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
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: +55 12345
printf("\%d", i);
printf("\%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
CPU Cache

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

Cache miss

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

printf("%d", i);

Cache miss Request
Response

i

printf("%d", i);

Cache hit
DRAM access,
slow

printf("%d", i);

Request
Response

Cache miss

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

Cache miss

DRAM access,
slow

Cache hit

No DRAM access,
much faster

printf("%d", i);

Request

Response

CPU Cache
Flush+Reload

ATTACKER

flush
access

Shared Memory

VICTIM

access
Flush+Reload

ATTACKER

VICTIM

Shared Memory

flush
access
cached
cached
access
cached
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
Flush+Reload

ATTACKER

flush
access

Shared Memory

VICTIM

access
Flush+Reload

ATTACKER

flush
access

Shared Memory

VICTIM

access
Flush+Reload

ATTACKER

flush
access

fast if victim accessed data, slow otherwise

Shared Memory

ACCESS

VICTIM

access
Memory Access Latency

![Cache Hits](graph.png)

- **Latency [Cycles]**
- **Number of Accesses**

- Cache Hits
Out-of-order Execution
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
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);
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);
1 char data = *(char*)0xfffffffff81a000e0;
2 printf("%c\n", data);
Building Meltdown

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

```plaintext
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*)0xfffffffffff81a000e0;
printf("%c\n", data);
```

```
segfault at fffffff81a000e0 ip 0000000000400535
sp 00007ffce4a80610 error 5 in reader
```

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

```c
1  *(volatile char*)0;
2  array[84 * 4096] = 0;  // unreachable
```
• Adapted code

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

• Static code analyzer is not happy

1  warning: Dereference of null pointer
2  *(volatile char*)0;
• 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
• Combine the two things

1 char data = *(char*)0xffffffff81a000e0;
2 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
Leaking Passwords from your Password Manager

![Saved Logins](image)

- **https://accounts.google.com**
  - Username: secretpw0
  - Password: [Redacted]
  - Date: 28. Dez. 2017

- **https://signn.ebay.com**
  - Username: Meltdown@gmail.com
  - Password: [Redacted]
  - Date: 28. Dez. 2017

- **https://www.amaz.com**
  - Username: Meltdown@gmail.com
  - Password: [Redacted]
  - Date: 28. Dez. 2017

- **https://www.facebook.com**
  - Username: fb1234!
  - Password: [Redacted]
  - Date: 28. Dez. 2017

- **https://instag.com**
  - Username: Meltdown@gmail.com
  - Password: [Redacted]
  - Date: 28. Dez. 2017

---

- **Login Sites**
  - [https://accounts.google.com](https://accounts.google.com)
  - [https://signn.ebay.com](https://signn.ebay.com)
  - [https://www.amaz.com](https://www.amaz.com)
  - [https://www.facebook.com](https://www.facebook.com)
  - [https://instag.com](https://instag.com)

---

- **Remove Remove Remove All**
- **Remove**
- **Hide Passwords**
- **Close**
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

User memory  Not mapped

context switch

SMAP + SMEP  Kernel memory

Interrupt dispatcher

addr. space

addr. space

Interrupt dispatcher

addr. space
• We published KAISER in May 2017
• We published KAISER in May 2017
• Intel and others improved and merged it into Linux as KPTI (Kernel Page Table Isolation)
• 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
• 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”
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

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

if <access in bounds>
Branch Prediction

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

if <access in bounds>

- processor predicts outcomes of branches
- predictions are based on previous behavior
- predictions help with executing more things in parallel
index = 0;

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)

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

char* data = "textKEY";

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

char* data = "textKEY";

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

char* data = "textKEY";

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

char* data = "textKEY";

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

0
index = 1;

char* data = "textKEY";

if (index < 4)
    Speculate
else
    Prediction

LUT[data[index] * 4096]

0
Spectre Variant 1

```c
index = 1;
char* data = "textKEY";
if (index < 4)
    LUT[data[index] * 4096]
else
    Prediction
    0
```
index = 2;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]
Spectre Variant 1

```c
index = 2;

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

Prediction

0
index = 2;

char* data = "textKEY";

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

0
index = 3;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]
index = 3;

char* data = "textKEY";

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

char* data = "textKEY";

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

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)
    Speculate
    then
        Prediction
        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
{
    Prediction
    LUT[0]
}
index = 5;

char* data = "textKEY";

if (index < 4)

then

Prediction

LUT[data[index] * 4096]

else

0
index = 5;

char* data = "textKEY";

if (index < 4)
    Speculate
    then
        LUT[data[index] * 4096]
    Prediction
    else
        0
Spectre Variant 1

```c
index = 5;
char* data = "textKEY";

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

Execute

```
0
```
Spectre Variant 1

```
index = 6;

char* data = "textKEY";

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

index = 6;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

else

Prediction

0
index = 6;
char* data = "textKEY";

if (index < 4)
    LUT[data[index] * 4096]
else
    0
index = 6;

char* data = "textKEY";

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

else
Prediction

Execute

0
Branch Prediction: Other Patterns (Untested)

- type check

```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();
}
```
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

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

```c
1  kvm_x86_ops->handle_external_intr(vcpu);
2
3 struct  kvm_x86_ops *kvm_x86_ops;
4
5 static struct  kvm_x86_ops vmx_x86_ops = {
6  [...]
7    .handle_external_intr =
8      vmx_handle_external_intr,
9  [...]
10 };

(code simplified)
```
Spectre Variant 2: Indirect Branches

- instruction stream does not contain target address

```c
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

```c
kvm_x86_ops->handle_external_intr(vcpu);

struct kvm_x86_ops *kvm_x86_ops;

static struct kvm_x86_ops vmx_x86_ops = {
    handle_external Infragistics =
        vmx_handle_external Infragistics,
    ...
};
```

(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

```
1  kvm_x86_ops->handle_external_intr(vcpu);
2
3 struct  kvm_x86_ops *kvm_x86_ops;
4
5 static struct  kvm_x86_ops vmx_x86_ops = {
6     [...]
7     .handle_external_intr =
8     vmx_handle_external_intr,
9     [...]
10    };
11
(CODE SIMPLIFIED)
```
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)
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?

Inject misspeculation to controlled addresses across security domains

Attack goal: Leak host memory from inside a KVM guest
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:

- 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"
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

- **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"
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

- 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

(processes 1 and 2 are connected with arrows showing the flow of execution)

- CLFLUSH indirect call target pointer
- series of N taken conditional branches
- indirect call
- measure test variable access time
- CLFLUSH test variable

- read test variable

(explicit execution barriers omitted from diagram)
Predictor Reversing: Minimal Test

- hyperthreaded

CLFLUSH indirect call target pointer

series of N taken conditional branches

indirect call

measure test variable access time

CLFLUSH test variable

read test variable

(explicit execution barriers omitted from diagram)
Predictor Reversing: Minimal Test

- hyperthreaded
- same code

process 1

CLFLUSH indirect call target pointer

series of N taken conditional branches

indirect call

measure test variable access time

CLFLUSH test variable

(explicit execution barriers omitted from diagram)
**Predictor Reversing: Minimal Test**

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

(process 1)

- CLFLUSH indirect call target pointer
- series of N taken conditional branches
- indirect call
- measure test variable access time
- CLFLUSH test variable

(process 2)

- read test variable

(misprediction)

(explicit execution barriers omitted from diagram)
Predictor Reversing: Minimal Test

- 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
(explicit execution barriers omitted from diagram)
Predictor Reversing: Minimal Test

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

(explicit execution barriers omitted from diagram)
Predictor Reversing: Minimal Test

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

(explicit execution barriers omitted from diagram)
Variant 2: first brittle PoC [in initial writeup]

- shortcuts for minimal PoC

... direct branches only
collide low 31 bits of source address, assume relative target
→ leak rate: \( \approx 6 \text{ bits/second} \)
→ 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
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

... direct branches only

\[ \text{leak rate: } \approx 6 \text{ bits/second} \]

- 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
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

为目标地址的低31位直接分支，假设相对目标。

→ 泄漏率：≈ 6 bits/second
→ 几乎所有注入尝试都失败！

CPU 区分了注入和虚拟机执行。

→ 理论：
  - 注入仅适用于“单调目标”预测
  - CPU 倾向于基于历史的预测
  - 注入在基于历史的预测因系统噪声导致的逐出时有效
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

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
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: \( \approx 6 \text{ bits/second} \)
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
  → 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: \( \approx 6 \text{ bits/second} \) — 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: $\approx 6$ 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: \( \approx 6 \text{ bits/second} \) — 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?

"monotonic target" prediction
- uses branch source address for lookup

*injection seems to work, but not usually used*

*reverse this sufficiently for injections?*
Predictor Reversing: History Length

- ≈ 26 branches stored
- Measurements get weird around the boundary [and are not yet entirely correct]
Predictor Reversing: History Length

- ≈ 26 branches stored
- measurements get weird around the boundary [and are not yet entirely correct]
Predictor Reversing: Branch Types

- taken conditional branch
- not-taken conditional branch
- unconditional direct jump
- unconditional indirect branch
- RET
- IRETQ
Predictor Reversing: Branch Types

- taken conditional branch
- not-taken conditional branch
- unconditional direct jump
- unconditional indirect branch
- RET
- IRETQ

on Haswell:

- 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
  - measure test variable access time
  - CLFLUSH test variable
  - read test variable
  - misprediction?
Predictor Reversing: Branch Types

on Haswell:

- taken conditional branch

Diagram:

- process 1
  - 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
  - measure test variable access time
  - CLFLUSH test variable

- process 2
  - read test variable

Red line indicates misprediction?
Predictor Reversing: Branch Types

- taken conditional branch
- not-taken conditional branch

on Haswell:
- taken conditional branch ✓
- not-taken conditional branch ✗
Predictor Reversing: Branch Types

on Haswell:
- taken conditional branch ✓
- not-taken conditional branch ☒
- unconditional direct jump ✓
Predictor Reversing: Branch Types

on Haswell:
- taken conditional branch ✓
- not-taken conditional branch ✗
- unconditional direct jump ✓
- unconditional indirect branch ✓
Predictor Reversing: Branch Types

on Haswell:

- taken conditional branch ✓
- not-taken conditional branch ✗
- unconditional direct jump ✓
- unconditional indirect branch ✓
- RET ✓
Predictor Reversing: Branch Types

on Haswell:
- taken conditional branch ✓
- not-taken conditional branch ❌
- unconditional direct jump ✓
- unconditional indirect branch ✓
- RET ✓
- IRETQ ❌
Address Bits in History

process 1

CLFLUSH indirect call target pointer

series of N taken conditional branches (normalize)

indirect call to one of two targets

RET (address A)

indirect call

measure test variable access time

CLFLUSH test variable

misprediction?

process 2

RET (address B)

read test variable
Address Bits in History

→ only low 20 bits of any address affect history
Predictor Reversing: Branch Type influence?

process 1

- CLFLUSH indirect call target pointer
- series of N taken conditional branches (normalize)
- indirect call
- taken cond branch
- RET

process 2

- other branch type
- RET
Full History Control

- kinda like ROP

<table>
<thead>
<tr>
<th>IRETQ frame</th>
<th>RET frame</th>
<th>IRETQ frame</th>
<th>...</th>
<th>RET frame</th>
<th>IRETQ frame</th>
<th>RET frame</th>
<th>IRETQ frame</th>
</tr>
</thead>
</table>

creates one history entry

pivot stack to here; execute 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

IRETQ frame
---
RET frame
IRETQ frame
...
RET frame
IRETQ frame
RET frame
IRETQ frame

creates one history entry
pivot stack to here; execute 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

choice of instruction determines RET source

$2^{20}$ bytes of RETs

IRETQ

RET

...
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) \([\text{dump\_hyper\_bhb, hyper\_btb\_brute}]\)
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) \([\text{dump_hyper_bhb, hyper_btb_brute}]\)

• identify **L3 cache sets** using brute-force timing-based testing of eviction sets \([\text{cacheset_identify}]\)

• determine **physical address of guest page** using “load from physical address” gadget and timing \([\text{find_phys_mapping_kassist}]\)
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 phymap 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) \([\text{dump_hyper_bhb, hyper_btb_brute}]\)
- identify **L3 cache sets** using brute-force timing-based testing of eviction sets \([\text{cacheset_identify}]\)
- determine **physical address of guest page** using “load from physical address” gadget and timing \([\text{find_phys_mapping_kassist}]\)
- determine **address of physmap region** using memory load gadget and timing \([\text{find_page_offset}]\)
- select **L3 set containing the legitimate indirect call target** using brute force \([\text{select_set}]\)
Leaking host address bits (BHB)

- 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)

- 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

Approach: execute an indirect call and observe where the CPU jumps
Leaking host address bits (BTB)

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)

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 \textit{Flush+Reload}

```assembly
1 ; controlled r8, r9
2 mov rax,r8
3 movsx r15,r9d
4 ; load page_offset_base
5 mov r8,QWORD PTR [r15*8-0x7e594c40]
6 lea rdi,[rax+r8*1]
7 ; page_offset_base + phys_addr_guess
8 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
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
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 → 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

- 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

- Workaround: insert instructions stopping speculation
  - x86: LFENCE
  - ARM: CSDB

Available on all Intel CPUs, retrofitted to existing ARMv7 and ARMv8
Spectre Variant 1 Mitigations

- Workaround: insert instructions stopping speculation
  → insert after every bounds check
Spectre Variant 1 Mitigations

- Workaround: insert instructions stopping speculation
  - 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
// 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);
}
Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
Intel released microcode updates

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

```assembly
1  push call_target
2  call 1f
3  2: ; speculation will continue here
4  lfence ; speculation barrier
5  jmp 2b ; endless loop
6  1:
7  lea 8(% rsp ), % rsp ; restore stack pointer
8  ret ; the actual call to < call_target >

→ always predict to enter an endless loop
→ instead of the correct (or wrong) target function

• On Skylake or newer:
• ret may fall-back to the BTB for prediction
→ microcode patches to prevent that
```
Retpoline (compiler extension)

```assembly
1       push <call_target>
2       call 1f
3           2: ; speculation will continue here
4       lfence ; speculation barrier
5       jmp 2b ; endless loop
6       1:
7       lea 8(%rsp), %rsp ; restore stack pointer
8       ret ; the actual call to <call_target>
```

→ always predict to enter an endless loop
Retpoline (compiler extension)

```
1    push <call_target>
2    call 1f
3     2:    ; speculation will continue here
4     lfence    ; speculation barrier
5     jmp 2b    ; endless loop
6     1:    
7     lea 8(%rsp), %rsp ; restore stack pointer
8     ret    ; the actual call to <call_target>
```

→ always predict to enter an endless loop

• instead of the correct (or wrong) target function
Retpoline (compiler extension)

1. push <call_target>
2. call 1f
3. 2: ; speculation will continue here
4. lfence ; speculation barrier
5. jmp 2b ; endless loop
6. 1:
7. lea 8(%rsp), %rsp ; restore stack pointer
8. ret ; the actual call to <call_target>

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?
Spectre Variant 2 Mitigations (Software)

Retpoline (compiler extension)

```assembly
1 push <call_target>
2 call 1f
3  
4    2: ; speculation will continue here
5    lfence ; speculation barrier
6    jmp 2b ; endless loop
7  
8 1:    
9    lea 8(%rsp), %rsp ; restore stack pointer
10    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:
Retpoline (compiler extension)

```
1    push <call_target>
2    call 1f
3    2:    ; speculation will continue here
4    lfence    ; speculation barrier
5    jmp 2b    ; endless loop
6    1:
7    lea 8(%rsp), %rsp    ; restore stack pointer
8    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)

```assembly
1 push  <call_target>
2 call  1f
3 2: ; speculation will continue here
4 lfence ; speculation barrier
5 jmp  2b ; endless loop
6 1:
7 lea  8(%rsp), %rsp ; restore stack pointer
8 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
  → Own timer using timing thread
- Flush instruction only privileged
  → 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:

- 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
What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
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
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”
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
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:

- it was documented in the Intel manual
- only now we understand the implications
When you read the manuals...

After learning about an exploitable microarchitectural behavior you realize:

- it was documented in the Intel manual
After learning about an exploitable microarchitectural behavior you realize:

- it was documented in the Intel manual
- only now we understand the implications
• sometimes you can’t see the wood for the trees: everything was documented
• 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