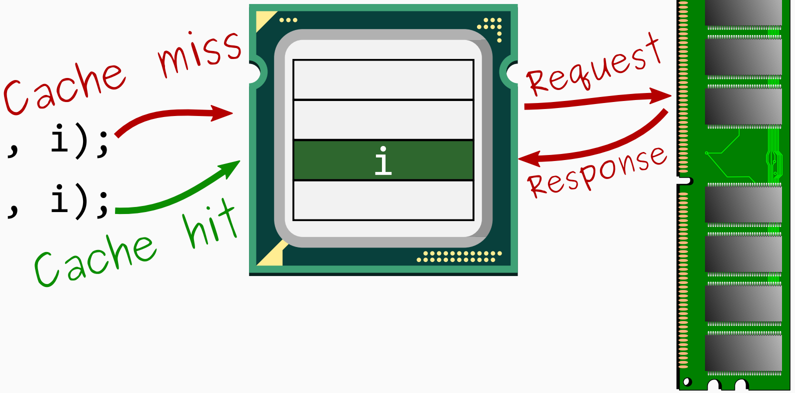
```
printf("%d", i);
printf("%d", i);
```



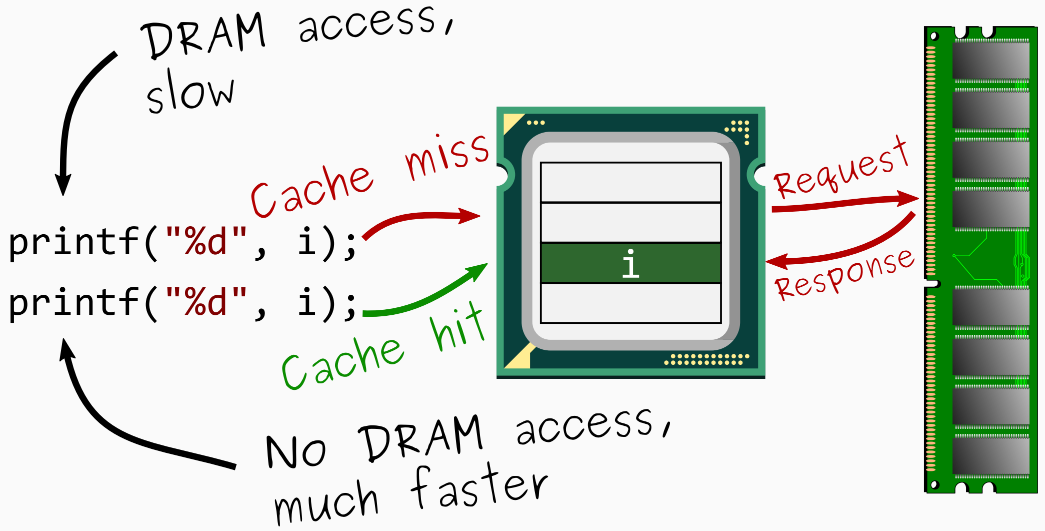
CPU Cache

DRAM access, slow

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



CPU Cache



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

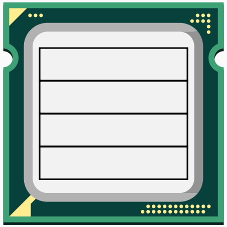
Flush+Reload

Shared Memory

ATTACKER

flush

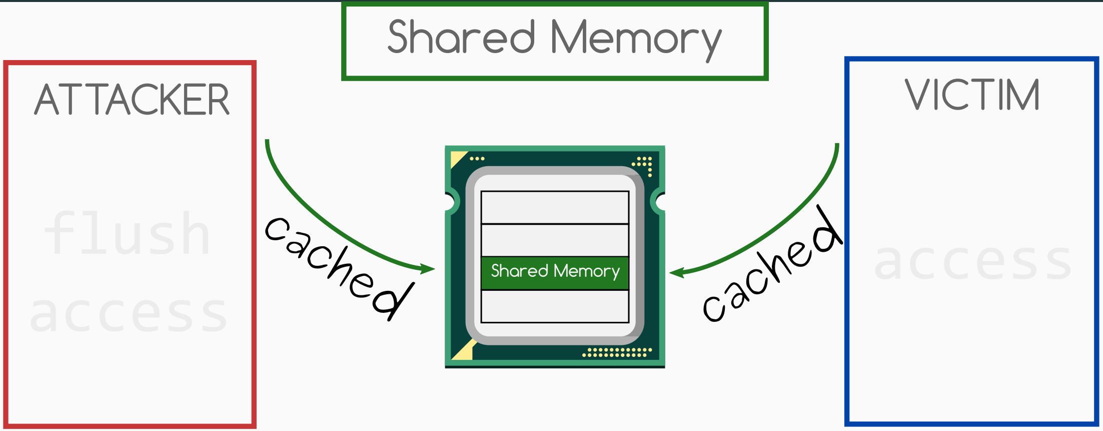
access



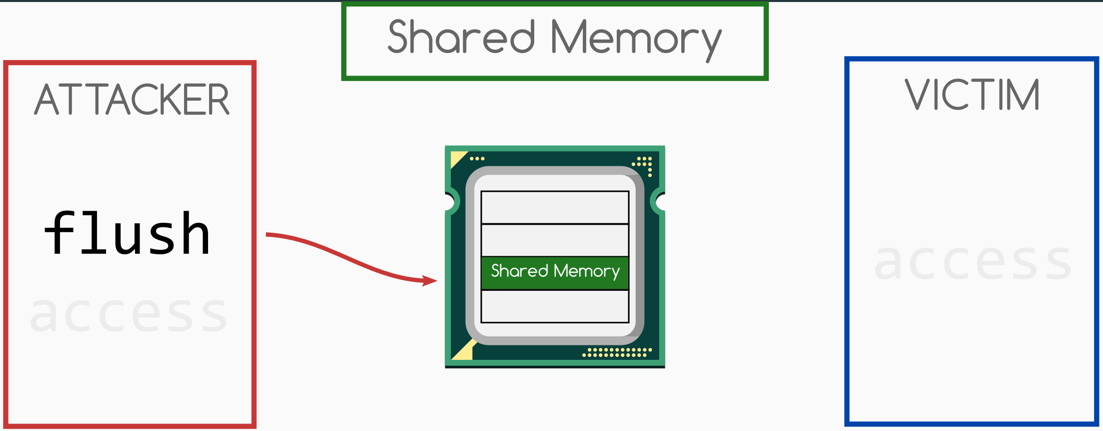
VICTIM

access

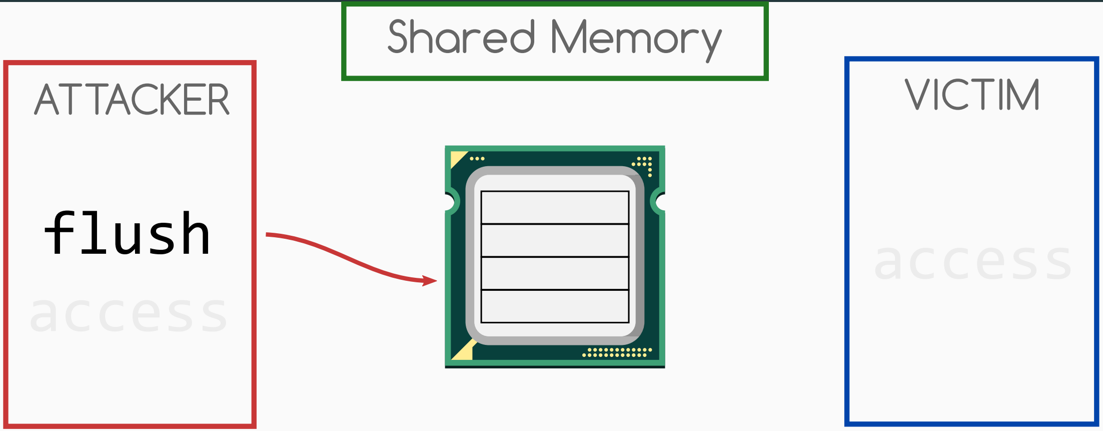
Flush+Reload



Flush+Reload



Flush+Reload



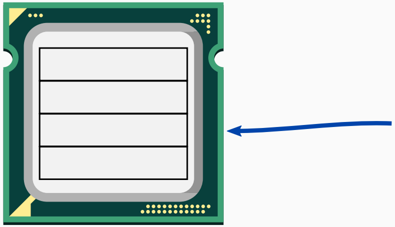
Flush+Reload

Shared Memory

ATTACKER

flush

access



VICTIM

access

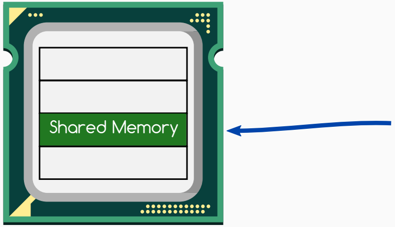
Flush+Reload

Shared Memory

ATTACKER

flush

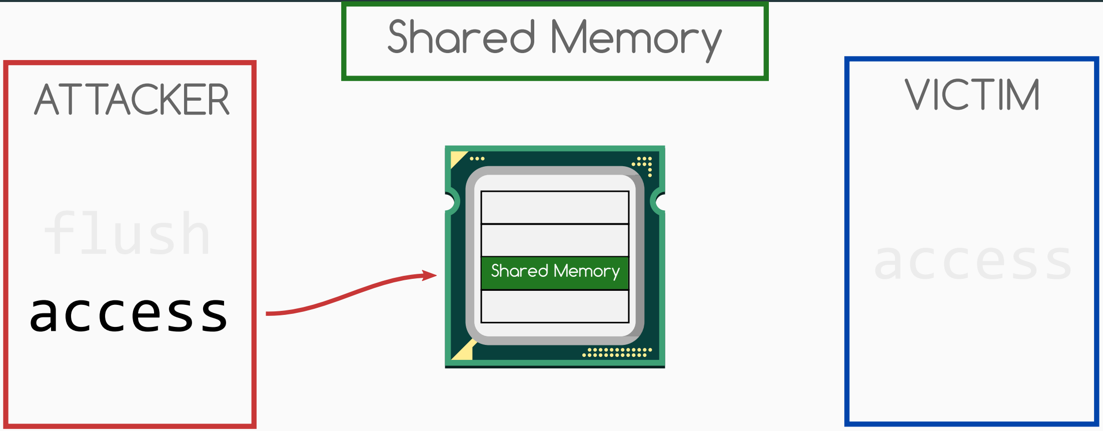
access



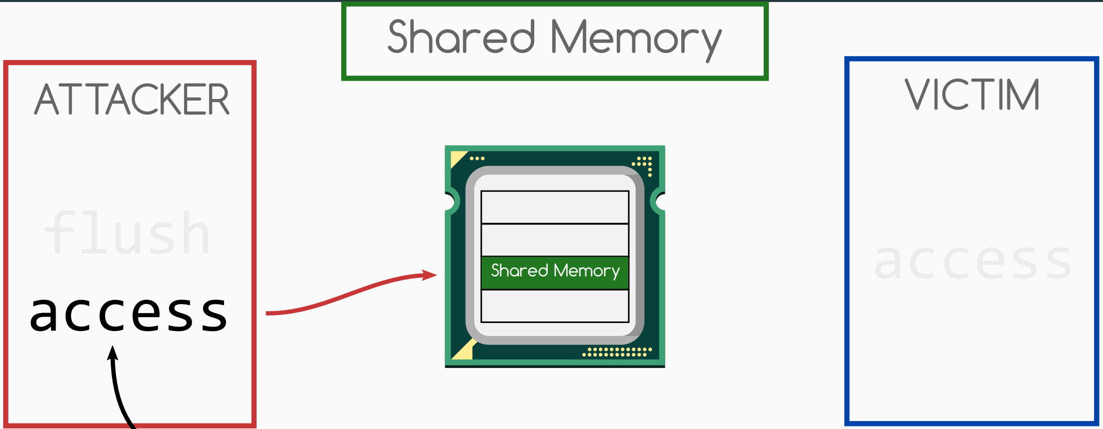
VICTIM

access

Flush+Reload

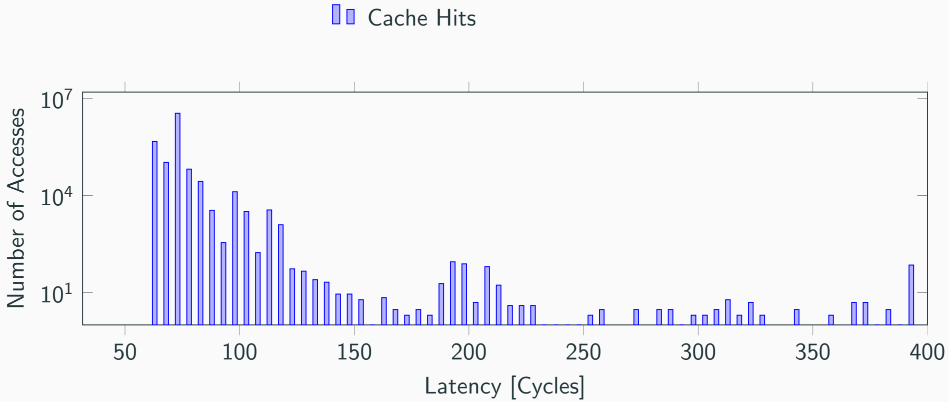


Flush+Reload

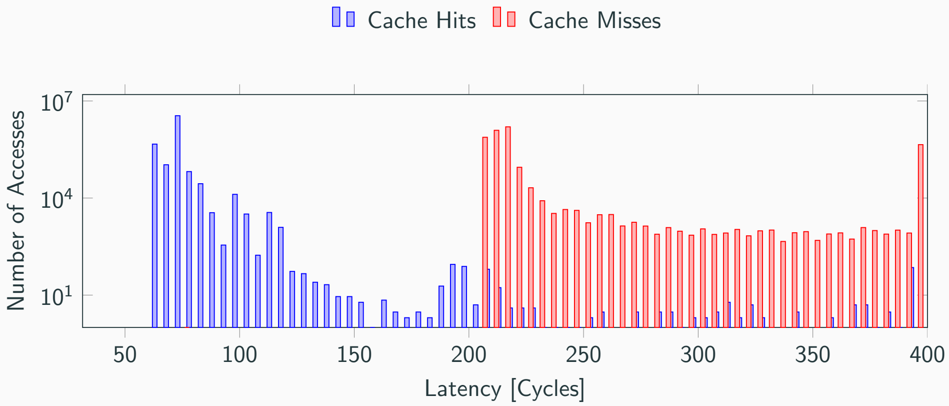


fast if victim accessed data,
slow otherwise

Memory Access Latency



Memory Access Latency



Out-of-order Execution

5. Cook everything until vegetables are soft

6. Add spices to taste [illegible] for 10 minutes

7. Serve with cooked and peeled potatoes

Wait for an hour

Wait for an hour

LATENCY

1. Wash and cut vegetables
2. Pick the basil leaves and set aside
3. Heat 2 tablespoons of oil in a pan
4. Fry vegetables until golden and softened

1. Wash and cut vegetables

2. Pick the basil leaves and set aside

3. Heat 2 tablespoons of oil in a pan

4. Fry vegetables until golden and softened

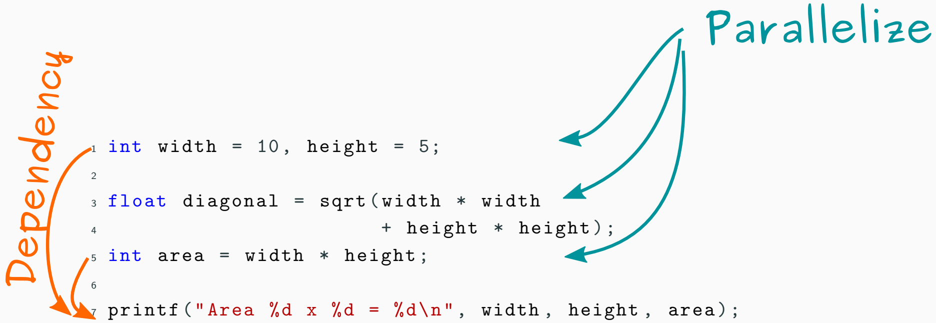
Parallelize

Dependency

Out-of-order Execution

```c
int width = 10, height = 5;

float diagonal = sqrt(width * width
                      + height * height);
int area = width * height;

printf("Area %d x %d = %d\n", width, height, area);
```

Out-of-order Execution



```c
int width = 10, height = 5;

float diagonal = sqrt(width * width
                      + height * height);
int area = width * height;

printf("Area %d x %d = %d\n", width, height, area);
```

Building Meltdown

```c
char data = *(char*)0xffffffff81a000e0;
printf("%c\n", data);
```

Building Meltdown

```c
char data = *(char*)0xffffffff81a000e0;
printf("%c\n", data);
```

```
segfault at ffffffff81a000e0 ip 0000000000400535
      sp 00007ffce4a80610 error 5 in reader
```

Building Meltdown

```c
char data = *(char*)0xffffffff81a000e0;
printf("%c\n", data);
```

```
segfault at ffffffff81a000e0 ip 0000000000400535
        sp 00007ffce4a80610 error 5 in reader
```

- Kernel addresses are not accessible

Building Meltdown

```c
char data = *(char*)0xffffffff81a000e0;
printf("%c\n", data);
```

```
segfault at ffffffff81a000e0 ip 0000000000400535
        sp 00007ffce4a80610 error 5 in reader
```

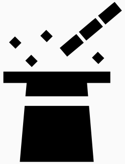
- Kernel addresses are not accessible
- Are privilege checks also done when executing instructions out of order?

Building Meltdown

- Adapted code

```
*(volatile char*)0;
array[84 * 4096] = 0; // unreachable
```

Building Meltdown



- Adapted code

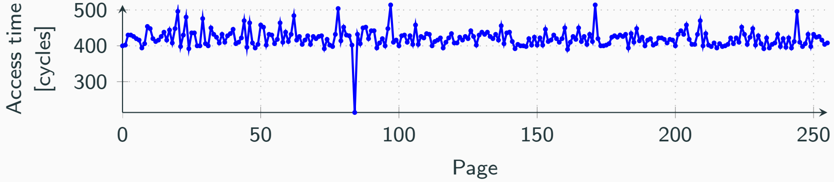
```
*(volatile char*)0;
array[84 * 4096] = 0; // unreachable
```

- Static code analyzer is not happy

```
warning: Dereference of null pointer
        *(volatile char*)0;
```

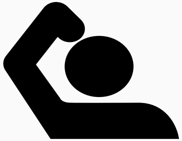
Building Meltdown

- Flush+Reload over all pages of the array

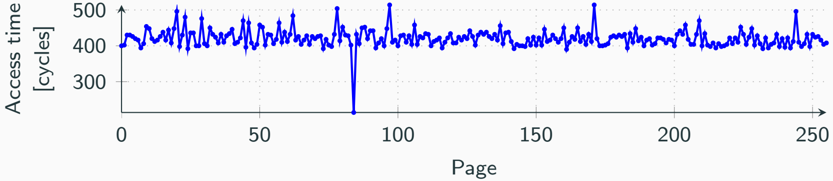


- "Unreachable" code line was actually executed

Building Meltdown



- Flush+Reload over all pages of the array



- "Unreachable" code line was actually executed
- Exception was only thrown afterwards

Building Meltdown

- Combine the two things

```
char data = *(char*)0xffffffff81a000e0;
array[data * 4096] = 0;
```

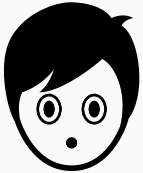
Building Meltdown

- Combine the two things

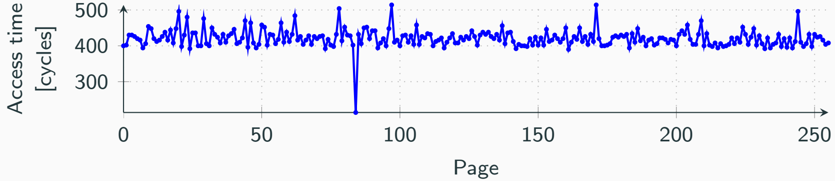
```
char data = *(char*)0xffffffff81a000e0;
array[data * 4096] = 0;
```

- Then check whether any part of array is cached

Building Meltdown

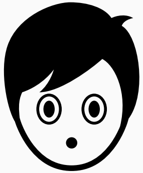


- Flush+Reload over all pages of the array

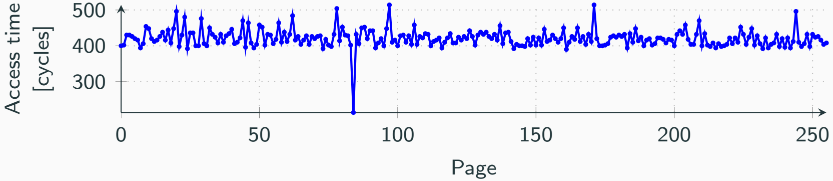


- Index of cache hit reveals data

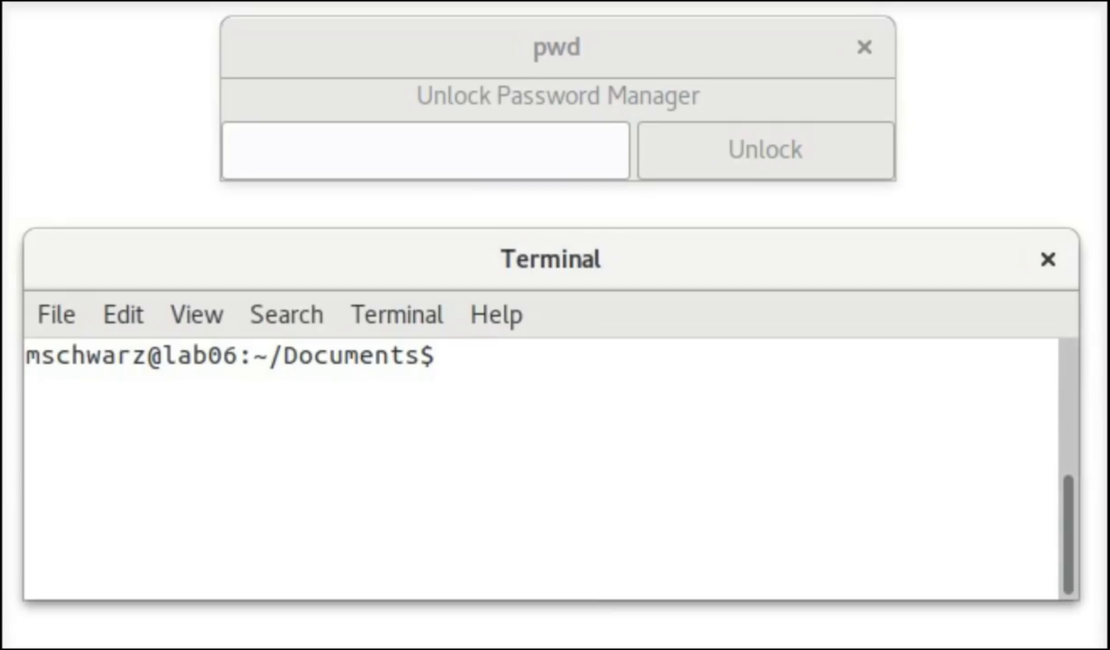
Building Meltdown



- Flush+Reload over all pages of the array

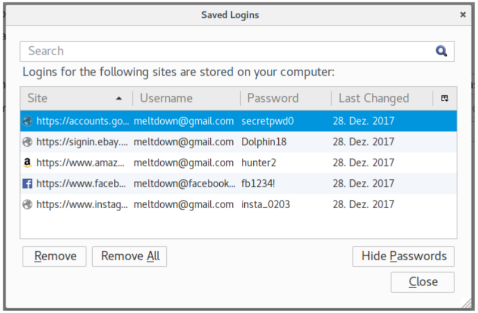


- Index of cache hit reveals data
- Permission check is in some cases not fast enough



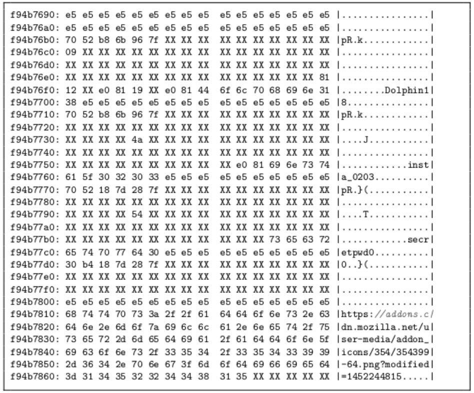
```
mschwarz@lab06:~/Documents$
```

Leaking Passwords from your Password Manager



Logins for the following sites are stored on your computer:

Site	Username	Password	Last Changed
https://accounts.go...	meltdown@gmail.com	secretpwd0	28. Dez. 2017
https://signin.ebay....	meltdown@gmail.com	Dolphin18	28. Dez. 2017
https://www.amaz...	meltdown@gmail.com	hunter2	28. Dez. 2017
https://www.faceb...	meltdown@facebook...	fb1234!	28. Dez. 2017
https://www.instag...	meltdown@gmail.com	insta_0203	28. Dez. 2017



```
f94b7690: e5 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |................|
f94b76a0: e5 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |................|
f94b76b0: 70 52 b8 6b 96 7f XX XX XX  XX XX XX XX XX XX XX |pR.k............|
f94b76c0: 09 XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b76d0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b76e0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX 81 |................|
f94b76f0: 12 XX e0 81 19 XX e0 81 44  6f 6c 70 68 69 6e 31 |........Dolphin1|
f94b7700: 38 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |8...............|
f94b7710: 70 52 b8 6b 96 7f XX XX XX  XX XX XX XX XX XX XX |pR.k............|
f94b7720: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7730: XX XX XX XX 4a XX XX XX XX  XX XX XX XX XX XX XX |....J...........|
f94b7740: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7750: XX XX XX XX XX XX XX XX XX  XX e0 81 69 6e 73 74 |............inst|
f94b7760: 61 5f 30 32 30 33 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |a_0203..........|
f94b7770: 70 52 18 7d 28 7f XX XX XX  XX XX XX XX XX XX XX |pR.}(...........|
f94b7780: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7790: XX XX XX XX 54 XX XX XX XX  XX XX XX XX XX XX XX |....T...........|
f94b77a0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b77b0: XX XX XX XX XX XX XX XX XX  XX XX XX 73 65 63 72 |............secr|
f94b77c0: 65 74 70 77 64 30 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |etpwd0..........|
f94b77d0: 30 b4 18 7d 28 7f XX XX XX  XX XX XX XX XX XX XX |0..}(...........|
f94b77e0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b77f0: XX XX XX XX XX XX XX XX XX  XX XX XX XX XX XX XX |................|
f94b7800: e5 e5 e5 e5 e5 e5 e5 e5 e5  e5 e5 e5 e5 e5 e5 e5 |................|
f94b7810: 68 74 74 70 73 3a 2f 2f 61  64 64 6f 6e 73 2e 63 |https://addons.c|
f94b7820: 64 6e 2e 6d 6f 7a 69 6c 6c  61 2e 6e 65 74 2f 75 |dn.mozilla.net/u|
f94b7830: 73 65 72 2d 6d 65 64 69 61  2f 61 64 64 6f 6e 5f |ser-media/addon_|
f94b7840: 69 63 6f 6e 73 2f 33 35 34  2f 33 35 34 33 39 39 |icons/354/354399|
f94b7850: 2d 36 34 2e 70 6e 67 3f 6d  6f 64 69 66 69 65 64 |-64.png?modified|
f94b7860: 3d 31 34 35 32 32 34 34 38  31 35 XX XX XX XX XX |=1452244815.....|
```

Kernel **A**ddress **I**solation to have **S**ide channels **E**fficiently **R**emoved

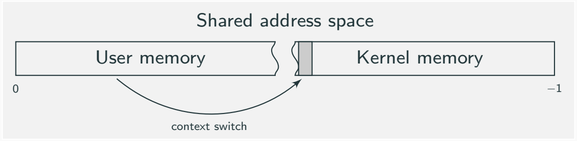
KAISER /ˈkʌɪzə/

1. [german] Emperor, ruler of an empire
2. largest penguin, emperor penguin

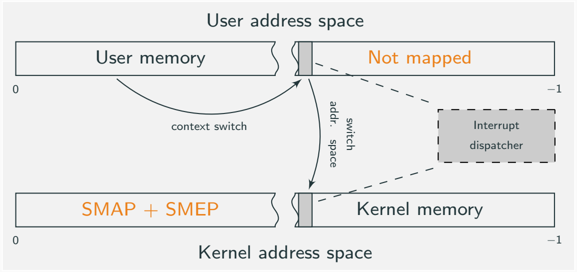
Kernel **A**ddress **I**solation to have **S**ide channels **E**fficiently **R**emoved

KAISER Illustration

Without KAISER:



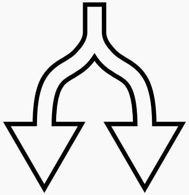
With KAISER:



KAISER

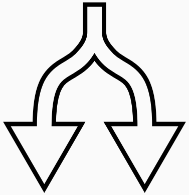
- We published KAISER in May 2017

KAISER



- We published KAISER in May 2017
- Intel and others improved and merged it into Linux as KPTI (Kernel Page Table Isolation)

KAISER



- We published KAISER in May 2017
- Intel and others improved and merged it into Linux as KPTI (Kernel Page Table Isolation)
- Microsoft implemented similar concept in Windows 10

KAISER

- We published KAISER in May 2017
- Intel and others improved and merged it into Linux as KPTI (Kernel Page Table Isolation)
- Microsoft implemented similar concept in Windows 10
- Apple implemented it in macOS 10.13.2 and called it “Double Map”

KAISER

- We published KAISER in May 2017
- Intel and others improved and merged it into Linux as KPTI (Kernel Page Table Isolation)
- Microsoft implemented similar concept in Windows 10
- Apple implemented it in macOS 10.13.2 and called it “Double Map”
- All share the same idea: switching address spaces on context switch

Meltdown and Spectre

MELTDOWN

SPECTRE

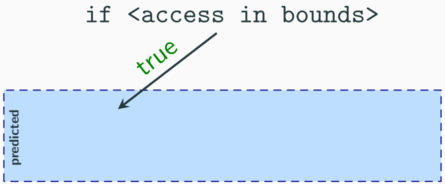
~~Meltdown and~~ Spectre

Branch Prediction

```
if <access in bounds>
```

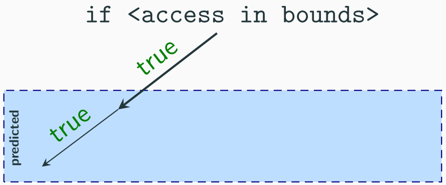
- processor predicts outcomes of branches
- predictions are based on previous behavior
- predictions help with executing more things in parallel

Branch Prediction



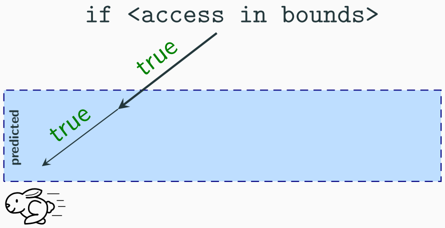
- processor predicts outcomes of branches
- predictions are based on previous behavior
- predictions help with executing more things in parallel

Branch Prediction



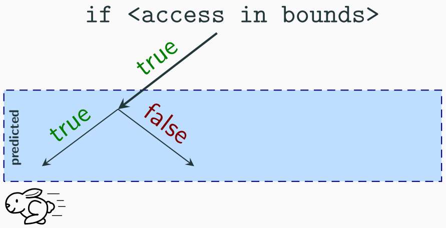
- processor predicts outcomes of branches
- predictions are based on previous behavior
- predictions help with executing more things in parallel

Branch Prediction



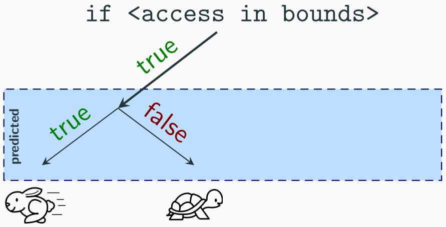
- processor predicts outcomes of branches
- predictions are based on previous behavior
- predictions help with executing more things in parallel

Branch Prediction



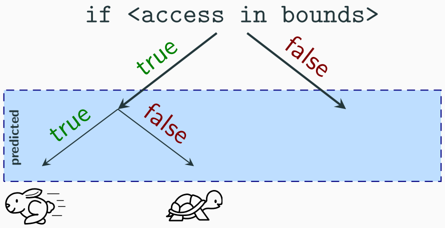
- processor predicts outcomes of branches
- predictions are based on previous behavior
- predictions help with executing more things in parallel

Branch Prediction



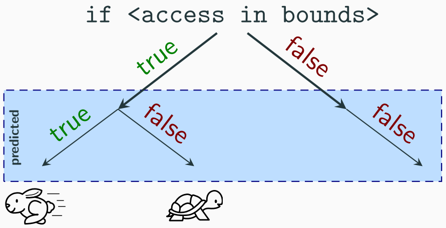
- processor predicts outcomes of branches
- predictions are based on previous behavior
- predictions help with executing more things in parallel

Branch Prediction



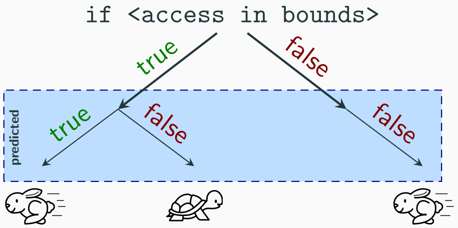
- processor predicts outcomes of branches
- predictions are based on previous behavior
- predictions help with executing more things in parallel

Branch Prediction



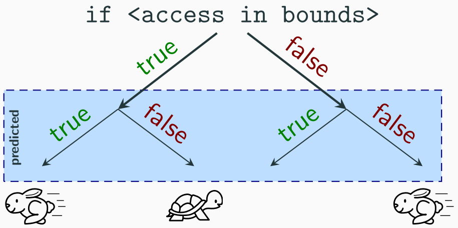
- processor predicts outcomes of branches
- predictions are based on previous behavior
- predictions help with executing more things in parallel

Branch Prediction



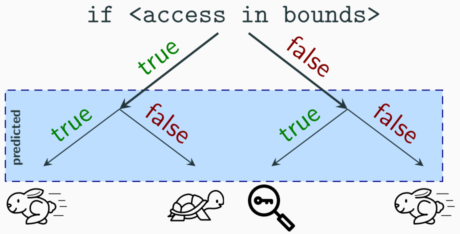
- processor predicts outcomes of branches
- predictions are based on previous behavior
- predictions help with executing more things in parallel

Branch Prediction



- processor predicts outcomes of branches
- predictions are based on previous behavior
- predictions help with executing more things in parallel

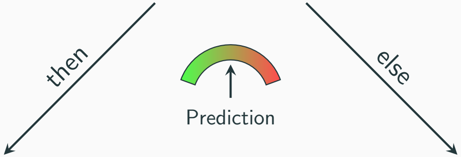
Branch Prediction



- processor predicts outcomes of branches
- predictions are based on previous behavior
- predictions help with executing more things in parallel

Spectre Variant 1

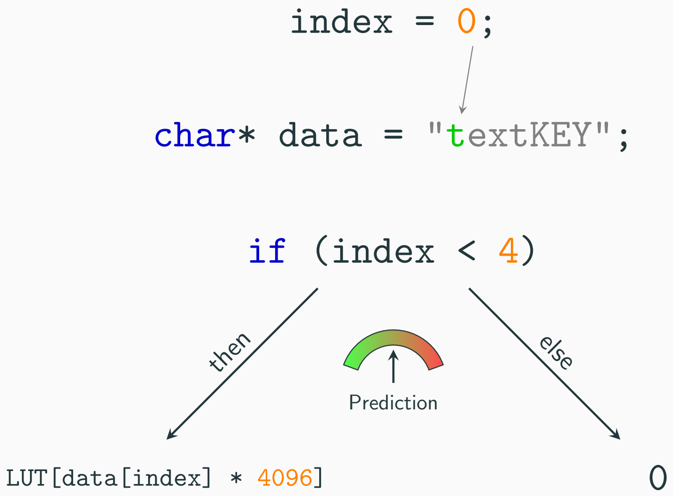
```
index = 0;

char* data = "textKEY";

if (index < 4)
```



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

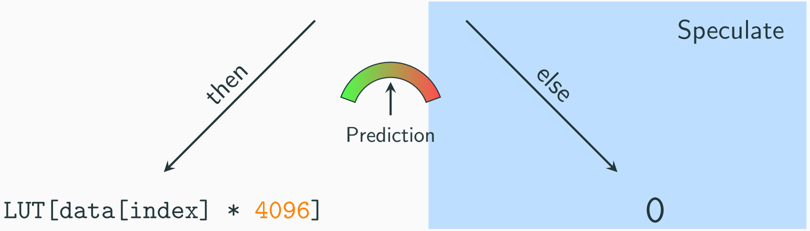
0

Spectre Variant 1

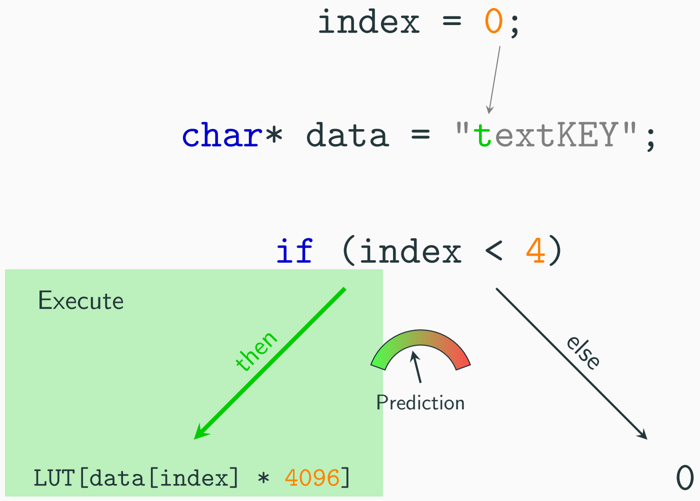

```
index = 0;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]                0
```

Spectre Variant 1

```
index = 0;

char* data = "textKEY";

if (index < 4)
```



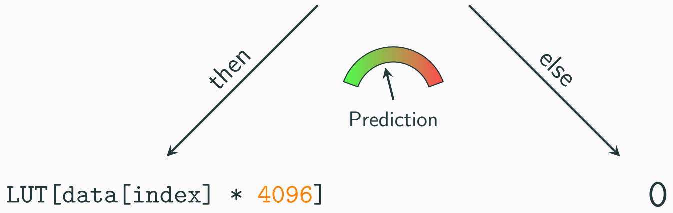
Spectre Variant 1



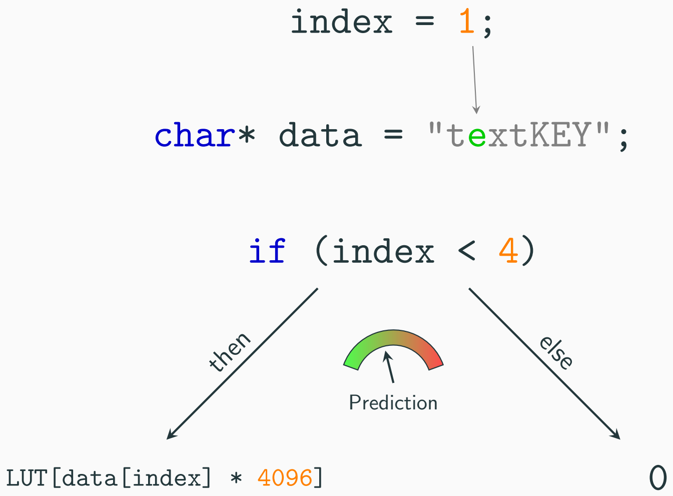
```
index = 0;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

0
```

Spectre Variant 1

```
index = 1;

char* data = "textKEY";

if (index < 4)
```



Spectre Variant 1

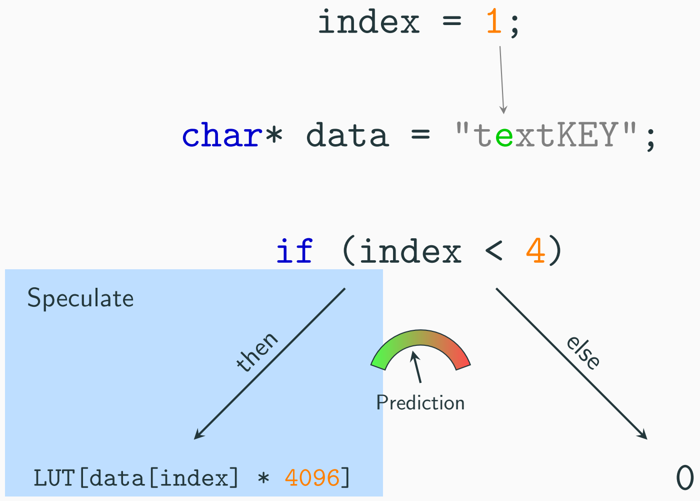


```
index = 1;

char* data = "textKEY";

if (index < 4)
```

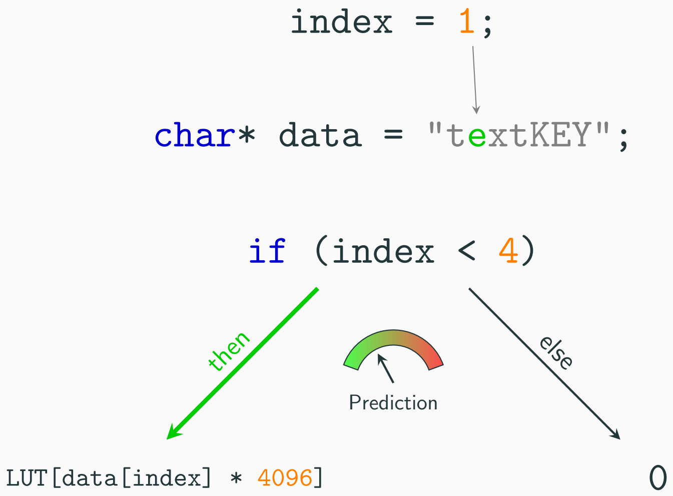
```
LUT[data[index] * 4096]
```

```
0
```

Spectre Variant 1



Spectre Variant 1

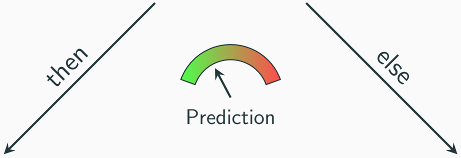


```
index = 1;

char* data = "textKEY";

if (index < 4)
```

```
LUT[data[index] * 4096]          0
```

Spectre Variant 1

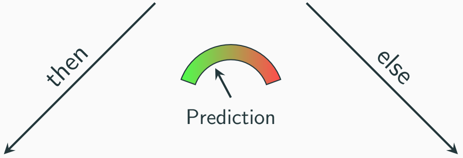
```
index = 2;

char* data = "textKEY";

if (index < 4)
```



```
LUT[data[index] * 4096]                0
```

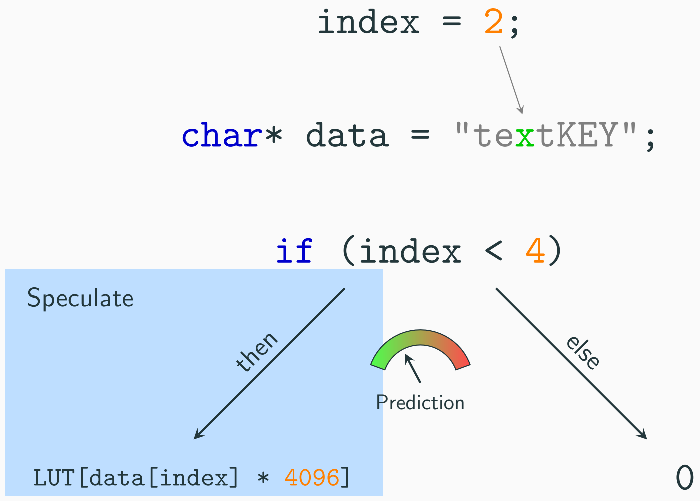
Spectre Variant 1

```
index = 2;

char* data = "textKEY";

if (index < 4)
```

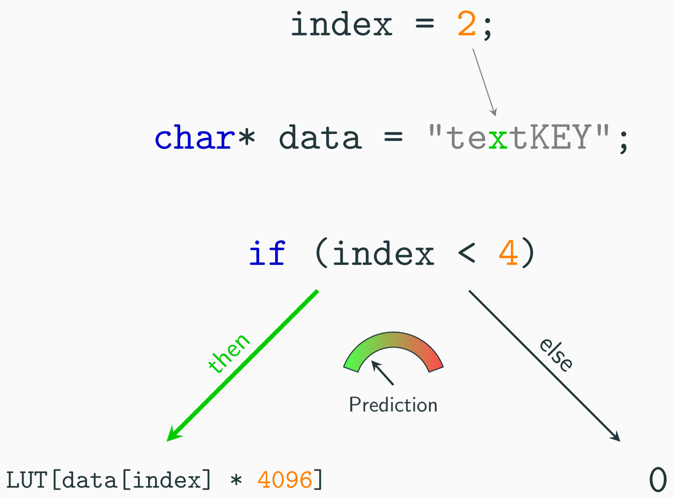


```
LUT[data[index] * 4096]                0
```

Spectre Variant 1



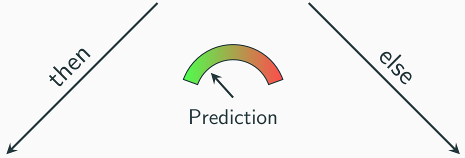
Spectre Variant 1



```
index = 2;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

0
```

Spectre Variant 1

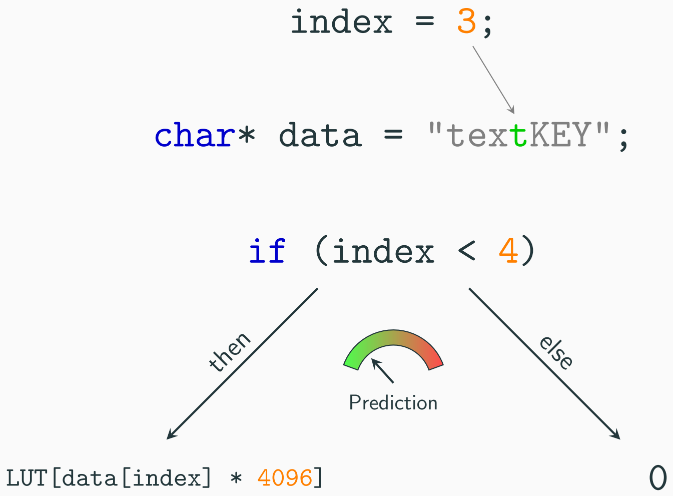
```
index = 3;

char* data = "textKEY";

if (index < 4)
```



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

0

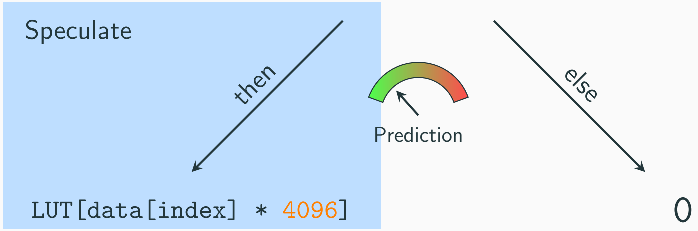
Spectre Variant 1



```
index = 3;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

0
```

Spectre Variant 1

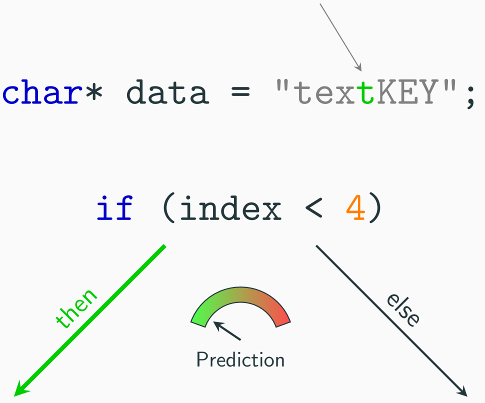
```
index = 3;

char* data = "textKEY";

if (index < 4)
```



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

Spectre Variant 1

```
index = 3;

char* data = "textKEY";

if (index < 4)
```

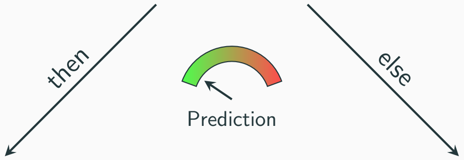


```
LUT[data[index] * 4096]
```

0

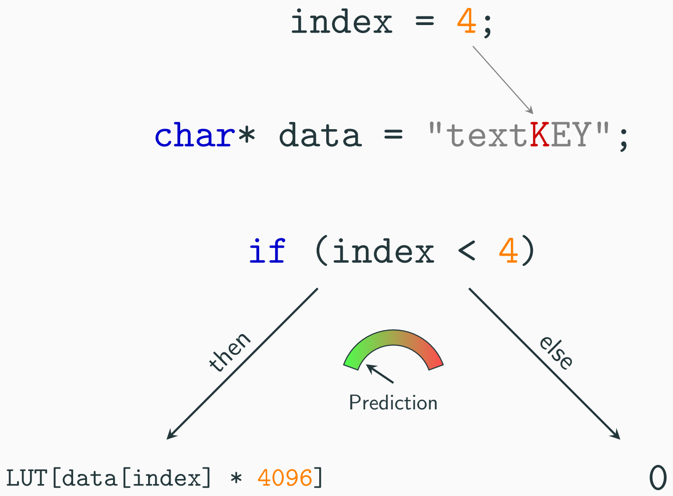
Spectre Variant 1

```
index = 4;

char* data = "textKEY";

if (index < 4)
```



```
LUT[data[index] * 4096]                0
```

Spectre Variant 1

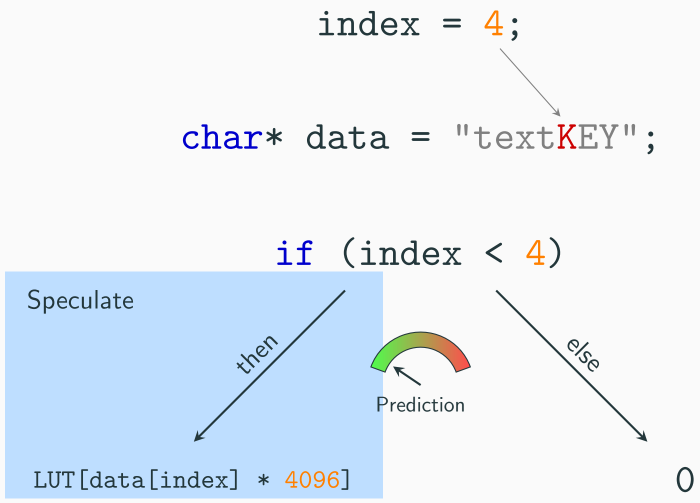


```
index = 4;

char* data = "textKEY";

if (index < 4)
```

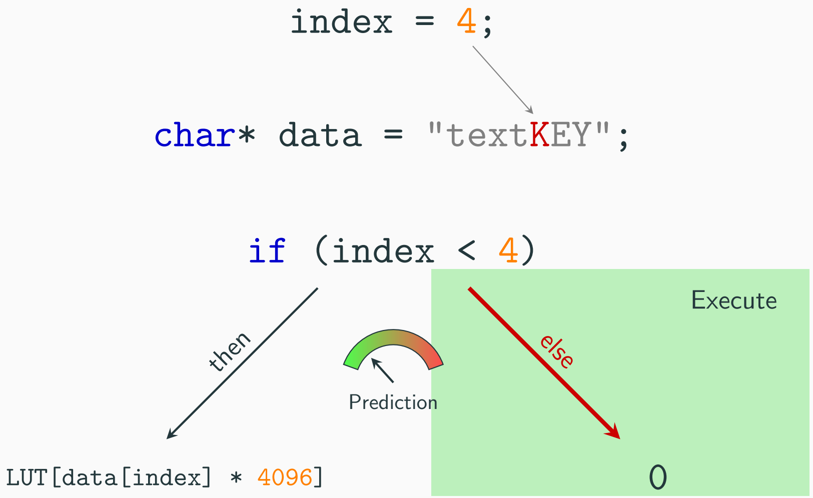
```
LUT[data[index] * 4096]
```

```
0
```

Spectre Variant 1



Spectre Variant 1

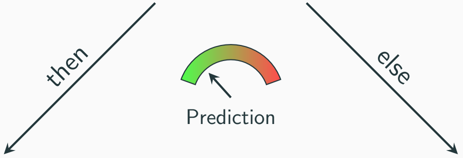


```
index = 4;

char* data = "textKEY";

if (index < 4)
```

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

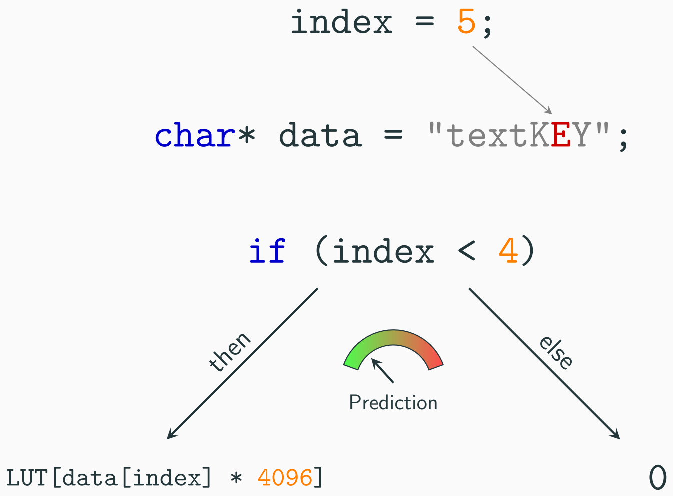
Spectre Variant 1

```
index = 5;

char* data = "textKEY";

if (index < 4)
```



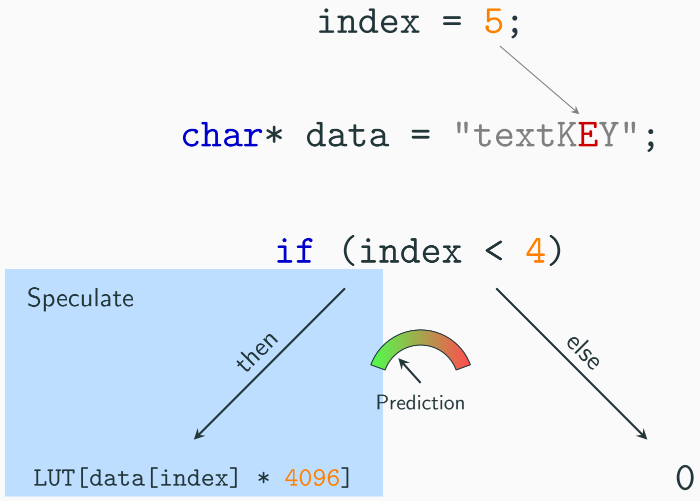
```
LUT[data[index] * 4096]                 0
```

Spectre Variant 1



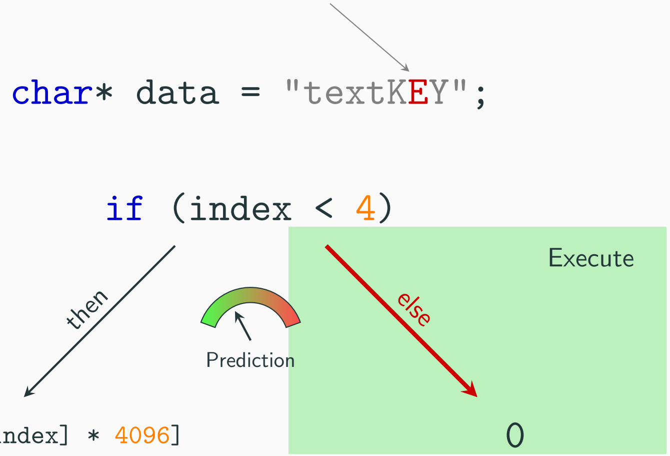
```
index = 5;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

0
```

Spectre Variant 1



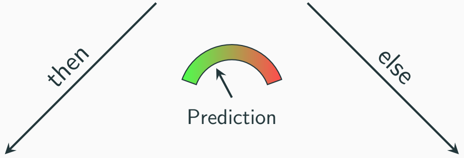
Spectre Variant 1

```
index = 5;

char* data = "textKEY";

if (index < 4)
```



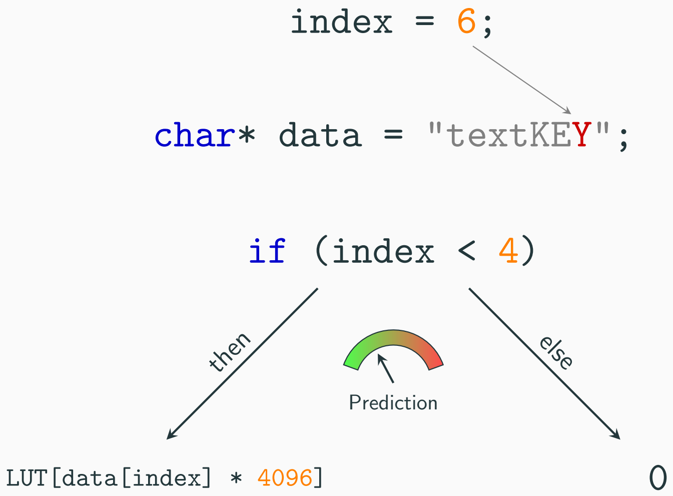
Spectre Variant 1

```
index = 6;

char* data = "textKEY";

if (index < 4)
```



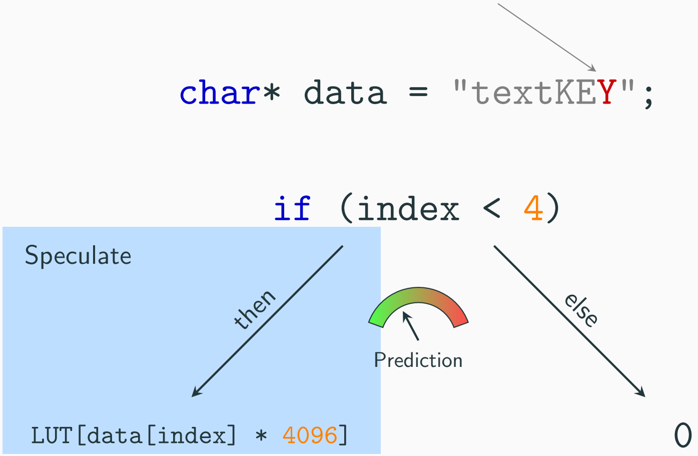
```
LUT[data[index] * 4096]          0
```

Spectre Variant 1



Spectre Variant 1

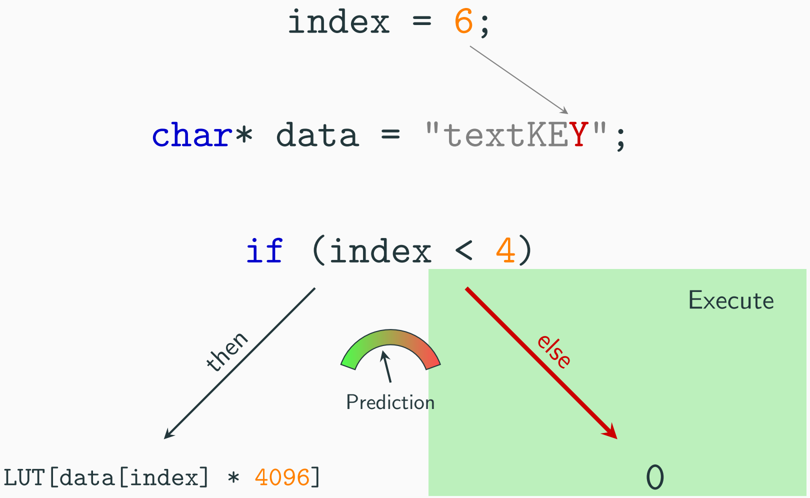
```
index = 6;

char* data = "textKEY";

if (index < 4)
```



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

```
0
```

Spectre Variant 1



```
index = 6;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

0
```

Branch Prediction: Other Patterns (Untested)

- type check

Branch Prediction: Other Patterns (Untested)

- type check
- out-of-bounds access into object table with function pointers

```c
struct foo_ops {
  void (*bar)(void);
};
struct foo {
  struct foo_ops *ops;
};

struct foo **foo_array;
size_t foo_array_len;

void do_bar(size_t idx) {
  if (idx >= foo_array_len) return;
  foo_array[idx]->ops->bar();
}
```

Spectre Variant 2: Indirect Branches

```
kvm_x86_ops->handle_external_intr(vcpu);

struct kvm_x86_ops *kvm_x86_ops;

static struct kvm_x86_ops vmx_x86_ops = {
[...]
  .handle_external_intr =
    vmx_handle_external_intr,
[...]
};
```

(code simplified)

Spectre Variant 2: Indirect Branches

- instruction stream does not contain target address

```
kvm_x86_ops->handle_external_intr(vcpu);

struct kvm_x86_ops *kvm_x86_ops;

static struct kvm_x86_ops vmx_x86_ops = {
[...]
  .handle_external_intr =
   vmx_handle_external_intr,
[...]
};
```

(code simplified)

Spectre Variant 2: Indirect Branches

- instruction stream does not contain target address
- target must be fetched from memory

```
kvm_x86_ops->handle_external_intr(vcpu);

struct kvm_x86_ops *kvm_x86_ops;

static struct kvm_x86_ops vmx_x86_ops = {
[...]
  .handle_external_intr =
   vmx_handle_external_intr,
[...]
};
```

(code simplified)

Spectre Variant 2: Indirect Branches

- instruction stream does not contain target address
- target must be fetched from memory
- CPU will speculate about branch target

```
kvm_x86_ops->handle_external_intr(vcpu);

struct kvm_x86_ops *kvm_x86_ops;

static struct kvm_x86_ops vmx_x86_ops = {
[...]
  .handle_external_intr =
    vmx_handle_external_intr,
[...]
};
```

(code simplified)

Branch Prediction

Branch Prediction

- state is stored in a Branch Target Buffer (BTB)

Branch Prediction

- state is stored in a Branch Target Buffer (BTB)
  - indexed and tagged by (on Intel Haswell):

Branch Prediction

- state is stored in a Branch Target Buffer (BTB)
  - indexed and tagged by (on Intel Haswell):
    - partial virtual address

Branch Prediction

- state is stored in a Branch Target Buffer (BTB)
  - indexed and tagged by (on Intel Haswell):
    - partial virtual address
    - recent branch history fingerprint [sometimes]

Branch Prediction

- state is stored in a Branch Target Buffer (BTB)
  - indexed and tagged by (on Intel Haswell):
    - partial virtual address
    - recent branch history fingerprint [sometimes]
- allowed to be wrong

Branch Prediction

- state is stored in a Branch Target Buffer (BTB)
    - indexed and tagged by (on Intel Haswell):
        - partial virtual address
        - recent branch history fingerprint [sometimes]
- allowed to be wrong

Branch Prediction

- state is stored in a Branch Target Buffer (BTB)
  - indexed and tagged by (on Intel Haswell):
    - partial virtual address
    - recent branch history fingerprint [sometimes]
- allowed to be wrong
- often not tagged by security domain

Branch Prediction

- state is stored in a Branch Target Buffer (BTB)
  - indexed and tagged by (on Intel Haswell):
    - partial virtual address
    - recent branch history fingerprint [sometimes]
- allowed to be wrong
- often not tagged by security domain

→ Break ASLR across security domains (“Jump over ASLR” paper)

Spectre Variant 2 Idea

- Why not also the other way round?

Spectre Variant 2 Idea

- Why not also the other way round?
- **Inject misspeculation** to controlled addresses across security domains

Spectre Variant 2 Idea

- Why not also the other way round?
- **Inject misspeculation** to controlled addresses across security domains
- Attack goal: Leak host memory from inside a KVM guest

Known Predictor Internals

- direct branches:

Known Predictor Internals

- direct branches:
    - bits 0-30 of the source go into BTB indexing function

Known Predictor Internals

- direct branches:
    - bits 0-30 of the source go into BTB indexing function
    - BTB collisions possible between different security contexts

Known Predictor Internals

- direct branches:
  - bits 0-30 of the source go into BTB indexing function
  - BTB collisions possible between different security contexts
- predictions are calculated for 32-byte blocks of source instructions

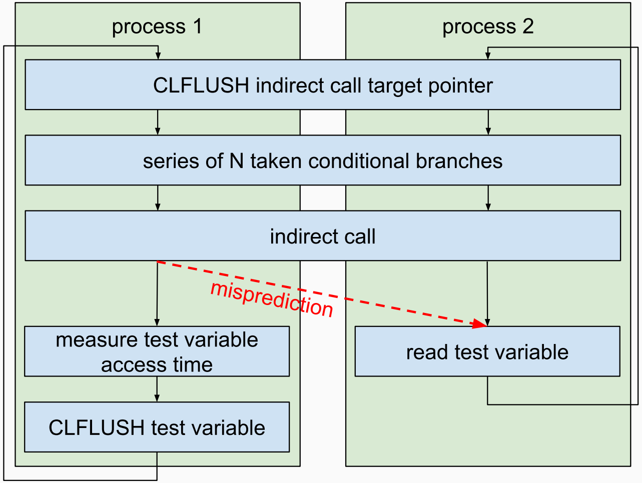
Known Predictor Internals

- direct branches:
    - bits 0-30 of the source go into BTB indexing function
    - BTB collisions possible between different security contexts
- predictions are calculated for 32-byte blocks of source instructions
- conditional branches: predicts both taken/not taken and target address

Known Predictor Internals

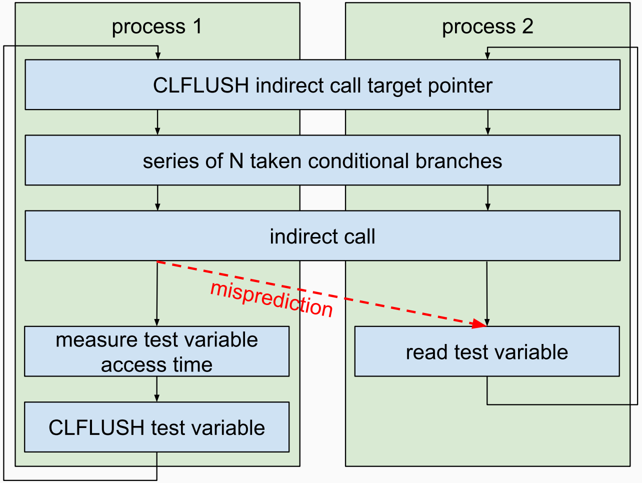
- direct branches:
  - bits 0-30 of the source go into BTB indexing function
  - BTB collisions possible between different security contexts
- predictions are calculated for 32-byte blocks of source instructions
- conditional branches: predicts both taken/not taken and target address
- indirect branches: two prediction modes:
  - “monotonic target”
  - “targets that vary in accordance with recent program behavior”

Predictor Reversing: Minimal Test


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches
indirect call
misprediction
measure test variable access time
read test variable
CLFLUSH test variable

(explicit execution barriers omitted from diagram)

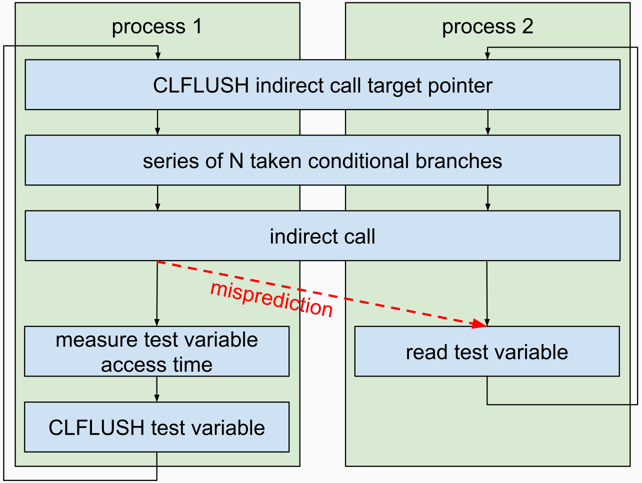
Predictor Reversing: Minimal Test


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches
indirect call
misprediction
measure test variable access time
read test variable
CLFLUSH test variable

(explicit execution barriers omitted from diagram)

- hyperthreaded

Predictor Reversing: Minimal Test

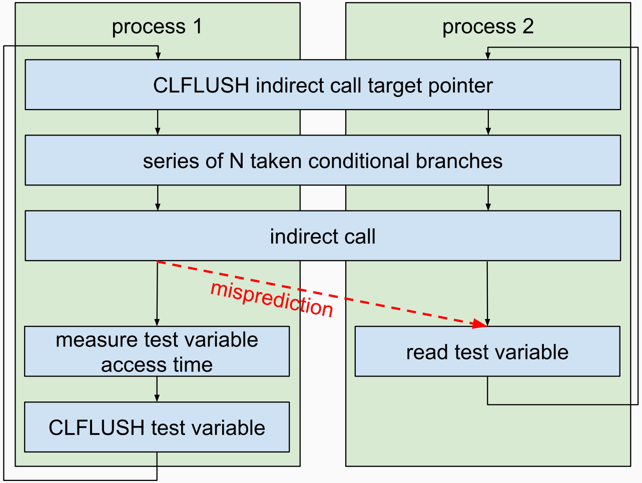


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches
indirect call
misprediction
measure test variable access time
read test variable
CLFLUSH test variable

(explicit execution barriers omitted from diagram)

- hyperthreaded
- same code

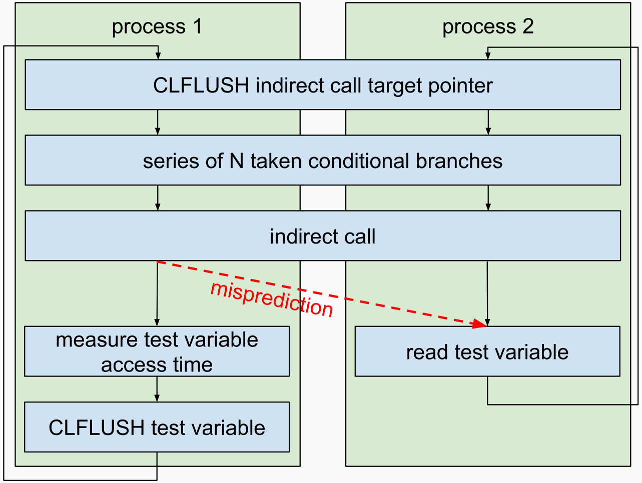
Predictor Reversing: Minimal Test


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches
indirect call
misprediction
measure test variable access time
read test variable
CLFLUSH test variable

(explicit execution barriers omitted from diagram)

- hyperthreaded
- same code
- same memory layout (no ASLR)

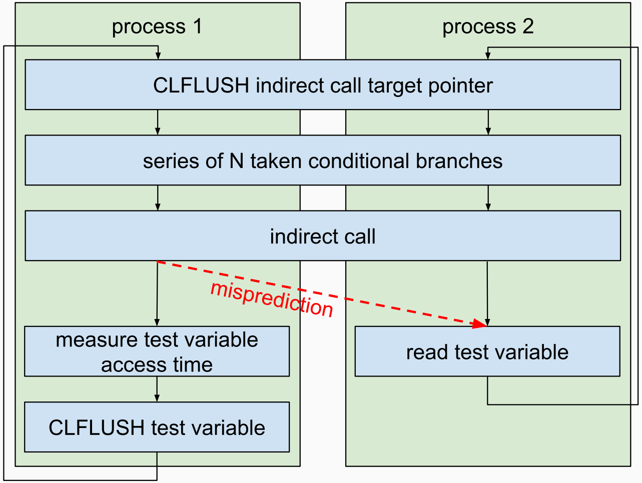
Predictor Reversing: Minimal Test


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches
indirect call
misprediction
measure test variable access time
CLFLUSH test variable
read test variable

(explicit execution barriers omitted from diagram)

- hyperthreaded
- same code
- same memory layout (no ASLR)
- different indirect call targets

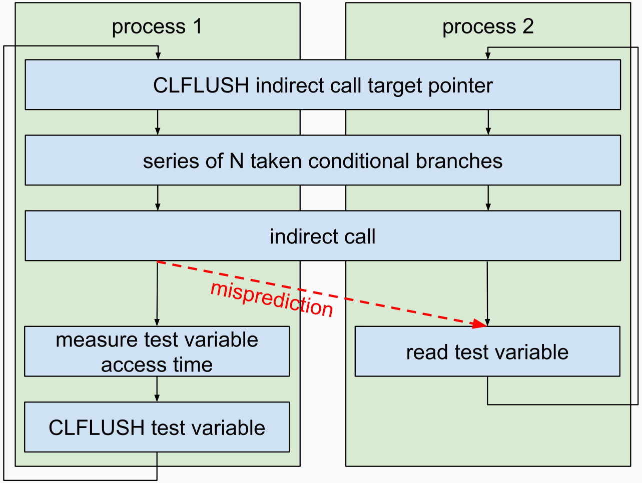
Predictor Reversing: Minimal Test


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches
indirect call
misprediction
measure test variable access time
read test variable
CLFLUSH test variable

(explicit execution barriers omitted from diagram)

- hyperthreaded
- same code
- same memory layout (no ASLR)
- different indirect call targets
- process 1: *Flush+Reload* loop (always miss)

Predictor Reversing: Minimal Test

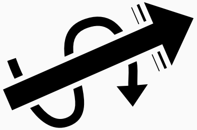

process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches
indirect call
misprediction
measure test variable access time
read test variable
CLFLUSH test variable

(explicit execution barriers omitted from diagram)

- hyperthreaded
- same code
- same memory layout (no ASLR)
- different indirect call targets
- process 1: *Flush+Reload* loop (always miss)
- target injection from process 2 can cause extra load

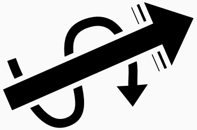
Variant 2: first brittle PoC [in initial writeup]

- shortcuts for minimal PoC



Variant 2: first brittle PoC [in initial writeup]

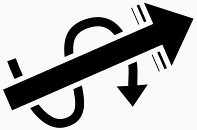
- shortcuts for minimal PoC
- BTB structure from prior research ("Jump over ASLR" paper)



Variant 2: first brittle PoC [in initial writeup]

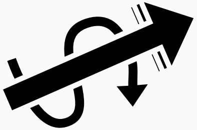
- shortcuts for minimal PoC
- BTB structure from prior research ("Jump over ASLR" paper)
  - Source address: low 31 bits

Variant 2: first brittle PoC [in initial writeup]



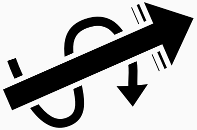
- shortcuts for minimal PoC
- BTB structure from prior research ("Jump over ASLR" paper)
    - Source address: low 31 bits
    - ... direct branches only

Variant 2: first brittle PoC [in initial writeup]



- shortcuts for minimal PoC
- BTB structure from prior research ("Jump over ASLR" paper)
    - Source address: low 31 bits
    - ... direct branches only
- collide low 31 bits of source address, assume relative target

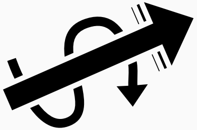
Variant 2: first brittle PoC [in initial writeup]



- shortcuts for minimal PoC
- BTB structure from prior research ("Jump over ASLR" paper)
  - Source address: low 31 bits
  - ... direct branches only
- collide low 31 bits of source address, assume relative target

$\rightarrow$ leak rate: $\approx$ 6 bits/second

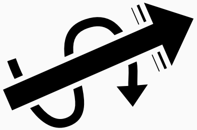
Variant 2: first brittle PoC [in initial writeup]



- shortcuts for minimal PoC
- BTB structure from prior research ("Jump over ASLR" paper)
  - Source address: low 31 bits
  - ... direct branches only
- collide low 31 bits of source address, assume relative target

$\rightarrow$ leak rate: $\approx$ 6 bits/second — almost all the injection attempts fail!

Variant 2: first brittle PoC [in initial writeup]

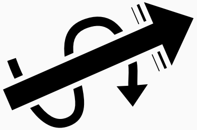


- shortcuts for minimal PoC
- BTB structure from prior research (“Jump over ASLR” paper)
    - Source address: low 31 bits
    - ... direct branches only
- collide low 31 bits of source address, assume relative target

→ leak rate: ≈ 6 bits/second — almost all the injection attempts fail!

→ CPU distinguishes injections and hypervisor execution

Variant 2: first brittle PoC [in initial writeup]



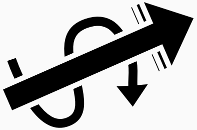
- shortcuts for minimal PoC
- BTB structure from prior research ("Jump over ASLR" paper)
  - Source address: low 31 bits
  - ... direct branches only
- collide low 31 bits of source address, assume relative target

→ leak rate: ≈ 6 bits/second — almost all the injection attempts fail!

→ CPU distinguishes injections and hypervisor execution

→ Theory:

Variant 2: first brittle PoC [in initial writeup]



- shortcuts for minimal PoC
- BTB structure from prior research (“Jump over ASLR” paper)
  - Source address: low 31 bits
  - ... direct branches only
- collide low 31 bits of source address, assume relative target

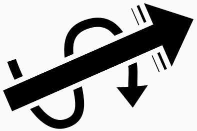
→ leak rate: ≈ 6 bits/second — almost all the injection attempts fail!

→ CPU distinguishes injections and hypervisor execution

→ Theory:

  - injection only works for “monotonic target” prediction

Variant 2: first brittle PoC [in initial writeup]



- shortcuts for minimal PoC
- BTB structure from prior research ("Jump over ASLR" paper)
  - Source address: low 31 bits
  - ... direct branches only
- collide low 31 bits of source address, assume relative target

→ leak rate: ≈ 6 bits/second — almost all the injection attempts fail!

→ CPU distinguishes injections and hypervisor execution

→ Theory:

  - injection only works for "monotonic target" prediction
  - CPU prefers history-based prediction

Variant 2: first brittle PoC [in initial writeup]

- shortcuts for minimal PoC
- BTB structure from prior research ("Jump over ASLR" paper)
  - Source address: low 31 bits
  - ... direct branches only
- collide low 31 bits of source address, assume relative target

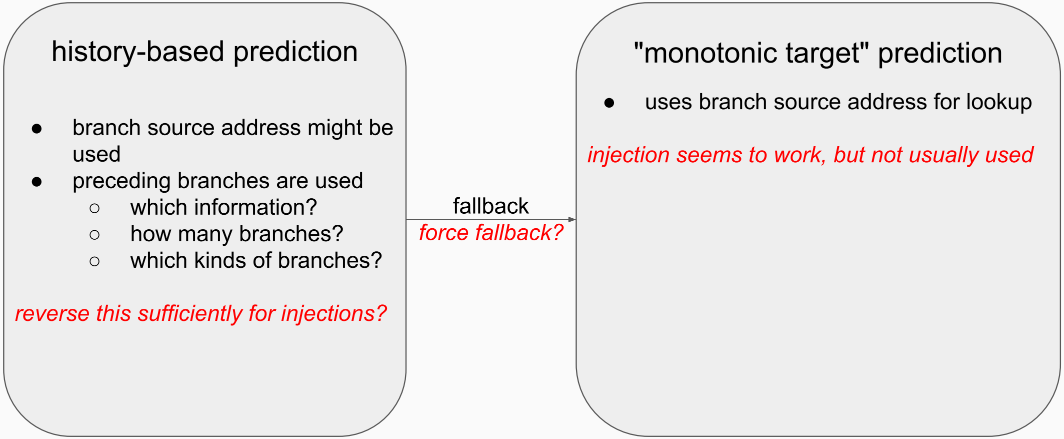
→ leak rate: ≈ 6 bits/second — almost all the injection attempts fail!

→ CPU distinguishes injections and hypervisor execution

→ Theory:

  - injection only works for "monotonic target" prediction
  - CPU prefers history-based prediction
  - injection works when history-based prediction fails due to system noise causing evictions

Branch Prediction Model



history-based prediction

- branch source address might be used
- preceding branches are used
    - which information?
    - how many branches?
    - which kinds of branches?

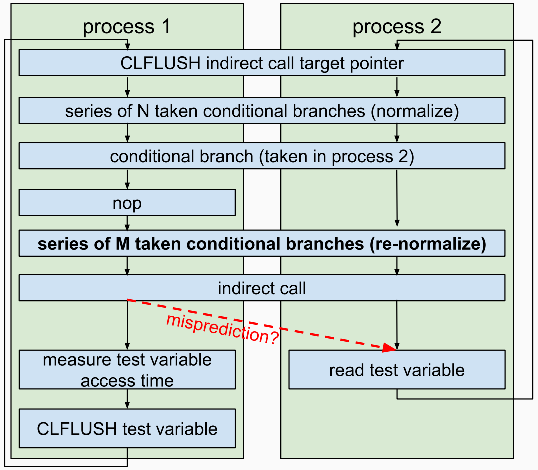
reverse this sufficiently for injections?

"monotonic target" prediction

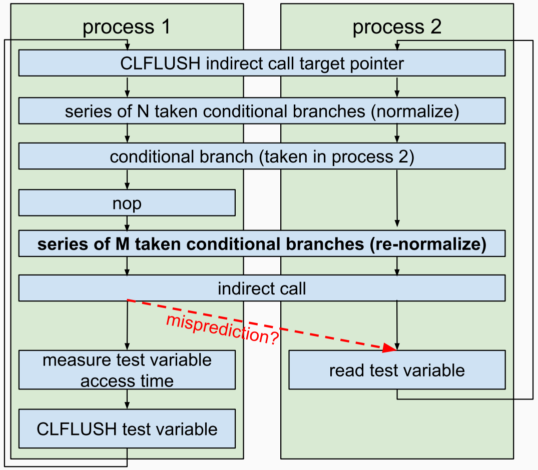
- uses branch source address for lookup

injection seems to work, but not usually used

Predictor Reversing: History Length


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches (normalize)
conditional branch (taken in process 2)
nop
series of M taken conditional branches (re-normalize)
indirect call
misprediction?
measure test variable access time
read test variable
CLFLUSH test variable

Predictor Reversing: History Length

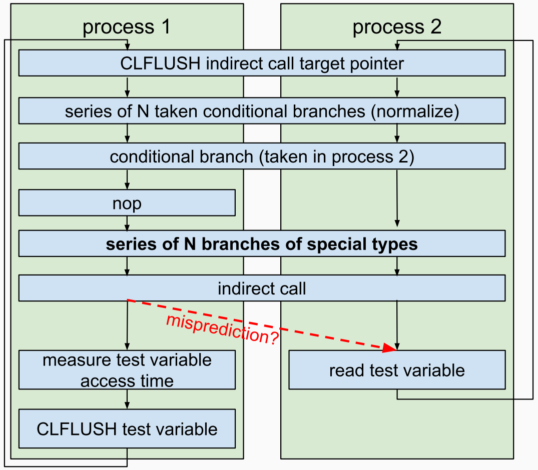


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches (normalize)
conditional branch (taken in process 2)
nop
series of M taken conditional branches (re-normalize)
indirect call
misprediction?
measure test variable access time
CLFLUSH test variable
read test variable

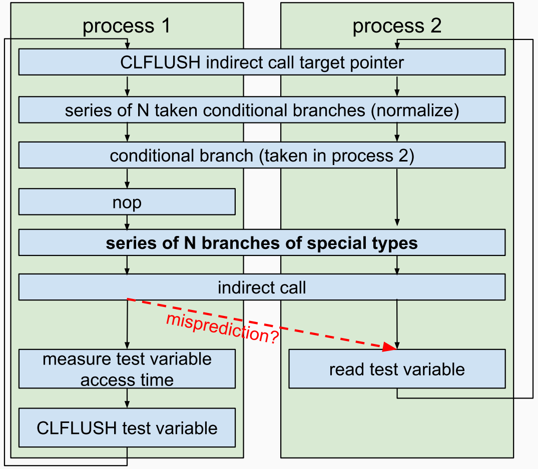
on Haswell:

- $\approx$ 26 branches stored
- measurements get weird around the boundary [and are not yet entirely correct]

Predictor Reversing: Branch Types

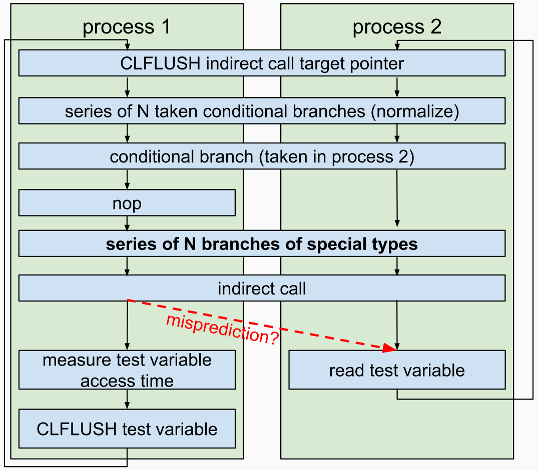

process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches (normalize)
conditional branch (taken in process 2)
nop
series of N branches of special types
indirect call
misprediction?
measure test variable access time
read test variable
CLFLUSH test variable

Predictor Reversing: Branch Types


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches (normalize)
conditional branch (taken in process 2)
nop
series of N branches of special types
indirect call
misprediction?
measure test variable access time
read test variable
CLFLUSH test variable

on Haswell:

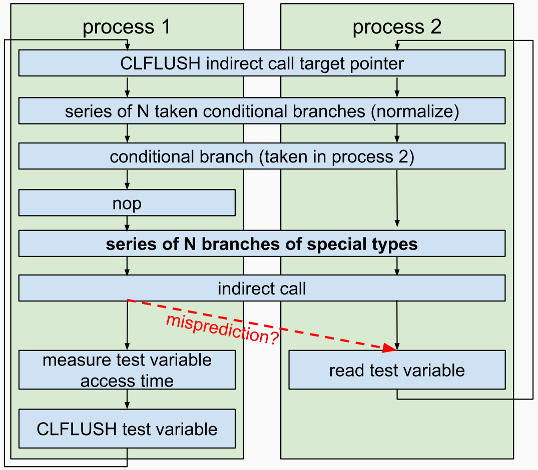
Predictor Reversing: Branch Types


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches (normalize)
conditional branch (taken in process 2)
nop
series of N branches of special types
indirect call
misprediction?
measure test variable access time
read test variable
CLFLUSH test variable

on Haswell:

- taken conditional branch ✓

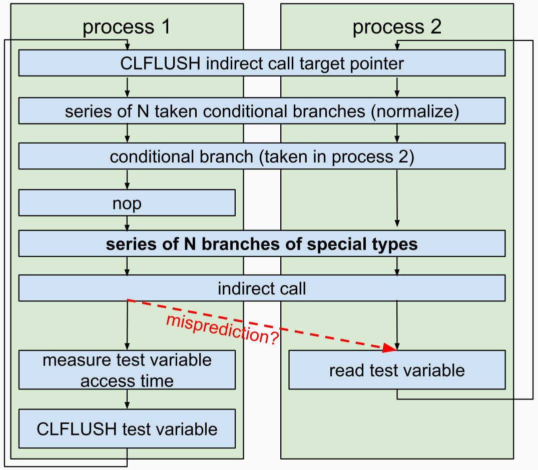
Predictor Reversing: Branch Types


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches (normalize)
conditional branch (taken in process 2)
nop
series of N branches of special types
indirect call
misprediction?
measure test variable access time
read test variable
CLFLUSH test variable

on Haswell:

- taken conditional branch ✓
- not-taken conditional branch ✗

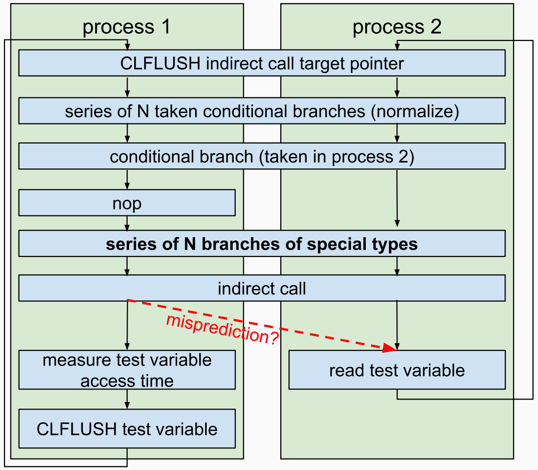
Predictor Reversing: Branch Types


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches (normalize)
conditional branch (taken in process 2)
nop
series of N branches of special types
indirect call
misprediction?
measure test variable access time
read test variable
CLFLUSH test variable

on Haswell:

- taken conditional branch ✓
- not-taken conditional branch ✗
- unconditional direct jump ✓

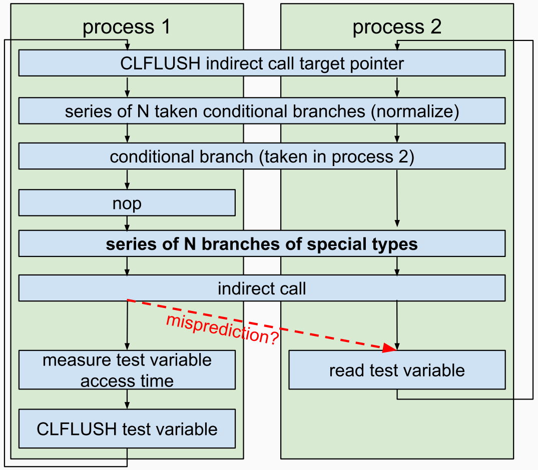
Predictor Reversing: Branch Types


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches (normalize)
conditional branch (taken in process 2)
nop
series of N branches of special types
indirect call
misprediction?
measure test variable access time
read test variable
CLFLUSH test variable

on Haswell:

- taken conditional branch ✓
- not-taken conditional branch ✗
- unconditional direct jump ✓
- unconditional indirect branch ✓

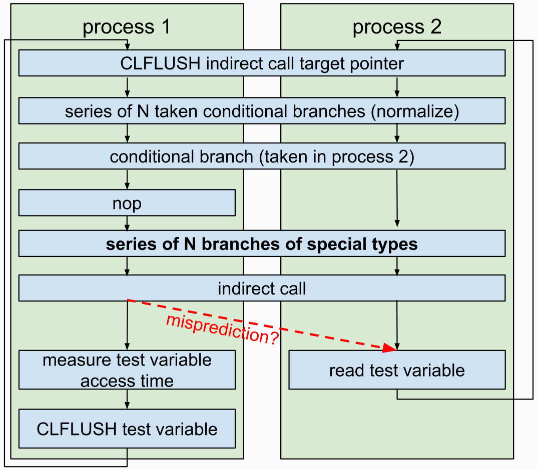
Predictor Reversing: Branch Types


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches (normalize)
conditional branch (taken in process 2)
nop
series of N branches of special types
indirect call
misprediction?
measure test variable access time
read test variable
CLFLUSH test variable

on Haswell:

- taken conditional branch ✓
- not-taken conditional branch ✗
- unconditional direct jump ✓
- unconditional indirect branch ✓
- RET ✓

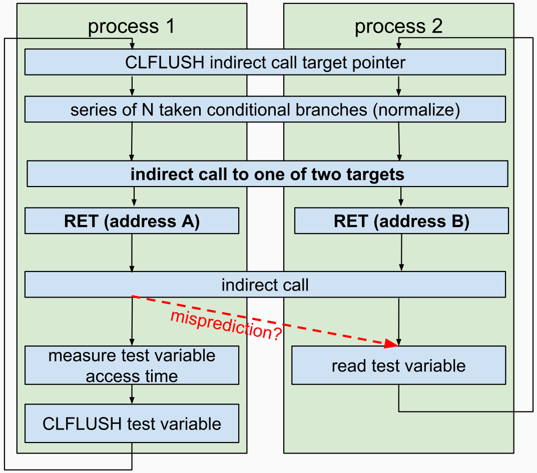
Predictor Reversing: Branch Types


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches (normalize)
conditional branch (taken in process 2)
nop
series of N branches of special types
indirect call
misprediction?
measure test variable access time
read test variable
CLFLUSH test variable

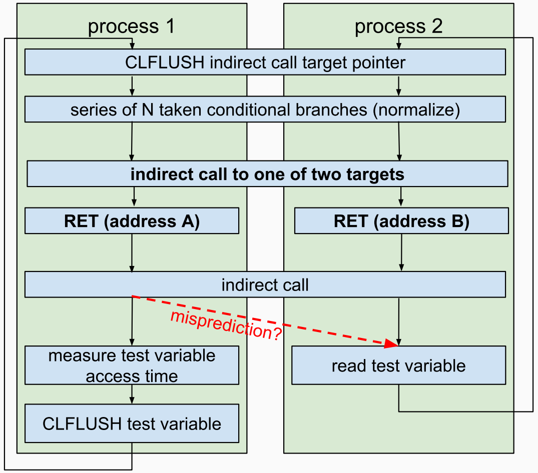
on Haswell:

- taken conditional branch ✓
- not-taken conditional branch ✗
- unconditional direct jump ✓
- unconditional indirect branch ✓
- RET ✓
- **IRETQ** ✗

Address Bits in History


process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches (normalize)
indirect call to one of two targets
RET (address A)
RET (address B)
indirect call
misprediction?
measure test variable access time
read test variable
CLFLUSH test variable

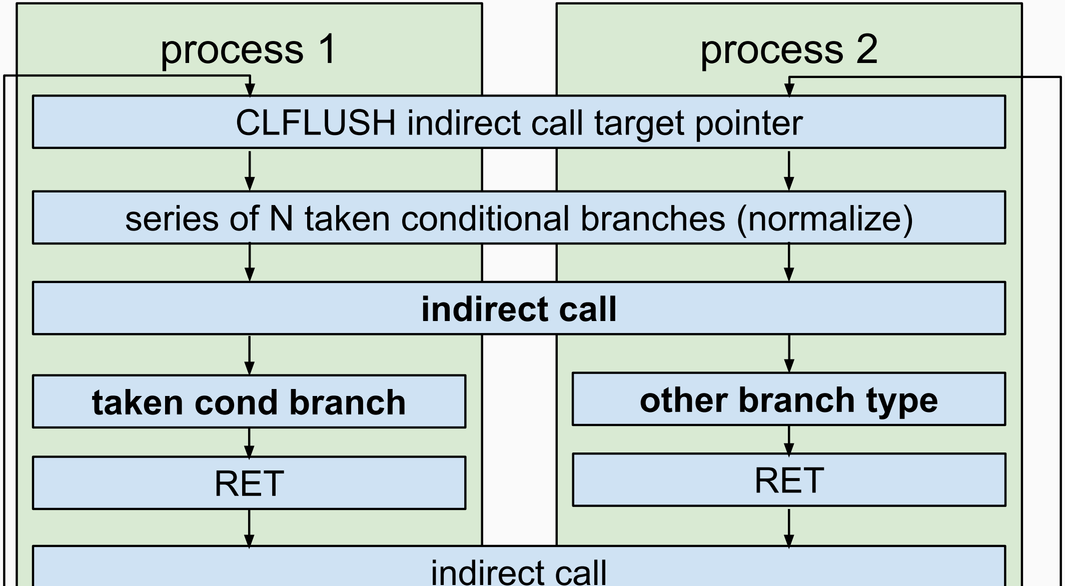
Address Bits in History



process 1
process 2
CLFLUSH indirect call target pointer
series of N taken conditional branches (normalize)
indirect call to one of two targets
RET (address A)
RET (address B)
indirect call
misprediction?
measure test variable access time
read test variable
CLFLUSH test variable

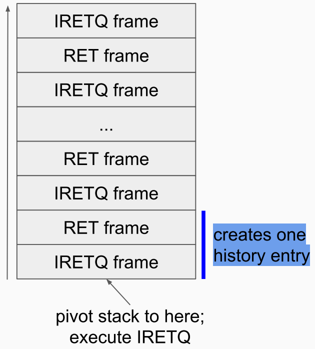
→ only low 20 bits of any address affect history

Predictor Reversing: Branch Type influence?



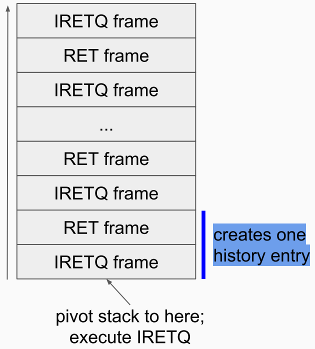
Full History Control

- kinda like ROP



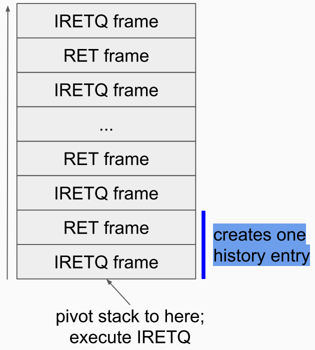
Full History Control

- kinda like ROP
- use RET instructions to add history entries
    - RET reads a target from RSP, jumps to the target, and advances RSP in one byte
    - RET target is fed into predictor as target
    - RET target is always an IRETQ



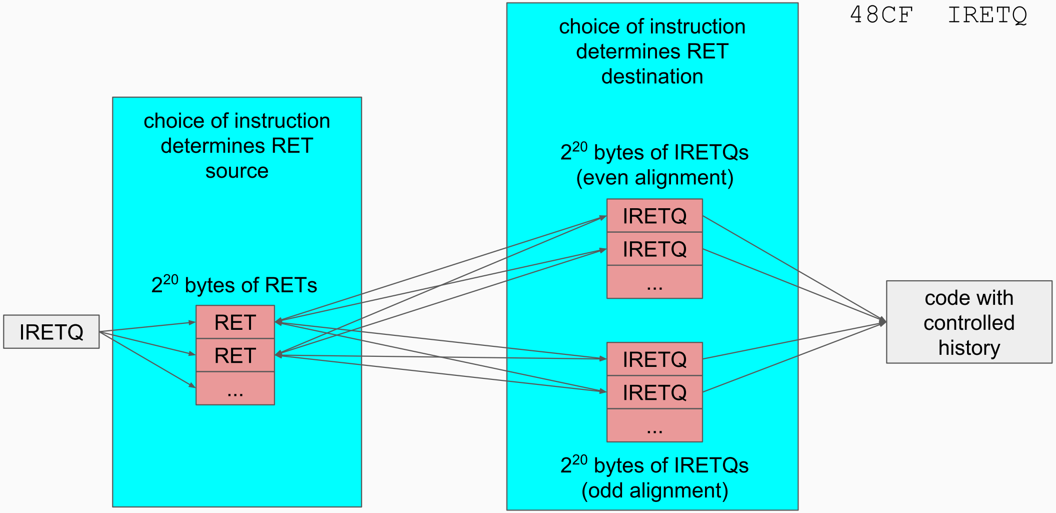
Full History Control

- kinda like ROP
- use RET instructions to add history entries
  - RET reads a target from RSP, jumps to the target, and advances RSP in one byte
  - RET target is fed into predictor as target
  - RET target is always an IRETQ
- use IRETQ instructions to move between RET instructions
  - IRETQ target is fed into predictor as source (by the following RET)
  - IRETQ target, apart from the last one, is always RET



Full History Control

```
C3      RET
48CF    IRETQ
```


IRETQ
choice of instruction determines RET source
2^{20} bytes of RETs
RET
RET
...
choice of instruction determines RET destination
2^{20} bytes of IRETQs (even alignment)
IRETQ
IRETQ
...
IRETQ
IRETQ
...
2^{20} bytes of IRETQs (odd alignment)
code with controlled history

History Buffer Structure (Haswell)

- a predictor with one bit of history (taken / not taken) per conditional branch [Agner Fog]

History Buffer Structure (Haswell)

- a predictor with one bit of history (taken / not taken) per conditional branch [Agner Fog]
- good: compact storage (only one bit per history entry)

History Buffer Structure (Haswell)

- a predictor with one bit of history (taken / not taken) per conditional branch [Agner Fog]
- good: compact storage (only one bit per history entry)
- mismatch: Haswell doesn't seem to store not-taken branches at all

History Buffer Structure (Haswell)

- a predictor with one bit of history (taken / not taken) per conditional branch [Agner Fog]
- good: compact storage (only one bit per history entry)
- mismatch: Haswell doesn't seem to store not-taken branches at all
    - must still be able to differentiate between "taken, not taken" and "not taken; taken"

History Buffer Structure (Haswell)

- a predictor with one bit of history (taken / not taken) per conditional branch [Agner Fog]
- good: compact storage (only one bit per history entry)
- mismatch: Haswell doesn't seem to store not-taken branches at all
    - must still be able to differentiate between "taken, not taken" and "not taken; taken"
    - address of taken branch is probably used

History Buffer Structure (Haswell)

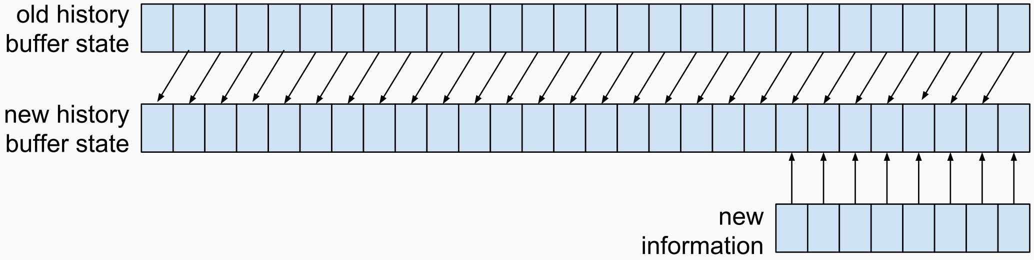
- a predictor with one bit of history (taken / not taken) per conditional branch [Agner Fog]
- good: compact storage (only one bit per history entry)
- mismatch: Haswell doesn't seem to store not-taken branches at all
    - must still be able to differentiate between "taken, not taken" and "not taken; taken"
    - address of taken branch is probably used
- mismatch: seems to differentiate between many targets for a single history entry

History Buffer Structure (Haswell)

- a predictor with one bit of history (taken / not taken) per conditional branch [Agner Fog]
- good: compact storage (only one bit per history entry)
- mismatch: Haswell doesn't seem to store not-taken branches at all
    - must still be able to differentiate between "taken, not taken" and "not taken; taken"
    - address of taken branch is probably used
- mismatch: seems to differentiate between many targets for a single history entry
- good: naturally forgets about old branches (shifted out)

History Buffer Structure (Haswell)

- a predictor with one bit of history (taken / not taken) per conditional branch [Agner Fog]
- good: compact storage (only one bit per history entry)
- mismatch: Haswell doesn't seem to store not-taken branches at all
    - must still be able to differentiate between "taken, not taken" and "not taken; taken"
    - address of taken branch is probably used
- mismatch: seems to differentiate between many targets for a single history entry
- good: naturally forgets about old branches (shifted out)



Attacking KVM: Overview

- goal: read from arbitrary host-kernel-virtual addresses
- attacker type: controls guest ring 0; knows precise host kernel build

Attacking KVM: Overview

- goal: read from arbitrary host-kernel-virtual addresses
- attacker type: controls guest ring 0; knows precise host kernel build
- misdirect first indirect call with memory operand after guest exit
    - provides speculative RIP control
    - requires breaking hypervisor code ASLR

Attacking KVM: Overview

- goal: read from arbitrary host-kernel-virtual addresses
- attacker type: controls guest ring 0; knows precise host kernel build
- misdirect first indirect call with memory operand after guest exit
    - provides speculative RIP control
    - requires breaking hypervisor code ASLR
- flush L3 cache line containing memory operand
    - requires L3 eviction sets (for long speculation)
    - requires identifying correct eviction set

Attacking KVM: Overview

- goal: read from arbitrary host-kernel-virtual addresses
- attacker type: controls guest ring 0; knows precise host kernel build
- misdirect first indirect call with memory operand after guest exit
  - provides speculative RIP control
  - requires breaking hypervisor code ASLR
- flush L3 cache line containing memory operand
  - requires L3 eviction sets (for long speculation)
  - requires identifying correct eviction set
- use gadget to call into BPF interpreter
  - requires register control: caller-saved registers stay intact after guest exit
  - requires data at known address: locate host physmap alias of guest memory

Attacking KVM: Overview

- goal: read from arbitrary host-kernel-virtual addresses
- attacker type: controls guest ring 0; knows precise host kernel build
- misdirect first indirect call with memory operand after guest exit
  - provides speculative RIP control
  - requires breaking hypervisor code ASLR
- flush L3 cache line containing memory operand
  - requires L3 eviction sets (for long speculation)
  - requires identifying correct eviction set
- use gadget to call into BPF interpreter
  - requires register control: caller-saved registers stay intact after guest exit
  - requires data at known address: locate host physmap alias of guest memory
- use BPF bytecode to read arbitrary host data and leak it

Attacking KVM: Steps Overview

- leak **host code address bits** from history buffer and branch target buffer (BTB) *[dump_hyper_bhb, hyper_btb_brute]*

Attacking KVM: Steps Overview

- leak **host code address bits** from history buffer and branch target buffer (BTB) *[dump_hyper_bhb, hyper_btb_brute]*
- identify **L3 cache sets** using brute-force timing-based testing of eviction sets *[cacheset_identify]*

Attacking KVM: Steps Overview

- leak **host code address bits** from history buffer and branch target buffer (BTB) *[dump_hyper_bhb, hyper_btb_brute]*
- identify **L3 cache sets** using brute-force timing-based testing of eviction sets *[cacheset_identify]*
- **determine physical address of guest page** using “load from physical address” gadget and timing *[find_phys_mapping_kassist]*

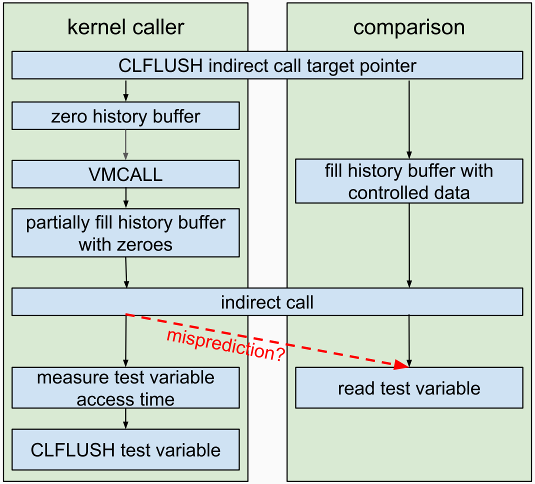
Attacking KVM: Steps Overview

- leak **host code address bits** from history buffer and branch target buffer (BTB) *[dump_hyper_bhb, hyper_btb_brute]*
- identify **L3 cache sets** using brute-force timing-based testing of eviction sets *[cacheset_identify]*
- **determine physical address of guest page** using “load from physical address” gadget and timing *[find_phys_mapping_kassist]*
- determine **address of physmap region** using memory load gadget and timing *[find_page_offset]*

Attacking KVM: Steps Overview

- leak **host code address bits** from history buffer and branch target buffer (BTB) *[dump_hyper_bhb, hyper_btb_brute]*
- identify **L3 cache sets** using brute-force timing-based testing of eviction sets *[cacheset_identify]*
- **determine physical address of guest page** using “load from physical address” gadget and timing *[find_phys_mapping_kassist]*
- determine **address of physmap region** using memory load gadget and timing *[find_page_offset]*
- select **L3 set containing the legitimate indirect call target** using brute force *[select_set]*

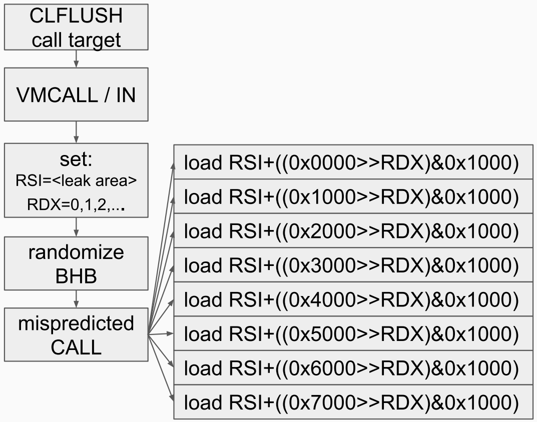
Leaking host address bits (BHB)


kernel caller
comparison
CLFLUSH indirect call target pointer
zero history buffer
VMCALL
partially fill history buffer with zeroes
fill history buffer with controlled data
indirect call
misprediction?
measure test variable access time
read test variable
CLFLUSH test variable

approach: dump history buffer contents

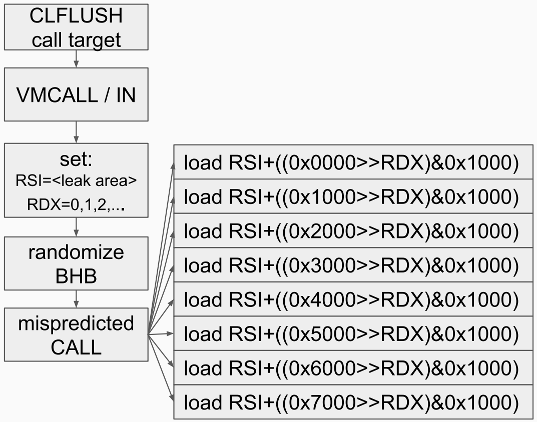
- fill history buffer with state from VMCALL
- shift out some of VMCALL state by padding history buffer with zeroes; leaving 2 bits of unknown information
- compare history buffer against controlled history buffer using misprediction

Leaking host address bits (BTB)


CLFLUSH call target
VMCALL / IN
set: RSI=<leak area> RDX=0,1,2,...
randomize BHB
mispredicted CALL
load RSI+((0x0000>>RDX)&0x1000)
load RSI+((0x1000>>RDX)&0x1000)
load RSI+((0x2000>>RDX)&0x1000)
load RSI+((0x3000>>RDX)&0x1000)
load RSI+((0x4000>>RDX)&0x1000)
load RSI+((0x5000>>RDX)&0x1000)
load RSI+((0x6000>>RDX)&0x1000)
load RSI+((0x7000>>RDX)&0x1000)

approach: execute an indirect call and observe where the CPU jumps

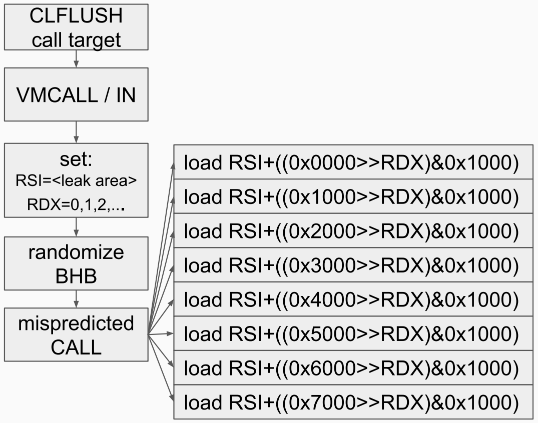
Leaking host address bits (BTB)


CLFLUSH call target
VMCALL / IN
set: RSI=<leak area> RDX=0,1,2,...
randomize BHB
mispredicted CALL
load RSI+((0x0000>>RDX)&0x1000)
load RSI+((0x1000>>RDX)&0x1000)
load RSI+((0x2000>>RDX)&0x1000)
load RSI+((0x3000>>RDX)&0x1000)
load RSI+((0x4000>>RDX)&0x1000)
load RSI+((0x5000>>RDX)&0x1000)
load RSI+((0x6000>>RDX)&0x1000)
load RSI+((0x7000>>RDX)&0x1000)

approach: execute an indirect call and observe where the CPU jumps

- perform VM exit (VMCALL / IN) to fill BTB with host jump addresses
- randomize history buffer to force predictor fallback
- execute CALL with mispredicted target

Leaking host address bits (BTB)


CLFLUSH call target
VMCALL / IN
set: RSI=<leak area> RDX=0,1,2,...
randomize BHB
mispredicted CALL
load RSI+((0x0000>>RDX)&0x1000)
load RSI+((0x1000>>RDX)&0x1000)
load RSI+((0x2000>>RDX)&0x1000)
load RSI+((0x3000>>RDX)&0x1000)
load RSI+((0x4000>>RDX)&0x1000)
load RSI+((0x5000>>RDX)&0x1000)
load RSI+((0x6000>>RDX)&0x1000)
load RSI+((0x7000>>RDX)&0x1000)

approach: execute an indirect call and observe where the CPU jumps

- perform VM exit (VMCALL / IN) to fill BTB with host jump addresses
- randomize history buffer to force predictor fallback
- execute CALL with mispredicted target
- place cache-signaling gadgets at all possible targets; two possible signals
- perform binary search over call targets

Locate Guest Page in Host Memory

Find host-physical address:

- poison BTB and evict function pointer from L1D+L2 → misspeculated host code
- Use physical-load gadget (see right) to brute-force physical address
  - test guesses with *Flush+Reload*

```
; controlled r8, r9
mov rax,r8
movsxd r15,r9d
; load page_offset_base
mov r8,QWORD PTR [r15*8-0x7e594c40]
lea rdi,[rax+r8*1]
; page_offset_base + phys_addr_guess
mov r12,QWORD PTR [r8+rax*1+0xf8]
```

Find host-virtual address:

- physmap is 1GiB-aligned
- bruteforce physmap base address
- test guesses by attempting to access `page_offset_base + phys_guest_page_address`

Full Attack: Leak Host Memory

1. place Spectre gadget BPF bytecode in guest memory

Full Attack: Leak Host Memory

1. place Spectre gadget BPF bytecode in guest memory
2. “Flush” leak area

Full Attack: Leak Host Memory

1. place Spectre gadget BPF bytecode in guest memory
2. “Flush” leak area
3. evict call target

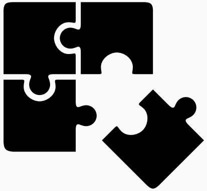
Full Attack: Leak Host Memory

1. place Spectre gadget BPF bytecode in guest memory
2. “Flush” leak area
3. evict call target
4. mistrain branch predictor to BPF interpreter call gadget

Full Attack: Leak Host Memory

1. place Spectre gadget BPF bytecode in guest memory
2. “Flush” leak area
3. evict call target
4. mistrain branch predictor to BPF interpreter call gadget
5. execute VMCALL

Full Attack: Leak Host Memory



1. place Spectre gadget BPF bytecode in guest memory
2. “Flush” leak area
3. evict call target
4. mistrain branch predictor to BPF interpreter call gadget
5. execute VMCALL
6. “Reload” leak area $\rightarrow$ obtain value

Defenses

Mitigating Spectre

- Trivial approach: disable speculative execution

Mitigating Spectre

- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation

Mitigating Spectre

- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation
- Problem: massive performance hit!

Mitigating Spectre

- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation
- Problem: massive performance hit!
- Also: How to disable it?

Mitigating Spectre

- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation
- Problem: massive performance hit!
- Also: How to disable it?
- Speculative execution is deeply integrated into CPU

Spectre Variant 1 Mitigations

Spectre Variant 1 Mitigations

- Workaround: insert instructions stopping speculation

Spectre Variant 1 Mitigations

- Workaround: insert instructions stopping speculation

→ insert after every bounds check

Spectre Variant 1 Mitigations

- Workaround: insert instructions stopping speculation

$\rightarrow$ insert after every bounds check

- x86: `LFENCE`, ARM: `CSDB`

Spectre Variant 1 Mitigations

- Workaround: insert instructions stopping speculation

→ insert after every bounds check

- x86: `LFENCE`, ARM: `CSDB`
- Available on all Intel CPUs, retrofitted to existing ARMv7 and ARMv8

Spectre Variant 1 Mitigations

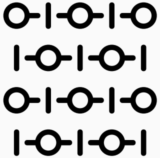
```
// Unprotected

int array[N];

int get_value(unsigned int n) {
  int tmp;

  if (n < N) {
    tmp = array[n]
  } else {
    tmp = FAIL;
  }

  return tmp;
}
```

Spectre Variant 1 Mitigations

```
// Unprotected

int array[N];

int get_value(unsigned int n) {
  int tmp;

  if (n < N) {
    tmp = array[n]
  } else {
    tmp = FAIL;
  }

  return tmp;
}
```

```
// Protected

int array[N];

int get_value(unsigned int n) {

  int *lower = array;
  int *ptr = array + n;
  int *upper = array + N;

  return
    __builtin_load_no_speculate
    (ptr, lower, upper, FAIL);

}
```

Spectre Variant 2 Mitigations (Microcode/MSRs)



Intel released microcode updates

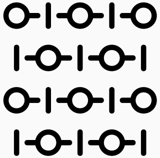
- Indirect Branch Restricted Speculation (IBRS):

Spectre Variant 2 Mitigations (Microcode/MSRs)

Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
    - Prevents branches at lower privilege level from influencing branches at higher privilege level

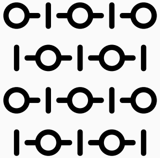
Spectre Variant 2 Mitigations (Microcode/MSRs)



Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
    - Prevents branches at lower privilege level from influencing branches at higher privilege level
    - Must be re-enabled on every switch to higher privileges

Spectre Variant 2 Mitigations (Microcode/MSRs)



Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
  - Prevents branches at lower privilege level from influencing branches at higher privilege level
  - Must be re-enabled on every switch to higher privileges
- Indirect Branch Predictor Barrier (IBPB):

Spectre Variant 2 Mitigations (Microcode/MSRs)

Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
    - Prevents branches at lower privilege level from influencing branches at higher privilege level
    - Must be re-enabled on every switch to higher privileges
- Indirect Branch Predictor Barrier (IBPB):
    - Flush branch-target buffer

Spectre Variant 2 Mitigations (Microcode/MSRs)

Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
  - Prevents branches at lower privilege level from influencing branches at higher privilege level
  - Must be re-enabled on every switch to higher privileges
- Indirect Branch Predictor Barrier (IBPB):
  - Flush branch-target buffer
- Single Thread Indirect Branch Predictors (STIBP):

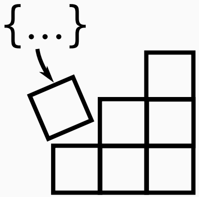
Spectre Variant 2 Mitigations (Microcode/MSRs)

Intel released microcode updates

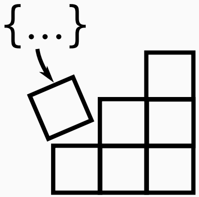
- Indirect Branch Restricted Speculation (IBRS):
  - Prevents branches at lower privilege level from influencing branches at higher privilege level
  - Must be re-enabled on every switch to higher privileges
- Indirect Branch Predictor Barrier (IBPB):
  - Flush branch-target buffer
- Single Thread Indirect Branch Predictors (STIBP):
  - Isolates branch prediction state between two hyperthreads

Spectre Variant 2 Mitigations (Software)

Retpoline (compiler extension)



Spectre Variant 2 Mitigations (Software)

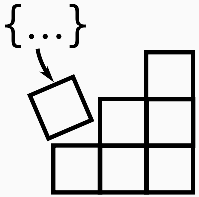


Retpoline (compiler extension)

```
        push <call_target>
        call 1f
    2:                          ; speculation will continue here
        lfence                  ; speculation barrier
        jmp 2b                  ; endless loop
    1:
        lea 8(%rsp), %rsp       ; restore stack pointer
        ret                     ; the actual call to <call_target>
```

→ always predict to enter an endless loop

Spectre Variant 2 Mitigations (Software)



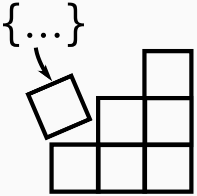
Retpoline (compiler extension)

```
    push <call_target>
    call 1f
  2:                       ; speculation will continue here
    lfence                 ; speculation barrier
    jmp 2b                 ; endless loop
  1:
    lea 8(%rsp), %rsp      ; restore stack pointer
    ret                    ; the actual call to <call_target>
```

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function

Spectre Variant 2 Mitigations (Software)



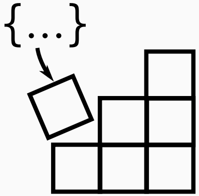
Retpoline (compiler extension)

```
      push <call_target>
      call 1f
    2:                        ; speculation will continue here
      lfence                  ; speculation barrier
      jmp 2b                  ; endless loop
    1:
      lea 8(%rsp), %rsp       ; restore stack pointer
      ret                     ; the actual call to <call_target>
```

$\rightarrow$ always predict to enter an endless loop

- instead of the correct (or wrong) target function $\rightarrow$ performance?

Spectre Variant 2 Mitigations (Software)



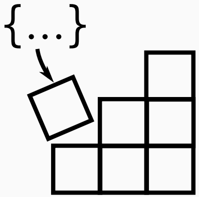
Retpoline (compiler extension)

```
        push <call_target>
        call 1f
    2:                        ; speculation will continue here
        lfence                ; speculation barrier
        jmp 2b                ; endless loop
    1:
        lea 8(%rsp), %rsp     ; restore stack pointer
        ret                   ; the actual call to <call_target>
```

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?
- On Skylake or newer:

Spectre Variant 2 Mitigations (Software)



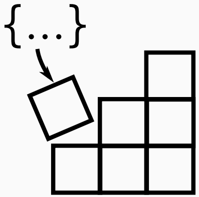
Retpoline (compiler extension)

```
    push <call_target>
    call 1f
  2:                        ; speculation will continue here
    lfence                  ; speculation barrier
    jmp 2b                  ; endless loop
  1:
    lea 8(%rsp), %rsp       ; restore stack pointer
    ret                     ; the actual call to <call_target>
```

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?
- On Skylake or newer:
  - **ret** may fall-back to the BTB for prediction

Spectre Variant 2 Mitigations (Software)



Retpoline (compiler extension)

```
    push <call_target>
    call 1f
  2:                    ; speculation will continue here
    lfence              ; speculation barrier
    jmp 2b              ; endless loop
  1:
    lea 8(%rsp), %rsp   ; restore stack pointer
    ret                 ; the actual call to <call_target>
```

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?
- On Skylake or newer:
    - **ret** may fall-back to the BTB for prediction

  → microcode patches to prevent that

What does not work

- Prevent access to high-resolution timer

What does not work

- Prevent access to high-resolution timer

→ Own timer using timing thread

What does not work

- Prevent access to high-resolution timer

→ Own timer using timing thread

- Flush instruction only privileged

What does not work

- Prevent access to high-resolution timer

$\rightarrow$ Own timer using timing thread

- Flush instruction only privileged

$\rightarrow$ Cache eviction through memory accesses

What does not work

- Prevent access to high-resolution timer

→ Own timer using timing thread

- Flush instruction only privileged

→ Cache eviction through memory accesses

- Just move secrets into secure world

What does not work

- Prevent access to high-resolution timer

→ Own timer using timing thread

- Flush instruction only privileged

→ Cache eviction through memory accesses

- Just move secrets into secure world

→ Spectre works on secure enclaves

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto $\rightarrow$ “software should be fixed”

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto $\rightarrow$ “software should be fixed”
- attacks on ASLR

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto $\rightarrow$ “software should be fixed”
- attacks on ASLR $\rightarrow$ “ASLR is broken anyway”

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto $\rightarrow$ “software should be fixed”
- attacks on ASLR $\rightarrow$ “ASLR is broken anyway”
- attacks on SGX and TrustZone

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
- attacks on SGX and TrustZone → “not part of the threat model”

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → "software should be fixed"
- attacks on ASLR → "ASLR is broken anyway"
- attacks on SGX and TrustZone → "not part of the threat model"
- Rowhammer attacks

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
- attacks on SGX and TrustZone → “not part of the threat model”
- Rowhammer attacks → “only affects cheap sub-standard modules”

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
- attacks on SGX and TrustZone → “not part of the threat model”
- Rowhammer attacks → “only affects cheap sub-standard modules”

→ for years we solely optimized for performance

When you read the manuals...

After learning about an exploitable microarchitectural behavior you realize:

When you read the manuals...

After learning about an exploitable microarchitectural behavior you realize:

- it was documented in the Intel manual

When you read the manuals...

After learning about an exploitable microarchitectural behavior you realize:

- it was documented in the Intel manual
- only now we understand the implications

Conclusions

- sometimes you can't see the wood for the trees: everything was documented

Conclusions

- sometimes you can't see the wood for the trees: everything was documented
- optimizations often have security implications

Conclusions

- sometimes you can't see the wood for the trees: everything was documented
- optimizations often have security implications
- dedicate more time into identifying problems and not solely in mitigating known problems

The Story of Meltdown and Spectre

Jann Horn & Daniel Gruss

May 17, 2018