Spectre Returns! Speculation Attacks Using Return Stack Buffer

Esmaeil Mohammadian, Khaled N. Khasawneh, Chengyue Song and Nael Abu-Ghazaleh

University of California, Riverside
New vulnerabilities in modern processors

Spectre Attacks: Exploiting Speculative Execution

Paul Kocher\textsuperscript{1}, Jana Horn\textsuperscript{2}, Anders Fagh\textsuperscript{3}, Daniel Genkin\textsuperscript{4}, Daniel Gruss\textsuperscript{5}, Werner Haas\textsuperscript{6}, Mike Hamburg\textsuperscript{7}, Moritz Lipp\textsuperscript{8}, Stefan Mangard\textsuperscript{6}, Thomas Prescher\textsuperscript{6}, Michael Schwarz\textsuperscript{6}, Yuvan Yarom\textsuperscript{6}

\textsuperscript{1} Independent (www.paulkocher.com), \textsuperscript{2} Google Project Zero, \textsuperscript{3} G DATA Advanced Analytics, \textsuperscript{4} University of Pennsylvania and University of Maryland, \textsuperscript{5} Graz University of Technology, \textsuperscript{6} Cyberus Technology, \textsuperscript{7} Rambus, Cryptography Research Division, \textsuperscript{8} University of Adelaide and Data61

Moritz Lipp\textsuperscript{1}, Michael Schwarz\textsuperscript{1}, Daniel Gruss\textsuperscript{5}, Werner Haas\textsuperscript{6}, Mike Hamburg\textsuperscript{7}, Moritz Lipp\textsuperscript{8}, Stefan Mangard\textsuperscript{6}, Thomas Prescher\textsuperscript{6}, Michael Schwarz\textsuperscript{6}, Yuvan Yarom\textsuperscript{6}

Graz University, Graz University of Technology, Independent

Spectre v1/v2/Meltdown(v3)

Jan 2018
New vulnerabilities in modern processors

Spectre v1/v2/Meltdown(v3)

Jan 2018

Speculative store bypass (v4)

May 2018

Spectre Attacks: Exploiting Speculative Execution

Paul Kocher\textsuperscript{1}, Jana Hron\textsuperscript{2}, Anders Foug\textsuperscript{3}, Daniel Genkin\textsuperscript{4},
Daniel Gruss\textsuperscript{5}, Werner Haas\textsuperscript{6}, Mike Hamburg\textsuperscript{7}, Moritz Lipp\textsuperscript{5},
Stefan Mangard\textsuperscript{6}, Thomas Prescher\textsuperscript{6}, Michael Schwarz\textsuperscript{5}, Yuval Yarom\textsuperscript{6}

\textsuperscript{1} Independent (www.paulkocher.com), \textsuperscript{2} Google Project Zero,
\textsuperscript{3} G DATA Advanced Analytics, \textsuperscript{4} University of Pennsylvania and University of Maryland,
\textsuperscript{5} Graz University of Technology, \textsuperscript{6} Cyberus Technology,
\textsuperscript{7} Rambus, Cryptography Research Division, \textsuperscript{8} University of Adelaide and Data61
New vulnerabilities in modern processors:

- Spectre v1/v2/Meltdown (v3)
  - Jan 2018

- Spectre Returns! Speculation Attacks using the Return Stack Buffer
  - Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song and Nael Abu-Ghazaleh
  - Computer Science and Engineering Department
  - University of California, Riverside
  - naelag@ucr.edu

- Speculative store bypass (v4)
  - May 2018

- SpectreRSB (v5?) / ret2spec
  - July 2018
New vulnerabilities in modern processors

Spectre Returns! Speculation Attacks using the Return Stack Buffer

Esmail Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song and Nael Abu-Ghazaleh
Computer Science and Engineering Department
University of California, Riverside
naelag@ucr.edu

Jan 2018

Spectre v1/v2/Meltdown(v3)
Spectre v1.1
Spectre v1.2
SGXpectre
Speculative store bypass (v4)
SpectreNG
SpectreRSB(v5?) / ret2spec

May 2018

July 2018
New vulnerabilities in modern processors

Spectre
- Spectre v1/v2/Meltdown(v3)
- Spectre v1.1
- Spectre v1.2
- SGXspectre
- Speculative store bypass (v4)
- SpectreNG
- SpectreRSB (v5?) / ret2spec
- NetSpectre

NetSpectre: Read Arbitrary Memory over Network

Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh,
Chengyu Song and Nael Abu-Ghazaleh
Computer Science and Engineering Department
University of California, Riverside
naelag@ucr.edu

Michael Schwarz
Graz University of Technology

Moritz Lipp
Graz University of Technology

Martin Schwarzl
Graz University of Technology

Daniel Gruss
Graz University of Technology
Main components of the Attack

Out of Order Execution

Side channel Attack

- [Diagram of Out of Order Execution]
- [Diagram of Side channel Attack]
Out of Order Execution (OoO)

- Speculation is critical to modern CPU performance
(OoO): Branch predictors

- During speculation processors guess the future stream instructions of the program

- Better prediction improve the performance by increasing number of the committed instruction
Branch predictors
Branch predictors

- Two hardware predictors:
Branch predictors

• Two hardware predictors:

  • **Direction predictor** guesses if branch is taken or not-taken (PHT)
Branch predictors

- Two hardware predictors:
  - **Direction predictor** guesses if branch is taken or not-taken (PHT)
  - **Target predictor** guesses the target of the branches (BTB)
Cache Side channel Attacks

• Access to the data inside the cache is fast

• Loading data from memory is too slow
Cache Side channel Attacks

• Access to the data inside the cache is fast

• Loading data from memory is too slow

• Exploits timing differences that are introduced by the caches
  • Flush and reload
  • Prime and probe
  • ...
Side channel: Flush+Reload Attack
Side channel: Flush+Reload Attack

1- **Flush** each line in the critical data
Side channel: Flush+Reload Attack

1- Flush each line in the critical data

2- Victim accesses critical data

- Each cache line is flushed and replaced through the L1-, L1-, and L2- ways.

- The victim accesses critical data, which is evicted from the L1 cache.

- The attacker observes the evicted data, revealing sensitive information.

Diagram:

- CPU1 and CPU2 with L1- and L1- caches.
- Shared L3 cache.
- Cache eviction and replacement process.
Side channel: Flush+Reload Attack

1. **Flush** each line in the critical data

2. Victim accesses critical data

3. **Reload** critical data (measure time)

![Cache Diagram]

- CPU1
  - Victim
  - L1-I
  - L2

- CPU2
  - Attacker
  - L1-I
  - L2

- Shared L3

![Evicted Cache Blocks]

- Set s
- Ways
Side channel: Flush+Reload Attack

1- Flush each line in the critical data

2- Victim accesses critical data

3- Reload critical data (measure time)
Putting it all together– Attacks!
Main idea of all Attacks
Main idea of all Attacks

1. Fool the processor to speculatively execute some instructions such that:
Main idea of all Attacks

1. Fool the processor to speculatively execute some instructions such that:
   • The instructions access sensitive data without permission (microarchitectural state changes)
   • Load the data into the cache
Main idea of all Attacks

1. Fool the processor to speculatively execute some instructions such that:
   • The instructions access sensitive data without permission (microarchitectural state changes)
   • Load the data into the cache

2. Read it from the side channel → broke isolation
   • Microarchitectural changes are not visible directly
Example of attacks
Example of attacks

• Spectre Variant 1:
Example of attacks

• Spectre Variant 1:
  • Train the Direction predictor (PHT) to bypass bound checking and leak sensitive data.
Example of attacks

• Spectre Variant 1:
  • Train the Direction predictor (PHT) to bypass bound checking and leak sensitive data.

• Spectre Variant 2:
Example of attacks

• Spectre Variant 1:
  • Train the Direction predictor (PHT) to bypass bound checking and leak sensitive data.

• Spectre Variant 2:
  • Pollute the target predictor (BTB) by injecting the address of malicious gadget into the BTB
  • Waiting for the victim to execute the malicious gadget speculatively and load sensitive data to the cache
Spectre returns!

Speculation Attacks using the Return Stack Buffer
Why Return Stack Buffer (RSB)?

• BTB can not predict the target of ret instructions properly.
Why Return Stack Buffer (RSB)?

- BTB can not predict the target of ret instructions properly.
Why Return Stack Buffer (RSB)?

- BTB can not predict the target of ret instructions properly.
Why Return Stack Buffer (RSB)?

- BTB can not predict the target of ret instructions properly.
Why Return Stack Buffer (RSB)?

- BTB can not predict the target of ret instructions properly.

```
0x0001 printf: ...
0x0005 ret
A: 0x0010 load
B: 0x0020 call printf load

Branch Target Buffer

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x0005</td>
<td>0x0010</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Why Return Stack Buffer (RSB)?

- BTB can not predict the target of ret instructions properly.
Why Return Stack Buffer (RSB)?

- BTB can not predict the target of ret instructions properly.
Why Return Stack Buffer (RSB)?

- BTB can not predict the target of ret instructions properly.

![Diagram showing the use of RSB with BTB](image)
Why Return Stack Buffer (RSB)?

• BTB can not predict the target of ret instructions properly.
Return Stack Buffer
Return Stack Buffer

- Predict address of `ret` instruction
Return Stack Buffer

- Predict address of *ret* instruction
- RSB is shared between two hardware threads
Return Stack Buffer

- Predict address of \texttt{ret} instruction
- RSB is shared between two hardware threads
- 16 to 24 entries
Return Stack Buffer

- Predict address of `ret` instruction
- RSB is shared between two hardware threads
- 16 to 24 entries
- Push `pc+4` onto the RSB on each `call` instruction
Return Stack Buffer

• Predict address of \textit{ret} instruction
• RSB is shared between two hardware threads
• 16 to 24 entries
• Push \texttt{pc+4} onto the RSB on each \textit{call} instruction
• Pop an address off the RSB on each \textit{ret} instruction
RSB Pollution

- Return Stack Buffer works perfectly for matched call/ret pairs.
RSB Pollution

• Return Stack Buffer works perfectly for matched *call/ret* pairs.

• RSB miss-speculates if return address in the RSB does not match the return address value in the software stack.
How to pollute RSB?
How to pollute RSB?

- S1: Overfill or Underfill of the RSB
- S2: Direct pollution of the RSB
- S3: Speculative pollution of the RSB
- S4: RSB uses across execution contexts
How to pollute RSB?

S1: Overfill or Underfill of the RSB

S2: Direct pollution of the RSB

S3: Speculative pollution of the RSB

S4: RSB uses across execution contexts
Direct pollution of the RSB

- **ret** → pop; jmp address;

- **call** ← push address; ret;
  push address; jmp address;
Direct pollution of the RSB

• **ret** → pop; jmp address;
  • Leave a value on RSB that has been removed from the software stack

• **call**
  push address; ret;

  push address; jmp address;
Direct pollution of the RSB

- **ret** → pop; jmp address;
  - Leave a value on RSB that has been removed from the software stack

- **call**
  - push address; ret;
  - push address; jmp address;
  - A return value exists on the software stack that is not matched by a value in the RSB
RSB use across execution contexts

- On a context switch, the RSB values left over from an executing thread are reused by the next thread.
SpectreRSB

- Attack 1: Same process
- Attack 2: Across threads/process
  - Colluding threads (user)
  - Colluding threads (kernel)
  - Cross-process
- Attack 3: Return in SGX
- Attack 4: Kernel from user
Attack 1: Basic Attack
Attack 1: Basic Attack

• Lunched from a process to part of its own address space
Attack 1: Basic Attack

• Lunched from a process to part of its own address space

• Break Sandbox boundaries
Attack 1: Basic Attack

• Lunched from a process to part of its own address space

• Break Sandbox boundaries
  • Difficult to implement the gadget to manipulate the stack using high level sandboxing primitives
Attack 1: Basic Attack

- Lunched from a process to part of its own address space
- Break Sandbox boundaries
  - Difficult to implement the gadget to manipulate the stack using high level sandboxing primitives
- Enables the attacker to read kernel memory via the Meltdown bug
Attack 1: Basic Attack

• Lunched from a process to part of its own address space

• Break Sandbox boundaries
  • Difficult to implement the gadget to manipulate the stack using high level sandboxing primitives

• Enables the attacker to read kernel memory via the Meltdown bug
  • KPTI prevents it
Attack 1: Basic Attack

0x00000010 pollute:
push %rbp
mov %rsp,%rbp
pop %rdi
pop %rdi
pop %rdi
pop %rdi
pop %rdi
clflush (%rsp)
retq

0x00000019 speculative:
call pollute
movzx (%[array],rbx)

0x00000020 main:
call speculative
rdtscp
access
rdtscp
Attack 1: Basic Attack

<table>
<thead>
<tr>
<th>Address</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000010</td>
<td><strong>pollute:</strong></td>
</tr>
<tr>
<td></td>
<td>push %rbp</td>
</tr>
<tr>
<td></td>
<td>mov %rsp,%rbp</td>
</tr>
<tr>
<td></td>
<td>pop %rdi</td>
</tr>
<tr>
<td></td>
<td>pop %rdi</td>
</tr>
<tr>
<td></td>
<td>pop %rdi</td>
</tr>
<tr>
<td></td>
<td>pop %rdi</td>
</tr>
<tr>
<td></td>
<td>pop %rbp</td>
</tr>
<tr>
<td></td>
<td>clflush (%rsp)</td>
</tr>
<tr>
<td></td>
<td>retq</td>
</tr>
<tr>
<td>0x00000019</td>
<td></td>
</tr>
<tr>
<td>0x00000020</td>
<td><strong>speculative:</strong></td>
</tr>
<tr>
<td>0x00000021</td>
<td>call pollute</td>
</tr>
<tr>
<td>0x00000022</td>
<td>movzx (%[array],rbx)</td>
</tr>
<tr>
<td>0x00000030</td>
<td><strong>main:</strong></td>
</tr>
<tr>
<td>0x00000031</td>
<td>call speculative</td>
</tr>
<tr>
<td>0x00000032</td>
<td>rdtscp</td>
</tr>
<tr>
<td></td>
<td>access</td>
</tr>
<tr>
<td></td>
<td>rdtscp</td>
</tr>
</tbody>
</table>

Software Stack

- 0x00000080 arguments
- 0x00000080

RSB
Attack 1: Basic Attack

0x00000010 pollute:
  push %rbp
  mov %rsp,%rbp
  pop %rdi
  pop %rdi
  pop %rdi
  pop %rdi
  pop %rdi
  pop %rbp
  clflush (%rsp)
  retq

0x00000019 speculative:
  call pollute
  movzx (%[array],rbx)

0x00000020 main:
  call speculative
  rdtscp
  access
  rdtscp

Software Stack

Local variables
  ebp old value
  0x00000080 arguments

RSB
  0x00000080
Attack 1: Basic Attack

0x00000010 pollute:
push %rbp
mov %rsp,%rbp
pop %rdi
pop %rdi
pop %rdi
pop %rdi
cld
retq

0x00000020 speculative:
call pollute
call speculative
movzx (%[array],rbx)

0x00000030 main:
call speculative
rdtscp
access
rdtscp
Attack 1: Basic Attack

```
0x0000010 pollute:
push %rbp
mov %rsp,%rbp
pop %rdi
pop %rdi
pop %rdi
pop %rdi
pop %rdi
clflush (%rsp)
retq
```

```
0x0000020 speculative:
call pollute
movzx (%[array],rbx)
```

```
0x0000030 main:
call speculative
rdtscp
access
rdtscp
```

Software Stack

Local variables
ebp old value
0x00000032
arguments
Local variables
ebp old value
0x00000080
arguments

RSB

0x00000032
0x00000080
Attack 1: Basic Attack

```
0x00000010 pollute:
push %rbp
mov %rsp,%rbp
pop %rdi
pop %rdi
pop %rdi
pop %rdi
pop %rdi
pop %rbp
clflush (%rsp)
retq
```

```
0x00000020 speculative:
call pollute
movzx (%[array],%rbx)
```

```
0x00000030 main:
call speculative
rdtscp
access
rdtscp
```
Attack 1: Basic Attack

0x00000010 pollute:
push %rbp
mov %rsp,%rbp
pop %rdi
pop %rdi
pop %rdi
pop %rdi
pop %rdi
pop %rbp
clflush (%rsp)
retq

speculative:
call pollute
movzx (%[array],rbx)

main:
call speculative
rdtscp
access
rdtscp

Software Stack

ebp old value
0x00000022
arguments
Local variables
0x00000032
arguments
Local variables
ebp old value
0x00000080
arguments

RSB

0x00000022
0x00000032
0x00000080
Attack 1: Basic Attack

```
0x00000010 pollute:
push %rbp
mov %rsp,%rbp
pop %rdi
pop %rdi
pop %rdi
pop %rdi
pop %rdi
clflush (%rsp)
retq
```

```
0x00000020 speculative:
call pollute
movzx (%[array],rbx)
```

```
0x00000030 main:
call speculative
rdtscp
access
rdtscp
```

Software Stack

<table>
<thead>
<tr>
<th>pollute</th>
<th>speculative</th>
<th>main</th>
</tr>
</thead>
<tbody>
<tr>
<td>ebp old value</td>
<td>0x00000022</td>
<td>arguments</td>
</tr>
<tr>
<td>arguments</td>
<td>0x00000032</td>
<td>Local variables</td>
</tr>
<tr>
<td>Local variables</td>
<td>0x00000080</td>
<td>arguments</td>
</tr>
<tr>
<td>ebp old value</td>
<td>0x00000022</td>
<td>arguments</td>
</tr>
<tr>
<td>arguments</td>
<td>0x00000032</td>
<td>Local variables</td>
</tr>
<tr>
<td>Local variables</td>
<td>0x00000080</td>
<td>arguments</td>
</tr>
</tbody>
</table>

RSB

| 0x00000022 |
| 0x00000032 |
| 0x00000080 |
## Attack 1: Basic Attack

**pollute**:  
```c
push %rbp
mov %rsp,%rbp
pop %rdi
pop %rdi
pop %rdi
pop %rdi
pop %rbp
clflush (%rsp)
retq
```

**speculative**:  
```c
call pollute
movz (%[array],rbx)
```

**main**:  
```c
call speculative
rdtscp
access
rdtscp
```

### Software Stack

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>pollute</td>
</tr>
<tr>
<td>0x00000009</td>
<td>speculative</td>
</tr>
<tr>
<td>0x0000000b</td>
<td>main</td>
</tr>
<tr>
<td>0x00000011</td>
<td>pollute</td>
</tr>
<tr>
<td>0x0000001b</td>
<td>speculative</td>
</tr>
<tr>
<td>0x0000001d</td>
<td>main</td>
</tr>
</tbody>
</table>

### Execution Path

- **pollute**: push %rbp, mov %rsp,%rbp, pop %rdi, pop %rdi, pop %rdi, pop %rdi, pop %rbp, clflush (%rsp), retq
- **speculative**: call pollute, movz (%[array],rbx)
- **main**: call speculative, rdtscp, access, rdtscp

### Arguments

- `arguments`
- `ebp old value`
- `local variables`

### RSB

- `0x00000022`
- `0x00000032`
- `0x00000080`
**Attack 1: Basic Attack**

```assembly
pollute:
push %rbp
mov %rsp,%rbp
pop %rdi
pop %rdi
pop %rdi
pop %rdi
pop %rbp
clflush (%rsp)
retq
```

```assembly
speculative:
call pollute
movzx (%[array],rbx)
```

```assembly
main:
call speculative
rdtscp
access
rdtscp
```
Attack 1: Basic Attack

0x00000010 pollute:
  push %rbp
  mov %rsp,%rbp
  pop %rdi
  pop %rdi
  pop %rdi
  pop %rdi
  pop %rbp
  clflush (%rsp)
  retq
  
0x00000019 speculative:
  call pollute
  movzx (%[array],rbx)

0x00000020 main:
  call speculative
  rdtscp
  access
  rdtscp

Software Stack

pollute
speculative
main

arguments
Local variables
ebp old value
Attack 1: Basic Attack

```
0x00000010 pollutec:
push %rbp
mov %rsp,%rbp
pop %rdi
pop %rdi
pop %rdi
pop %rdi
pop %rdi
pop %rbp
clflush (%rsp)
retq
```

```
0x00000020 speculative:
call pollute
movzx (%[array],rbx)
```
Attack 1: Basic Attack

```
0x00000010 pollute:
push %rbp
mov %rsp,%rbp
pop %rdi
pop %rdi
pop %rdi
pop %rdi
pop %rdi
clflush (%rsp)
retq

speculative:
call pollute
movzx (%[array],rbx)
```

```
0x00000020 main:
call speculative
rdtscp
access
rdtscp
```
Attack 1: Basic Attack

```
0x00000010 pollute:
push %rbp
mov %rsp, %rbp
pop %rdi
pop %rdi
pop %rdi
pop %rdi
pop %rbp
clflush (%rsp)
retq
```

```
0x00000021 speculative:
call pollute
movzx [%array], %rbx
```

```
0x00000030 main:
call speculative
rdtscp
access
rdtscp
```

Software Stack

- pollute
- speculative
- main

RSB

- 0x00000022
- 0x00000032 arguments
- 0x00000080 arguments
- ebp old value
- main

Arguments

- old value
- main
## Attack 1: Basic Attack

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000010</td>
<td>pollute:</td>
</tr>
<tr>
<td></td>
<td>push %rbp</td>
</tr>
<tr>
<td></td>
<td>mov %rsp,%rbp</td>
</tr>
<tr>
<td></td>
<td>pop %rdi</td>
</tr>
<tr>
<td></td>
<td>pop %rdi</td>
</tr>
<tr>
<td></td>
<td>pop %rdi</td>
</tr>
<tr>
<td></td>
<td>pop %rdi</td>
</tr>
<tr>
<td></td>
<td>clflush (%rsp)</td>
</tr>
<tr>
<td></td>
<td>retq</td>
</tr>
<tr>
<td>0x00000019</td>
<td></td>
</tr>
<tr>
<td>0x00000020</td>
<td>speculative:</td>
</tr>
<tr>
<td></td>
<td>call pollute</td>
</tr>
<tr>
<td></td>
<td>movzx (%[array],rbx)</td>
</tr>
<tr>
<td>0x00000021</td>
<td></td>
</tr>
<tr>
<td>0x00000022</td>
<td></td>
</tr>
<tr>
<td>0x00000030</td>
<td>main:</td>
</tr>
<tr>
<td></td>
<td>call speculative</td>
</tr>
<tr>
<td></td>
<td>rdtscp</td>
</tr>
<tr>
<td></td>
<td>access</td>
</tr>
<tr>
<td></td>
<td>rdtscp</td>
</tr>
</tbody>
</table>

### Software Stack

- **pollute**
  - 0x00000032 arguments
  - Local variables
  - ebp old value
  - 0x00000080 arguments
- **speculative**
- **main**
- **RSB**
  - 0x00000022
  - 0x00000032
  - 0x00000080
Attack 1: Basic Attack

0x00000010 pollute:
push %rbp
mov %rsp, %rbp
pop %rdi
pop %rdi
pop %rdi
pop %rdi
pop %rdi
clflush (%rsp)
retq

speculative:
call pollute
movzx (%[array], rbx)

main:
call speculative
rdtscp
access
rdtscp

Software Stack
- pollute
- speculative
- main

arguments
- 0x00000032
- 0x00000080
Local variables
ebp old value
- 0x00000032
- 0x00000080

RSB
- 0x00000022
- 0x00000032
- 0x00000080

Defenses

• Microcode patches
  • Lfence
  • IBRS
  • IBPB

• Software patches
  • Retpoline
  • RSBstuffing
Microcode patches
Microcode patches

- LFENCE
  - A barrier after branch instruction to stop speculative execution
Microcode patches

• LFENCE
  • A barrier after branch instruction to stop speculative execution

• Indirect Branch Restricted Speculation (IBRS)
  • Speculation of indirect branches restricted by IBRS
Microcode patches

• LFENCE
  • A barrier after branch instruction to stop speculative execution

• Indirect Branch Restricted Speculation (IBRS)
  • Speculation of indirect branches restricted by IBRS

• Indirect Branch Predictor Barrier (IBPB)
  • To prevent software running before the barrier to affect the indirect branch prediction of software running after the barrier
Software Patch: RSB refilling

- RSB underfill (Skylake+)

```c
void rsb_stuff(void) {
    asm(".rept 16\n" "call 1f\n"
"pause ; lfence\n"
"1:\n"
".endr\n"
"addq $(8 * 16),%rsp\n");
}
```
Software Patch: RSB refilling

- RSB underfill (Skylake+)
  - RSB switch to BTB if RSB is empty

```c
void rsb_stuff(void) {
    asm(".rept 16\n" "call 1f\n" "pause ; l fence\n" "1: \n" ".endr\n" ".endr\n" "addq $(8 * 16),%rsp\n");
}
```
Software Patch: RSB refilling

- RSB underfill (Skylake+)
  - RSB switch to BTB if RSB is empty
  - Enables attacker to bypass defense

```c
void rsb_stuff(void) {
    asm(".rept 16\n" "call 1f\n" "pause ; 1fence\n" "1: \n" ".endr\n" ".endr\n" "addq $(8 * 16),%rsp\n");
}
```
Software Patch: RSB refilling

- RSB underfill (Skylake+)
  - RSB switch to BTB if RSB is empty
  - Enables attacker to bypass defense
  - Fill the RSB with a sequence of benign address

```c
void rsb_stuff(void) {
    asm(".rept 16\n" "call 1f\n"
        "pause ; 1fence\n"
        "1: \n"
        ".endr\n"
        ".endr\n"
        "addq $(8 * 16),%rsp\n";)
}
```
Attack 2: Across different threads/process

• Attack setup:

  • The attacker and victim run on a same core (Share RSB)

  • Synchronize threads using futex operations to control their interleaving
Attack 2.a: Colluding threads (User)

```
RSB

Cache
0x11100101
```
Attack 2.a: Colluding threads (User)
Attack 2.a: Colluding threads (User)

1. Poison RSB

2. Flushing the stack from cache
Attack 2.a: Colluding threads (User)

1. Poison RSB

2. Flushing the stack from cache
Attack 2.a: Colluding threads (User)

1. Poison RSB

2. Flushing the stack from cache

Cache

0x11100101

RSB

0x1011

ret
Attack 2.a: Colluding threads (User)

Poison RSB
Flush the stack from cache
Speculative load

0x1011 movzx %al, %rbx
0x1012 shl &9, %rbx
0x1013 movzx (%[array], rbx, 1), %rcx

RSB
0x1011

Cache
0x11100101
Password
Attack 2.a: Colluding threads (User)

1. Poison RSB
2. Flushing the stack from cache
3. ret
4. Speculative load

0x1011 movzx %al, %rbx
0x1012 shl &9, %rbx
0x1013 movzx (%[array], rbx, 1), %rcx
Attack2.b: Colluding threads (kernel)
Attack2.b: Colluding threads (kernel)
Attack2.b: Colluding threads(kernel)

Blocking syscall(kernel)  Polluting the RSB
Attack2.b: Colluding threads(kernel)

- Blocking syscall(kernel)
- Polluting the RSB
- Flushing the cache
Attack2.b: Colluding threads (kernel)
Attack2.b: Colluding threads (kernel)

1. Blocking syscall (kernel)
2. Return from blocking call
3. Execute attacker’s gadget
4. Polluting the RSB
5. Flushing the cache
Attack2.b: Colluding threads (kernel)

Blocking syscall (kernel)

Polluting the RSB

Flush+Reload

Flush the cache

Return from blocking call

Execute attacker’s gadget
Discussion on Attack 2
Discussion on Attack 2

• RSB Refilling
Discussion on Attack 2

• RSB Refilling
  • Linux has developed it for Skylake+ processors
Discussion on Attack 2

• RSB Refilling
  • Linux has developed it for **Skylake**+ processors
Discussion on Attack 2

- RSB Refilling
  - Linux has developed it for *Skylake*+ processors
  - Xeon and older processor

![Intel Skylake and Xeon processors]
Discussion on Attack 2

• RSB Refilling
  • Linux has developed it for *Skylake+* processors  
    • Xeon and older processor
Discussion on Attack 2

• RSB Refilling
  • Linux has developed it for Skylake+ processors ✗
  • Xeon and older processor ✓
  • Microsoft windows does not implement it
Discussion on Attack 2

• RSB Refilling
  • Linux has developed it for **Skylake+** processors ✗
  • Xeon and older processor ✓
  • Microsoft windows does not implement it ✓
Discussion on Attack 2

• RSB Refilling
  • Linux has developed it for Skylake+ processors ❌
  • Xeon and older processor ✔️
  • Microsoft windows does not implement it ✔️
Discussion on Attack 2

• RSB Refilling
  • Linux has developed it for Skylake+ processors ✗
  • Xeon and older processor ✓
  • Microsoft windows does not implement it ✓

• Update: linux-mainline released a new patch based on our suggestion to refill RSB unconditionally
Discussion on Attack 2

• **RSB Refilling**
  • Linux has developed it for *Skylake*+ processors ❌
  • Xeon and older processor ✓
  • Microsoft Windows does not implement it ✓

• Update: Linux-mainline released a new patch based on our suggestion to refill RSB unconditionally

• Retpoline
Discussion on Attack 2

• RSB Refilling
  • Linux has developed it for Skylake+ processors ✗
  • Xeon and older processor ✓
  • Microsoft windows does not implement it ✓

• Update: linux-mainline released a new patch based on our suggestion to refill RSB unconditionally

• Retpoline
  • Only modifies indirect call and jmp
Discussion on Attack 2

- **RSB Refilling**
  - Linux has developed it for **Skylake+** processors
  - Xeon and older processor
  - Microsoft windows does not implement it

- **Update**: Linux-mainline released a new patch based on our suggestion to refill RSB unconditionally

- **Retpoline**
  - Only modifies indirect call and jmp
Discussion on Attack 2
Discussion on Attack 2

- SMEP
Discussion on Attack 2

• SMEP
  • Prevent the kernel attack if the attacker gadget is in the user space
Discussion on Attack 2

• SMEP
  • Prevent the kernel attack if the attacker gadget is in the user space
Discussion on Attack 2

• SMEP
  - Prevent the kernel attack if the attacker gadget is in the user space

• What if an attacker poison the RSB with an address from kernel (e.g. ebpf)
Discussion on Attack 2

• SMEP
  • Prevent the kernel attack if the attacker gadget is in the user space ✗
  • What if an attacker poison the RSB with an address from kernel(e.g ebpf) ✓
Discussion on Attack 2

• SMEP
  • Prevent the kernel attack if the attacker gadget is in the user space  
    ✗
  • What if an attacker poison the RSB with an address from kernel(e.g ebpf)  
    ✓

• IBPB /IBRS
Discussion on Attack 2

• **SMEP**
  - Prevent the kernel attack if the attacker gadget is in the user space ✗
  - What if an attacker poison the RSB with an address from kernel (e.g. ebpf) ✓

• **IBPB /IBRS**
  - Does it issue in correct place?
Discussion on Attack 2

• SMEP
  • Prevent the kernel attack if the attacker gadget is in the user space ✗
  • What if an attacker poison the RSB with an address from kernel(e.g ebpf) ✓

• IBPB /IBRS
  • Does it issue in correct place?
  • Does IBPB reset the RSB in the latest microcode version?
### Other Attack Scenarios

<table>
<thead>
<tr>
<th>Attack on SGX</th>
<th>Attack on other process</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Reveal Data from SGX enclave</td>
<td>• Run on the same core</td>
</tr>
<tr>
<td>• Triggering an unmatched return</td>
<td>• Need to know the address of victim’s stack</td>
</tr>
<tr>
<td>• IBPB prevent it based on the new contact with Intel engineer.</td>
<td>• Bypassing ASLR</td>
</tr>
<tr>
<td></td>
<td>• RSB refilling/IBPB may stop the attack</td>
</tr>
</tbody>
</table>
Conclusion

- We introduced a new variant of Spectre attack which exploits Return Stack buffer

- Discussed different types of SpectreRSB against existing microcode and software patches
Thank you!

Questions?