Spectre Returns! Speculation Attacks Using Return Stack Buffer

Esmaeil Mohammadian,

Khaled N. Khasawneh, Chengyue Song

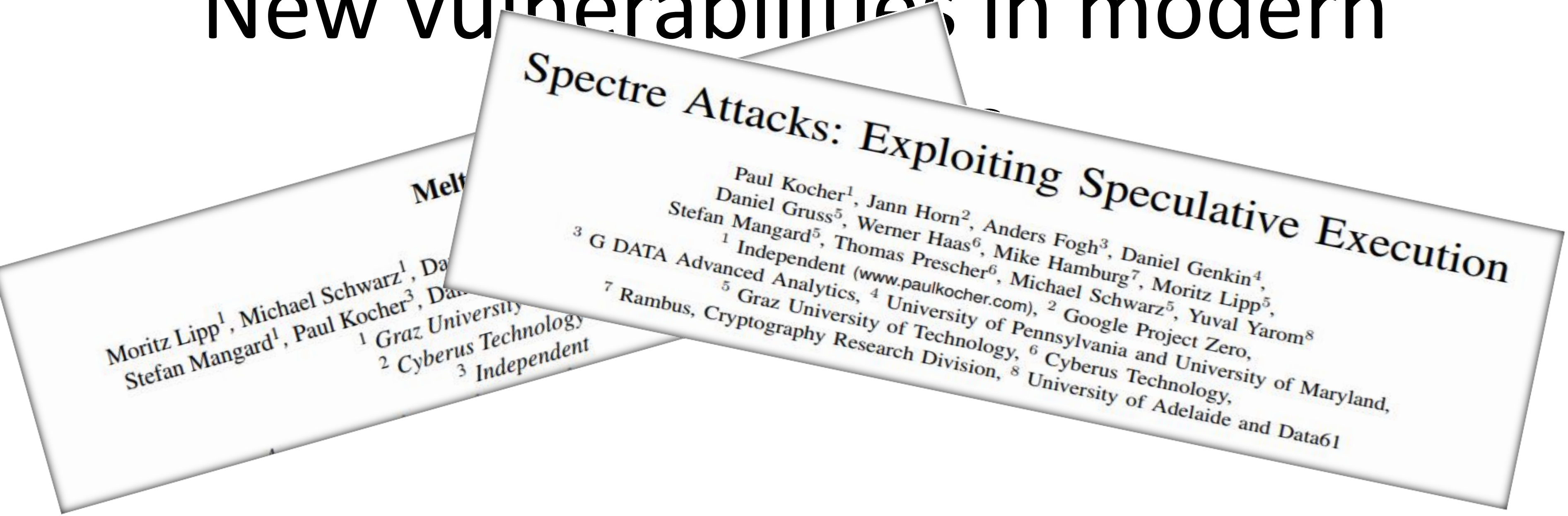
and Nael Abu-Ghazaleh

University of California, Riverside

UC RIVERSIDE UNIVERSITY OF CALIFORNIA

WOOT 2018 – BALTIMORE, USA

New vulnerabilities in modern



Melt

Moritz Lipp[1], Michael Schwarz[1], Da
Stefan Mangard[1], Paul Kocher[3], Da
[1] Graz Universit
[2] Cyberus Technolog
[3] Independent

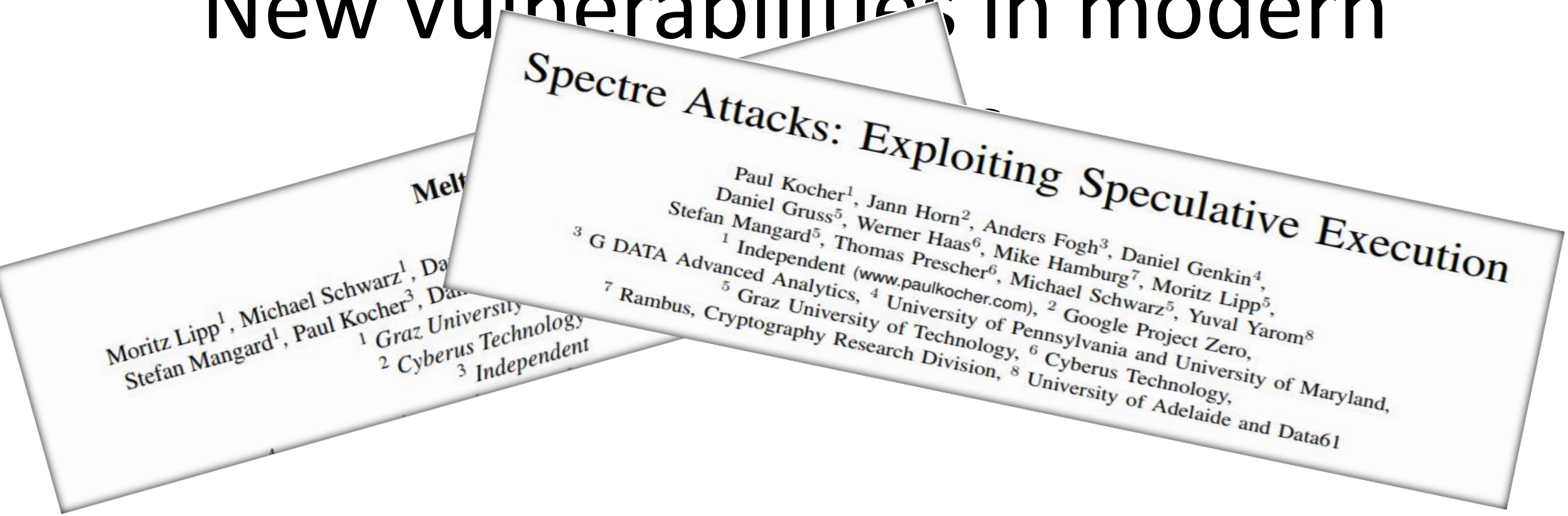
Spectre Attacks: Exploiting Speculative Execution

Paul Kocher[1], Jann Horn[2], Anders Fogh[3], Daniel Genkin[4],
Daniel Gruss[5], Werner Haas[6], Mike Hamburg[7], Moritz Lipp[5],
Stefan Mangard[5], Thomas Prescher[6], Michael Schwarz[5], Yuval Yarom[8]
[1] Independent (www.paulkocher.com), [2] Google Project Zero,
[3] G DATA Advanced Analytics, [4] University of Pennsylvania and University of Maryland,
[5] Graz University of Technology, [6] Cyberus Technology,
[7] Rambus, Cryptography Research Division, [8] University of Adelaide and Data61

Spectre
v1/v2/Meltdown(v3)

Jan 2018

New vulnerabilities in modern

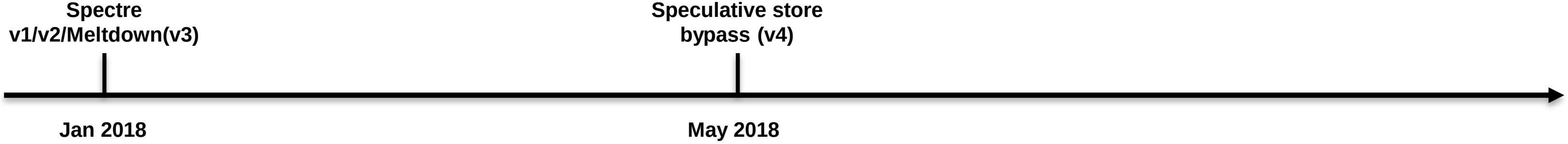


Melt

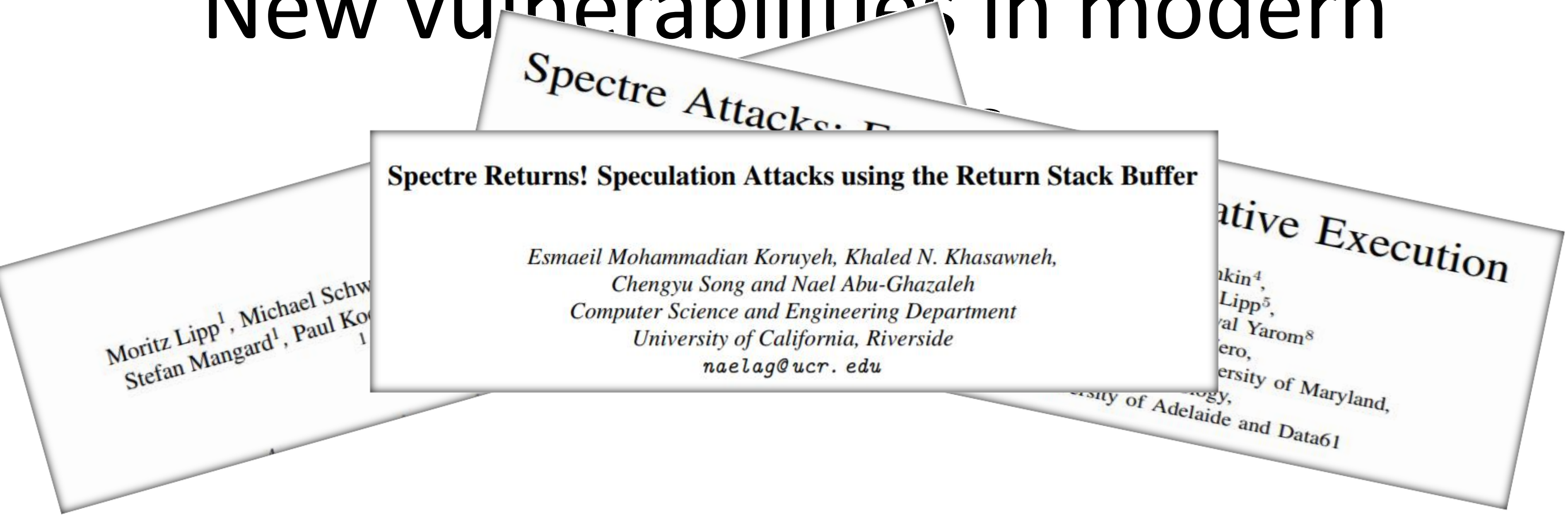
Moritz Lipp[1], Michael Schwarz[1], Da
Stefan Mangard[1], Paul Kocher[3], Da
[1] Graz Universit
[2] Cyberus Technolog
[3] Independent

Spectre Attacks: Exploiting Speculative Execution

Paul Kocher[1], Jann Horn[2], Anders Fogh[3], Daniel Genkin[4],
Daniel Gruss[5], Werner Haas[6], Mike Hamburg[7], Moritz Lipp[5],
Stefan Mangard[5], Thomas Prescher[6], Michael Schwarz[5], Yuval Yarom[8]
[1] Independent (www.paulkocher.com), [2] Google Project Zero,
[3] G DATA Advanced Analytics, [4] University of Pennsylvania and University of Maryland,
[5] Graz University of Technology, [6] Cyberus Technology,
[7] Rambus, Cryptography Research Division, [8] University of Adelaide and Data61



New vulnerabilities in modern



Spectre Attacks: E

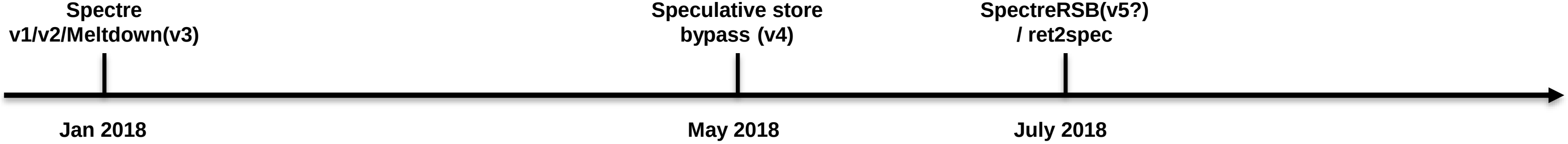
ative Execution

Moritz Lipp[1], Michael Schw
Stefan Mangard[1], Paul Ko
[1]

nkin[4],
Lipp[5],
al Yarom[8]
ero,
ersity of Maryland,
ogy,
rsity of Adelaide and Data61

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`



Spectre v1/v2/Meltdown(v3)

Jan 2018

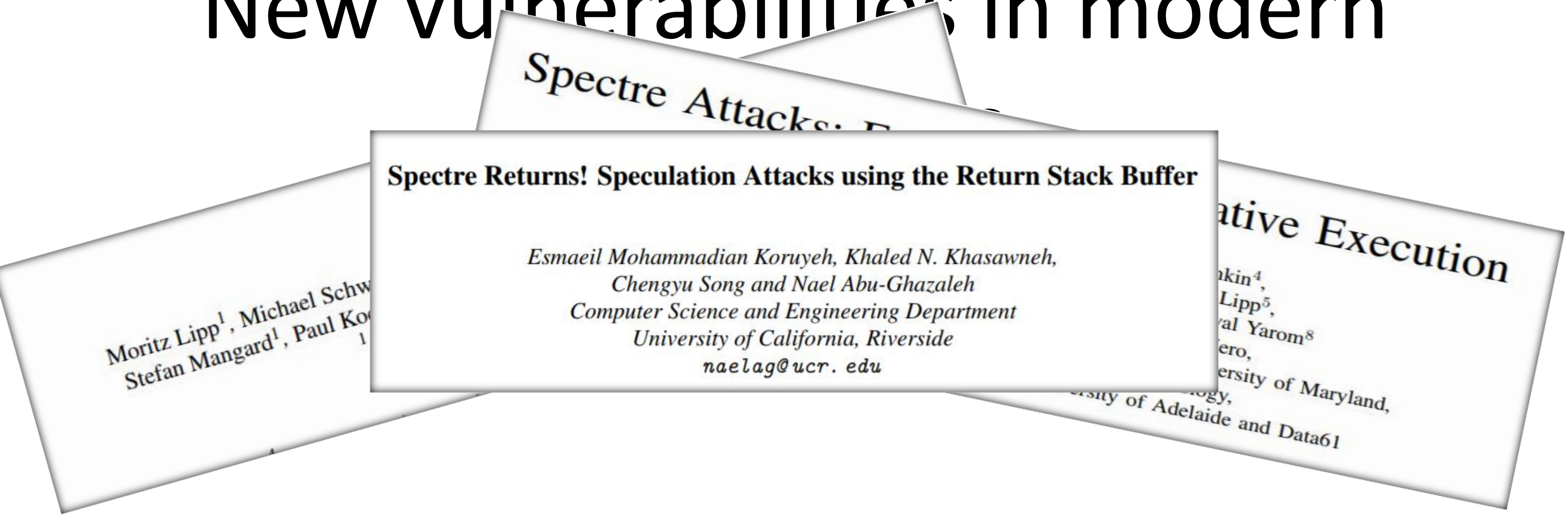
Speculative store bypass (v4)

May 2018

SpectreRSB(v5?) / ret2spec

July 2018

New vulnerabilities in modern



Spectre Attacks: E

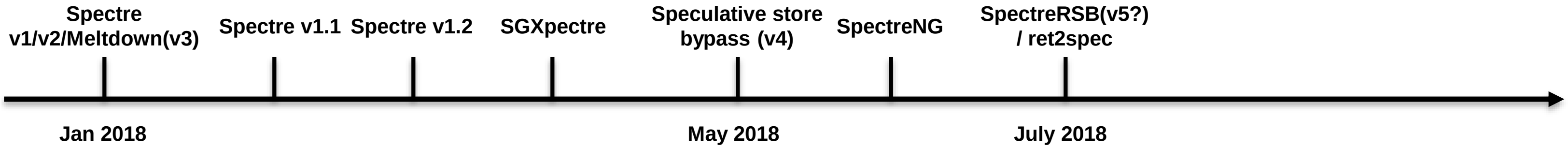
Moritz Lipp[1], Michael Schw
Stefan Mangard[1], Paul Ko
1

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

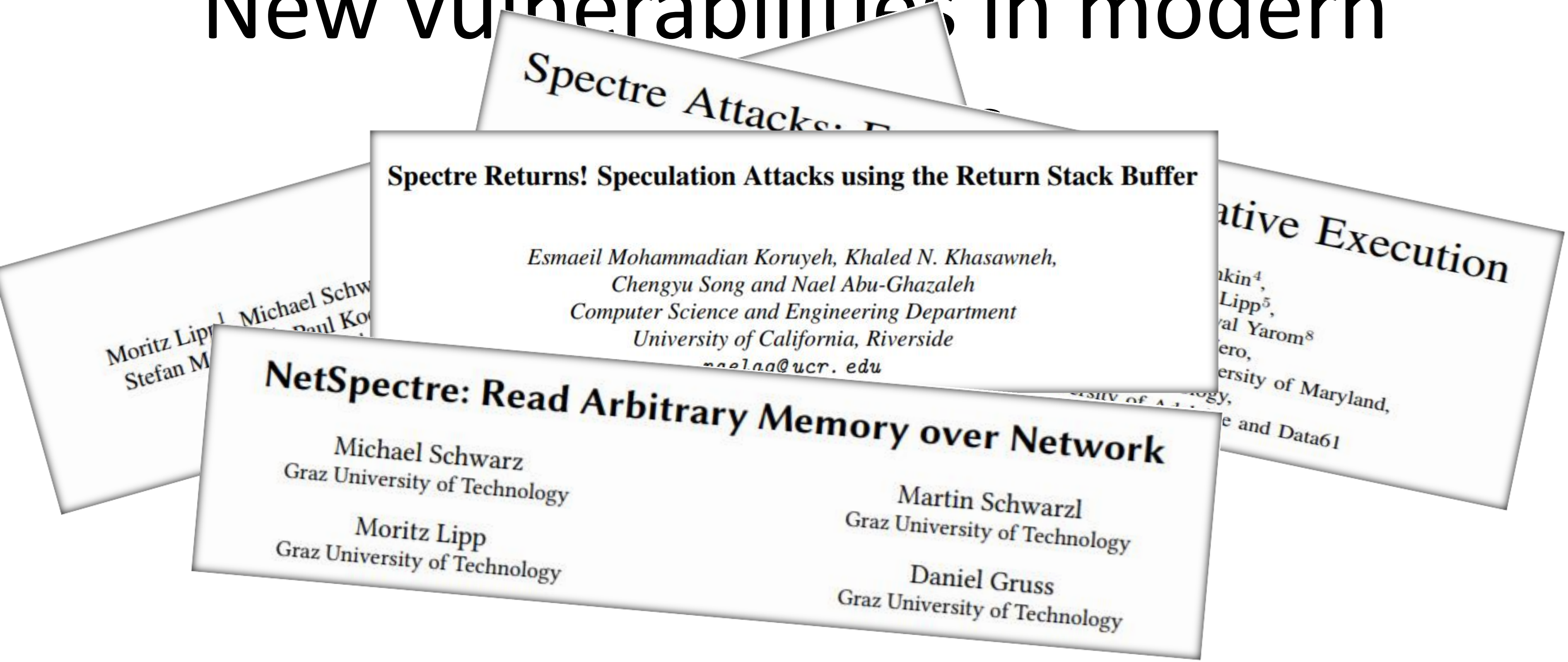
ative Execution

kin[4],
Lipp[5],
al Yarom[8]
ero,
ersity of Maryland,
ogy,
rsity of Adelaide and Data61



Spectre v1/v2/Meltdown(v3)
Spectre v1.1
Spectre v1.2
SGXpectre
Speculative store bypass (v4)
SpectreNG
SpectreRSB(v5?) / ret2spec
Jan 2018
May 2018
July 2018

New vulnerabilities in modern



Spectre Attacks: E

ative Execution

Moritz Lipp1, Michael Schw
Paul Ko
Stefan M

nkin4,
Lipp5,
al Yarom8
ero,
ersity of Maryland,
ogy,
sity of A
e and Data61

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

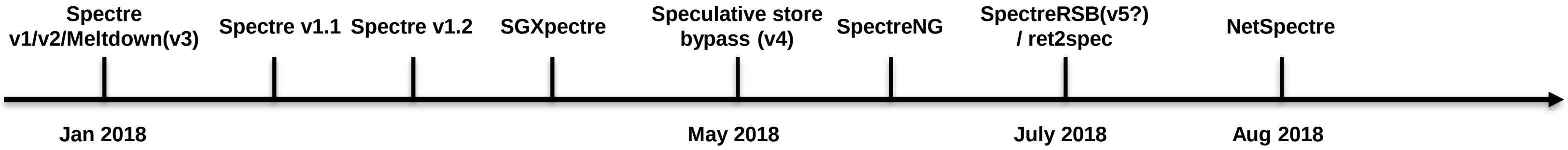
NetSpectre: Read Arbitrary Memory over Network

Michael Schwarz
Graz University of Technology

Moritz Lipp
Graz University of Technology

Martin Schwarzl
Graz University of Technology

Daniel Gruss
Graz University of Technology



Spectre v1/v2/Meltdown(v3)

Spectre v1.1

Spectre v1.2

SGXpectre

Speculative store bypass (v4)

SpectreNG

SpectreRSB(v5?) / ret2spec

NetSpectre

Jan 2018

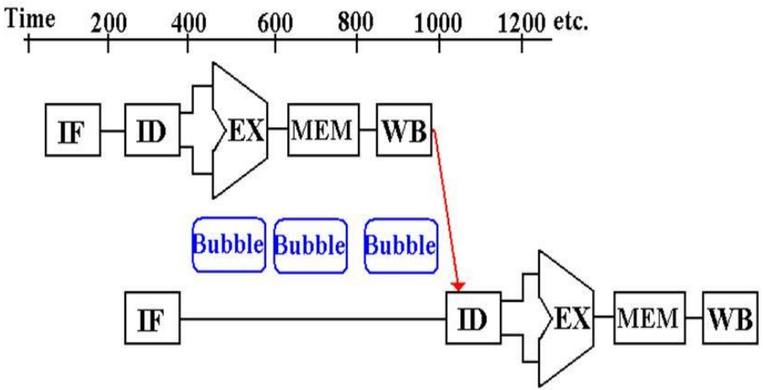
May 2018

July 2018

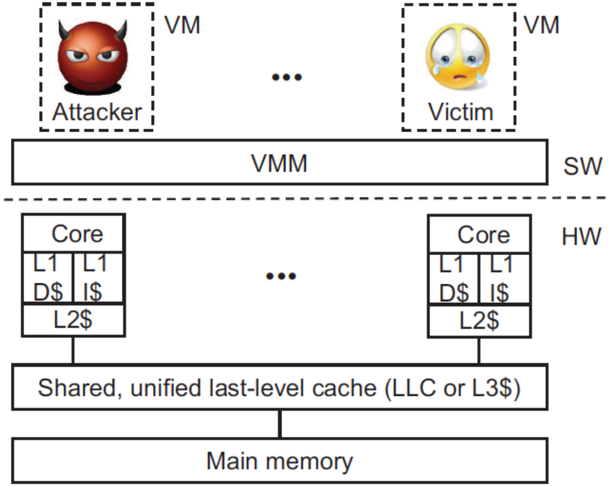
Aug 2018

Main components of the Attack

Out of Order Execution

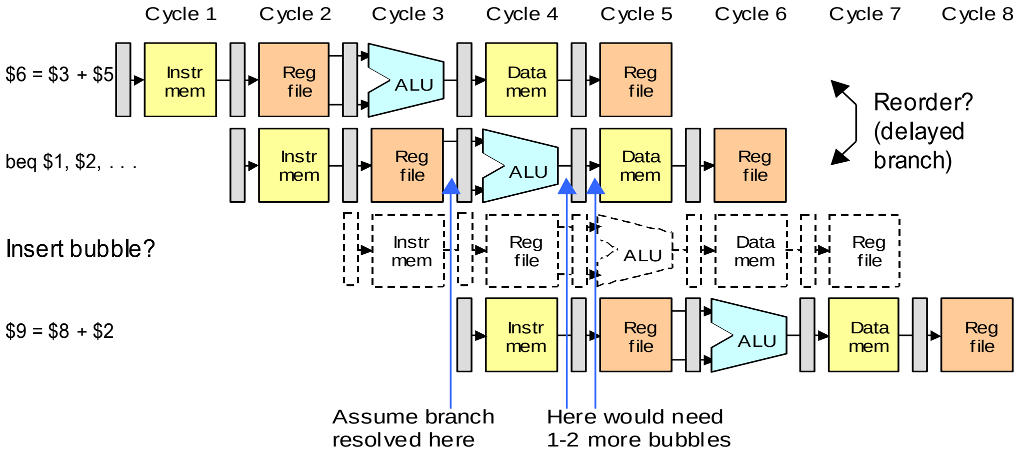


Side channel Attack



Out of Order Execution(OoO)

- Speculation is critical to modern CPU performance



(OoO): Branch predictors

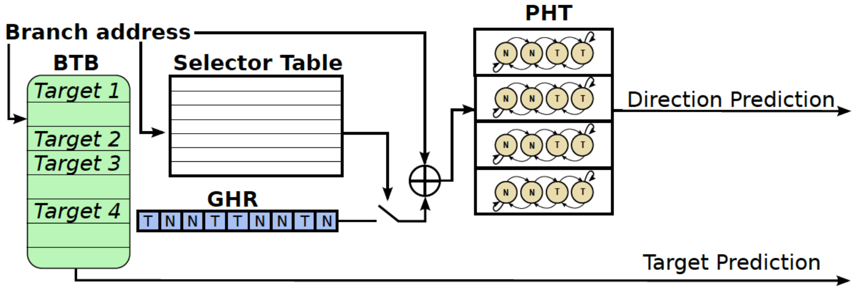
- During speculation processors **guess** the future stream instructions of the program

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

Branch predictors

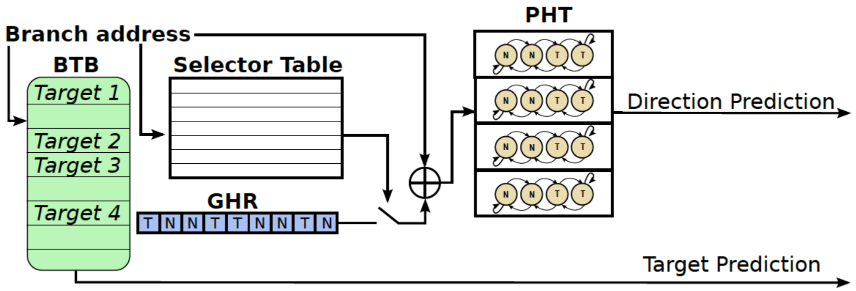
Branch predictors

- Two hardware predictors:



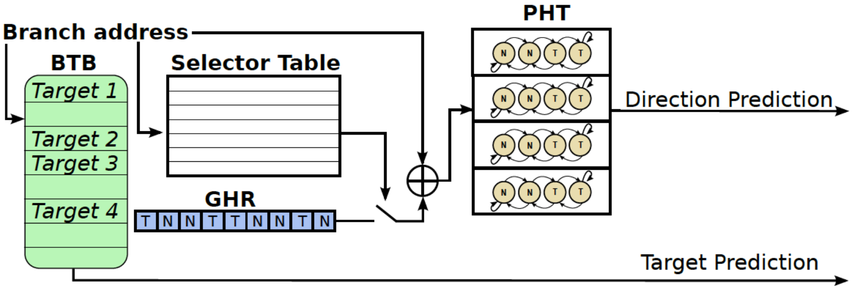
Branch predictors

- 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

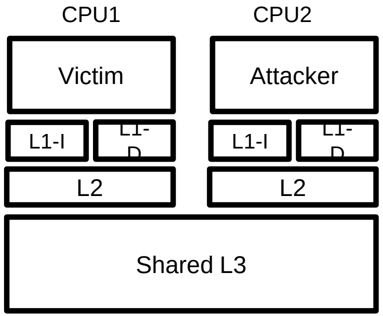
- Access to the data inside the cache is fast

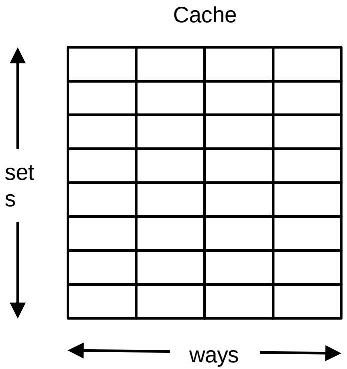
- Loading data from memory is too slow

Cache Side channel Attacks

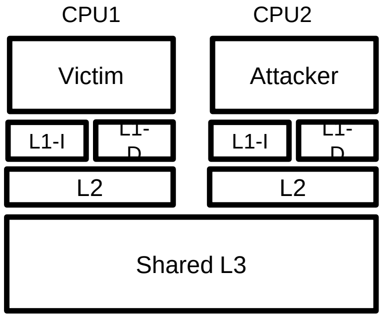
- Access to the data inside the cache is fast
- Loading data from memory is too slow
- Exploits timing differences that are introduced by the caches
  - Flush and reload
  - Prime and probe
  - …

Side channel: Flush+Reload Attack

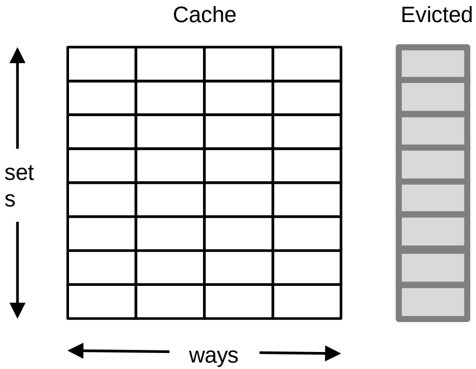




Side channel: Flush+Reload Attack



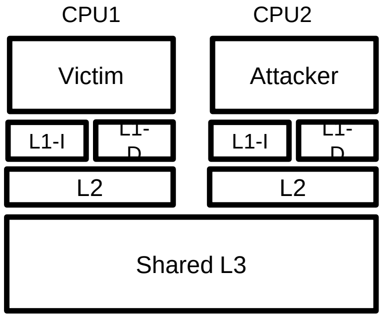
1- Flush each line in the critical data

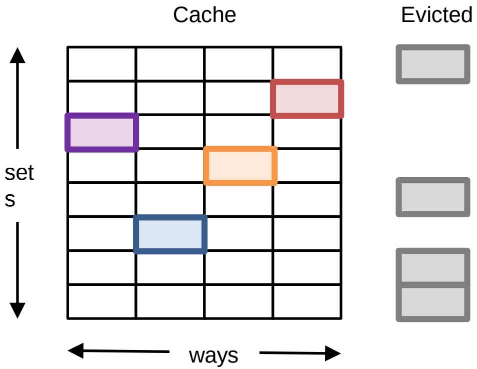


Side channel: Flush+Reload Attack

2- Victim accesses critical data

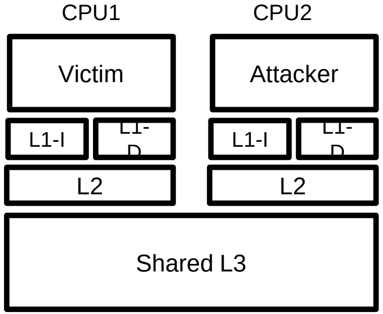
1- Flush each line in the critical data





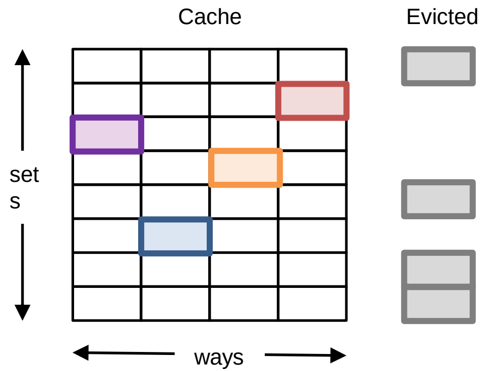
Side channel: Flush+Reload Attack

2- Victim accesses critical data



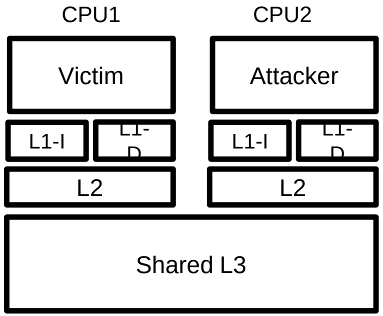
1- Flush each line in the critical data

3- Reload critical data (measure time)



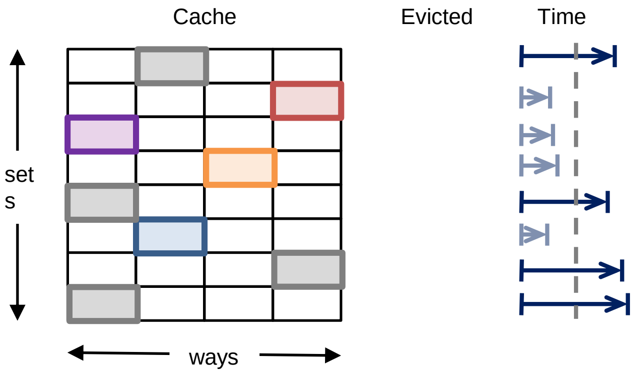
Side channel: Flush+Reload Attack

2- Victim accesses critical data



1- Flush each line in the 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

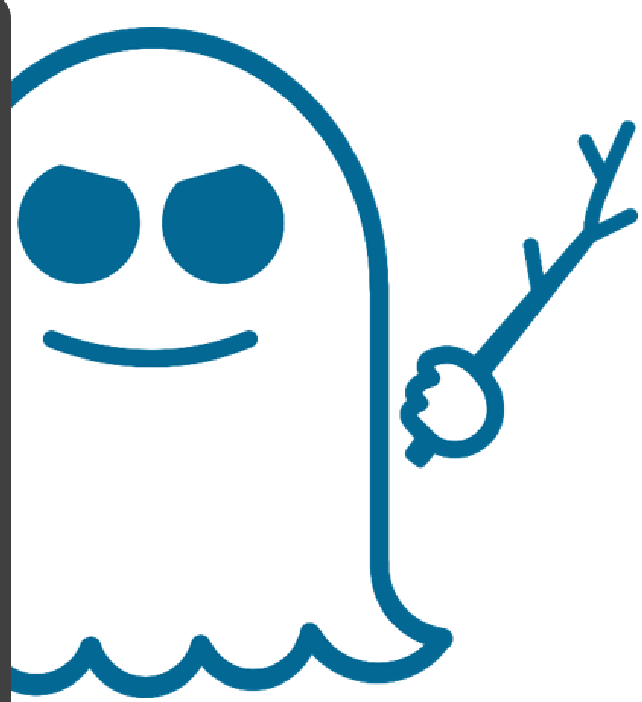
- Spectre Variant 1:
  - Train the Direction predictor (PHT) to bypass bound checking and leak sensitive data.
- Spectre Variant 2:

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.

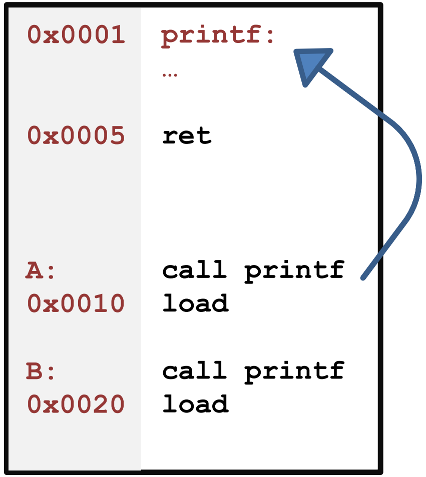
```
0x0001  printf:
        …

0x0005  ret



A:      call printf
0x0010  load

B:      call printf
0x0020  load
```

Branch Target Buffer

v	tag	target
1		
1		
0		

Why Return Stack Buffer (RSB)?

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



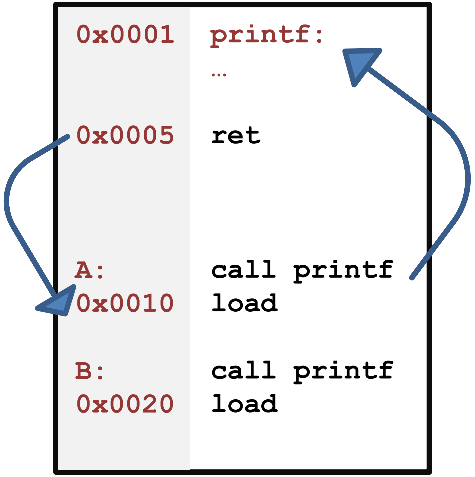
```
0x0001   printf:
         …

0x0005   ret



A:       call printf
0x0010   load

B:       call printf
0x0020   load
```

Branch Target Buffer

v	tag	target
1		
1		
0		

Why Return Stack Buffer (RSB)?

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



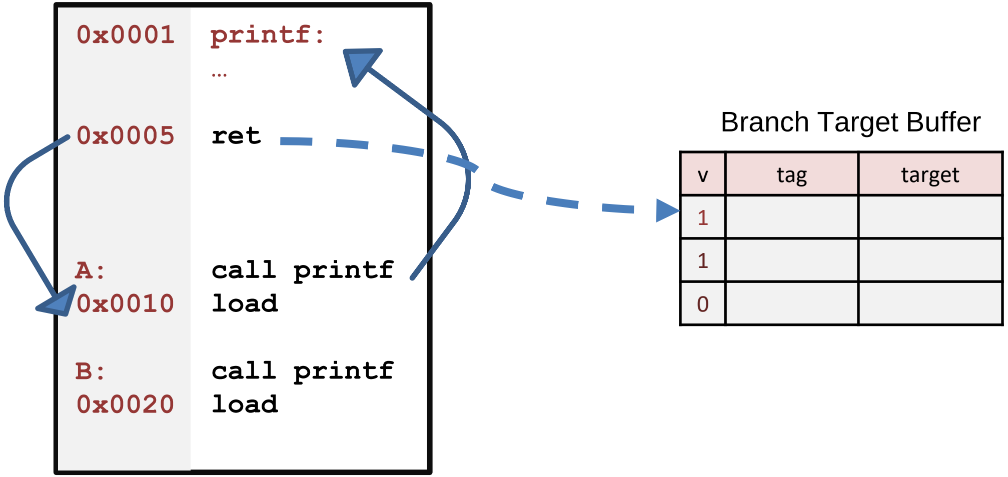
```
0x0001  printf:
        …
0x0005  ret

A:      call printf
0x0010  load

B:      call printf
0x0020  load
```

Branch Target Buffer

v	tag	target
1		
1		
0		

Why Return Stack Buffer (RSB)?

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



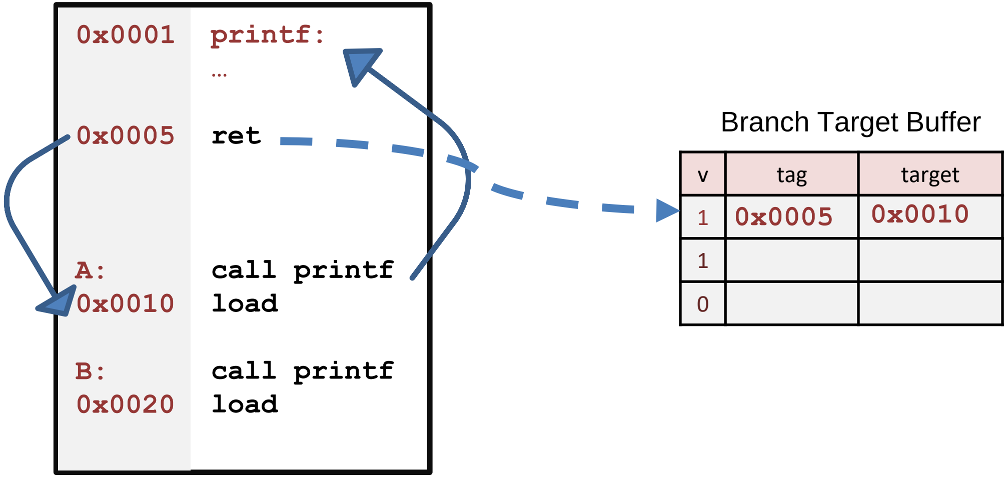
```
0x0001   printf:
         …

0x0005   ret



A:       call printf
0x0010   load

B:       call printf
0x0020   load
```

Branch Target Buffer

v	tag	target
1		
1		
0		

Why Return Stack Buffer (RSB)?

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



```
0x0001   printf:
         …

0x0005   ret


A:       call printf
0x0010   load

B:       call printf
0x0020   load
```

Branch Target Buffer

v	tag	target
1	0x0005	0x0010
1		
0		

Why Return Stack Buffer (RSB)?

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

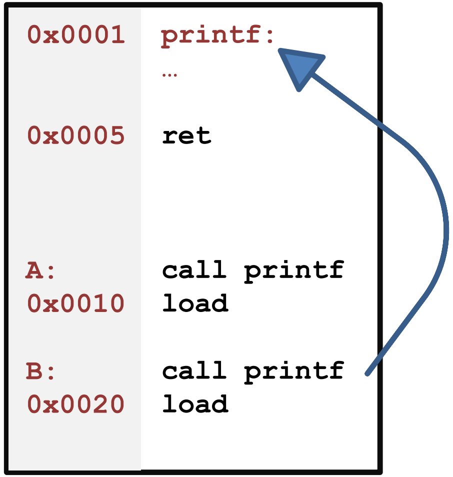
```
0x0001   printf:
         …

0x0005   ret



A:       call printf
0x0010   load

B:       call printf
0x0020   load
```

Branch Target Buffer

v	tag	target
1	0x0005	0x0010
1		
0		

Why Return Stack Buffer (RSB)?

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



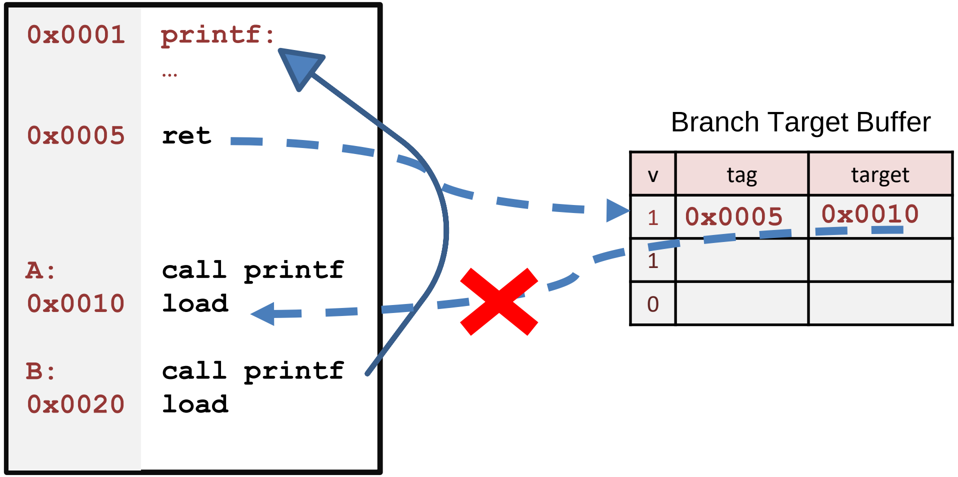
```
0x0001   printf:
         …

0x0005   ret


A:       call printf
0x0010   load

B:       call printf
0x0020   load
```

Branch Target Buffer

v	tag	target
1	0x0005	0x0010
1		
0		

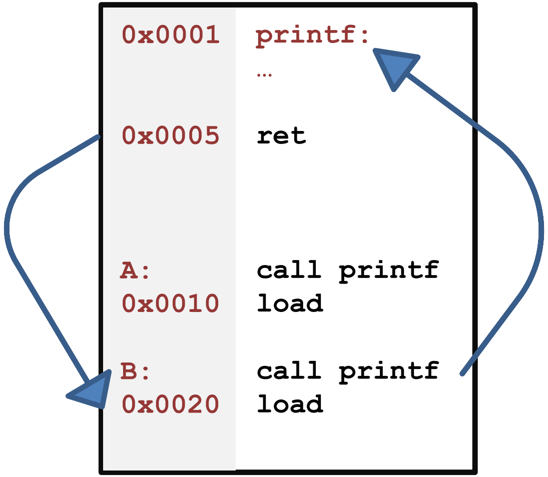
Why Return Stack Buffer (RSB)?

- 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:       call printf
0x0010   load

B:       call printf
0x0020   load
```

Branch Target Buffer

v	tag	target
1	0x0005	0x0010
1		
0		

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 ***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 ***ret*** instruction
- RSB is shared between two hardware threads
- 16 to 24 entries
- Push pc+4 onto the RSB on each *call* instruction
- Pop an address off the RSB on each *ret* instruction

RSB Pollution

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

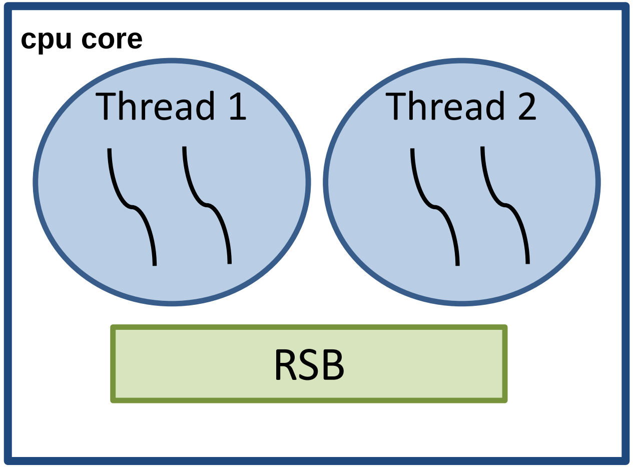
- call → *push address; ret;*

  → 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

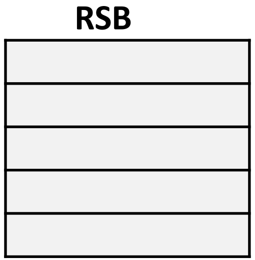
- Lunched from a process to part of its own address space

- 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

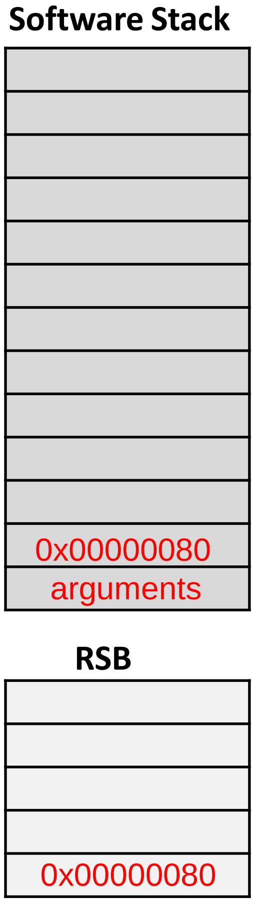
Attack 1: Basic Attack

```
0X00000010    pollute:
                   push %rbp
                   mov %rsp,%rbp
                   pop %rdi
                   pop %rdi
                   pop %rdi
                   pop %rdi
                   pop %rbp
                   clflush (%rsp)
0X00000019         retq


0X00000020    speculative:
0X00000021       call pollute
0X00000022☠      movzx (%[array],rbx)

0X00000030    main:
0X00000031    call speculative
0X00000032        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)
  0X00000019          retq


  0X00000020     speculative:
  0X00000021        call pollute
  0X00000022☠       movzx (%[array],rbx)

⇨ 0X00000030     main:
  0X00000031     call speculative
  0X00000032         rdtscp
                     access
                     rdtscp
```



Software Stack

0x00000080

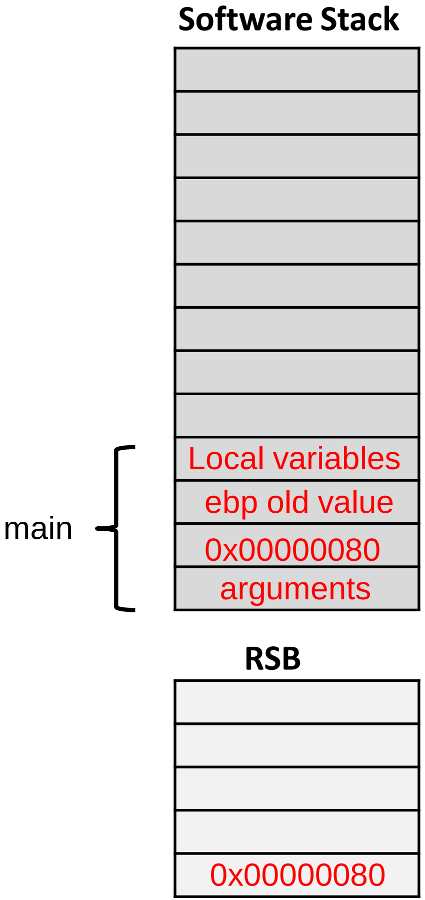
arguments

RSB

0x00000080

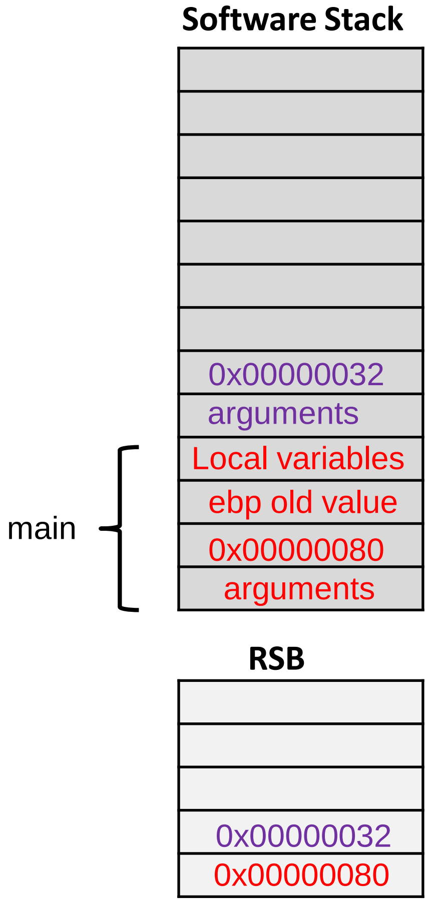
Attack 1: Basic Attack

```
0x00000010   pollute:
                  push %rbp
                  mov %rsp,%rbp
                  pop %rdi
                  pop %rdi
                  pop %rdi
                  pop %rdi
                  pop %rbp
                  clflush (%rsp)
0x00000019        retq


0x00000020   speculative:
0x00000021       call pollute
0x00000022☠      movzx (%[array],rbx)

0x00000030   main:
0x00000031   call speculative
0x00000032       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)
   0X00000019        retq


⇒  0X00000020   speculative:
   0X00000021     call pollute
   0X00000022☠    movzx (%[array],rbx)

   0X00000030   main:
   0X00000031   call speculative
   0X00000032       rdtscp
                    access
                    rdtscp
```



Software Stack

0x00000032

arguments

Local variables

ebp old value

0x00000080

arguments

main

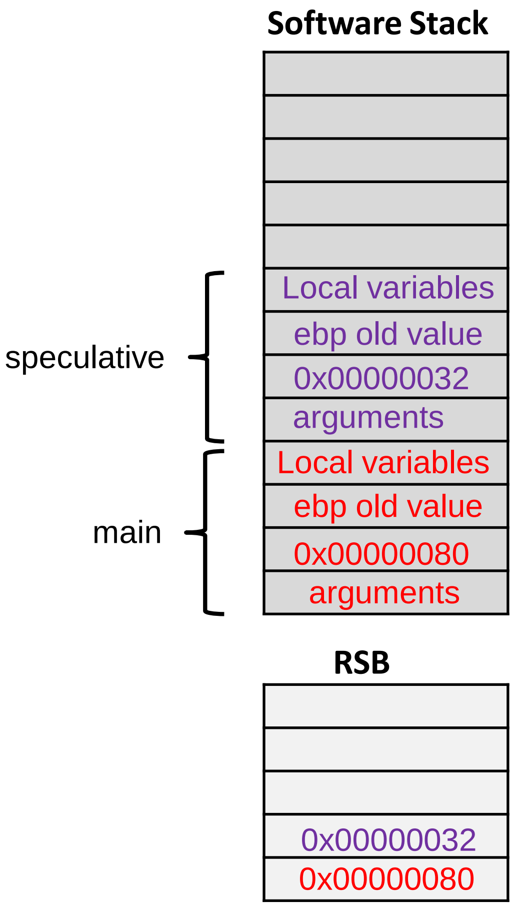
RSB

0x00000032

0x00000080

Attack 1: Basic Attack

```
0X00000010    pollute:
                   push %rbp
                   mov %rsp,%rbp
                   pop %rdi
                   pop %rdi
                   pop %rdi
                   pop %rdi
                   pop %rbp
                   clflush (%rsp)
0X00000019         retq


0X00000020    speculative:
0X00000021         call pollute
0X00000022☠      movzx (%[array],rbx)

0X00000030    main:
0X00000031    call speculative
0X00000032         rdtscp
                   access
                   rdtscp
```


Software Stack

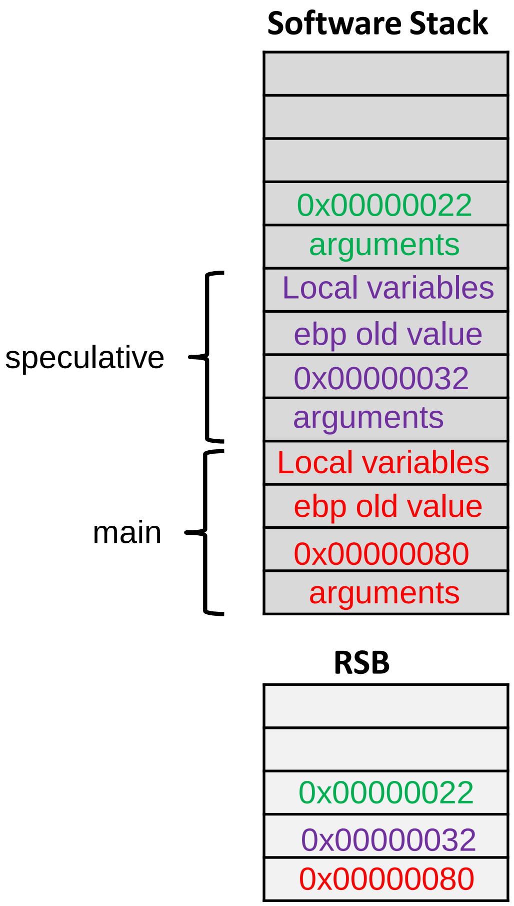
Local variables
ebp old value
0x00000032
arguments
Local variables
ebp old value
0x00000080
arguments

RSB

0x00000032
0x00000080

Attack 1: Basic Attack

```
⇒ 0x00000010   pollute:
                         push %rbp
                         mov %rsp,%rbp
                         pop %rdi
                         pop %rdi
                         pop %rdi
                         pop %rdi
                         pop %rbp
                         clflush (%rsp)
  0x00000019             retq


  0x00000020   speculative:
  0x00000021      call pollute
  0x00000022☠    movzx (%[array],rbx)

  0x00000030   main:
  0x00000031   call speculative
  0x00000032         rdtscp
                     access
                     rdtscp
```



Software Stack

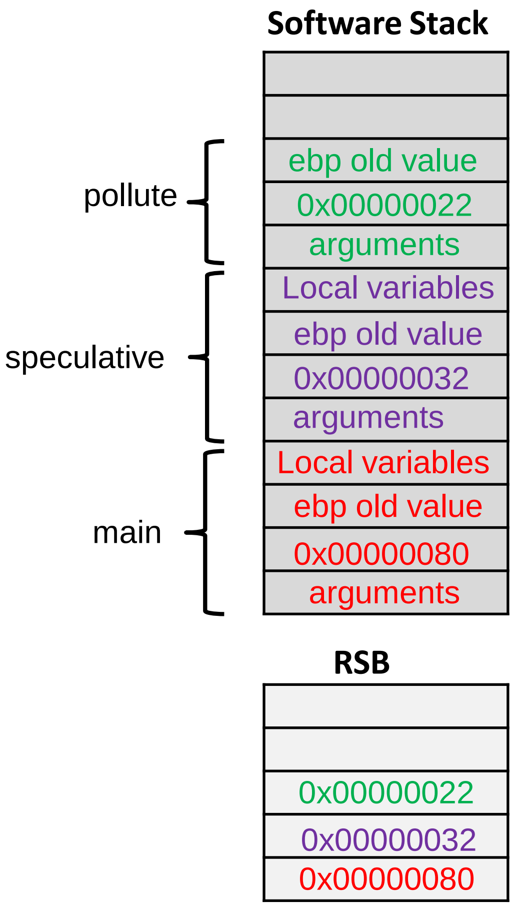
0x00000022
arguments
Local variables
ebp old value
0x00000032
arguments
Local variables
ebp old value
0x00000080
arguments

RSB

0x00000022
0x00000032
0x00000080

Attack 1: Basic Attack

```
0X00000010   pollute:
                  push  %rbp
                  mov  %rsp,%rbp
                  pop  %rdi
                  pop  %rdi
                  pop  %rdi
                  pop  %rdi
                  pop  %rbp
                  clflush  (%rsp)
0X00000019        retq


0X00000020   speculative:
0X00000021      call pollute
0X00000022☠     movzx  (%[array],rbx)

0X00000030   main:
0X00000031   call  speculative
0X00000032       rdtscp
                 access
                 rdtscp
```



Software Stack

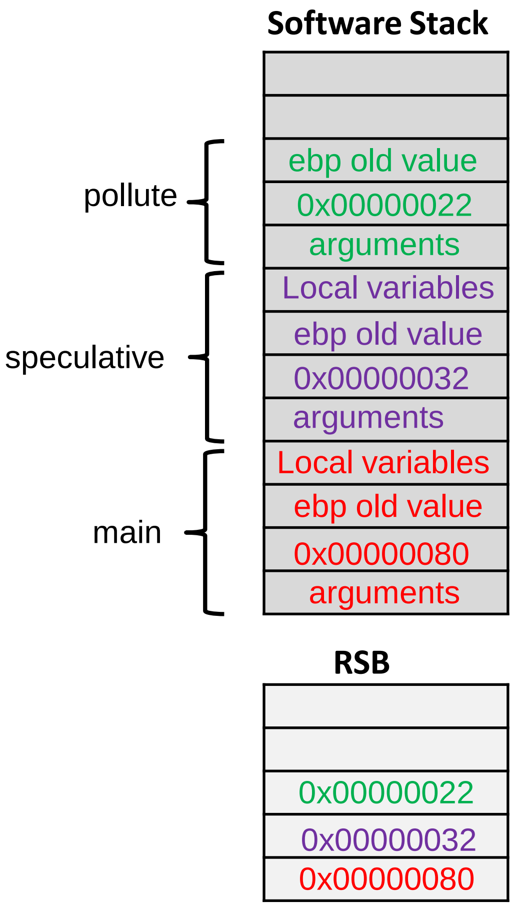
pollute	ebp old value
	0x00000022
	arguments
speculative	Local variables
	ebp old value
	0x00000032
	arguments
main	Local variables
	ebp old value
	0x00000080
	arguments

RSB

0x00000022
0x00000032
0x00000080

Attack 1: Basic Attack

```
0X00000010   pollute:
                  push  %rbp
                  mov  %rsp,%rbp
                  pop  %rdi
                  pop  %rdi
                  pop  %rdi
                  pop  %rdi
                  pop  %rbp
                  clflush (%rsp)
0X00000019        retq


0X00000020   speculative:
0X00000021      call pollute
0X00000022☠     movzx (%[array],rbx)

0X00000030   main:
0X00000031   call speculative
0X00000032        rdtscp
                  access
                  rdtscp
```



Software Stack

pollute: ebp old value, 0x00000022, arguments

speculative: Local variables, ebp old value, 0x00000032, arguments

main: Local variables, ebp old value, 0x00000080, arguments

RSB

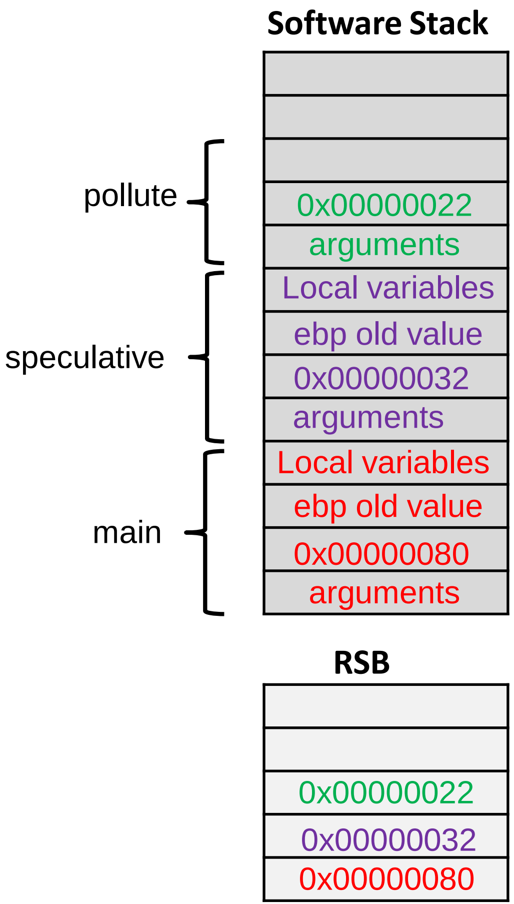
0x00000022

0x00000032

0x00000080

Attack 1: Basic Attack

```
0X00000010  pollute:
                 push %rbp
                 mov %rsp,%rbp
                 pop %rdi
                 pop %rdi
                 pop %rdi
                 pop %rdi
                 pop %rbp
                 clflush (%rsp)
0X00000019       retq


0X00000020  speculative:
0X00000021     call pollute
0X00000022☠    movzx (%[array],rbx)

0X00000030  main:
0X00000031  call speculative
0X00000032       rdtscp
                 access
                 rdtscp
```



Software Stack

pollute
0x00000022
arguments

speculative
Local variables
ebp old value
0x00000032
arguments

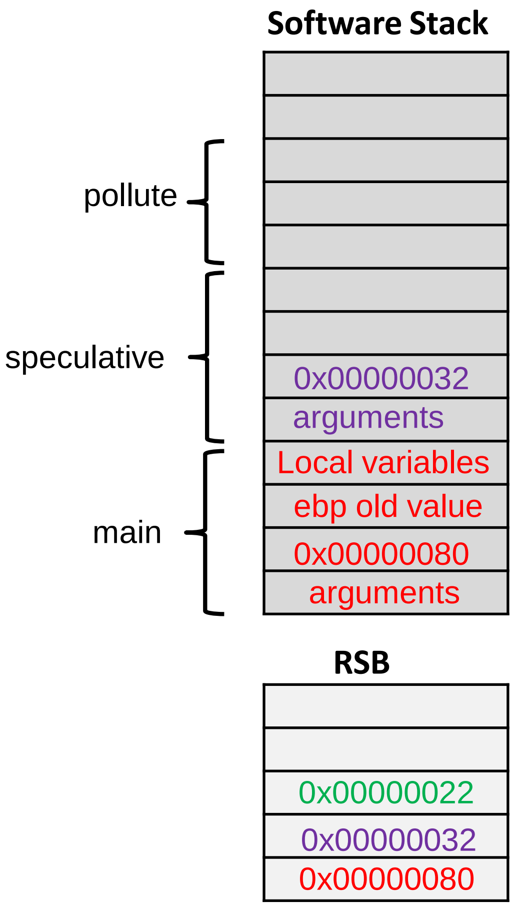
main
Local variables
ebp old value
0x00000080
arguments

RSB

0x00000022
0x00000032
0x00000080

Attack 1: Basic Attack

```
0X00000010   pollute:
                  push %rbp
                  mov %rsp,%rbp
                  pop %rdi
                  pop %rdi
                  pop %rdi
                  pop %rdi
                  pop %rbp
                  clflush (%rsp)
0X00000019        retq


0X00000020   speculative:
0X00000021     call pollute
0X00000022☠    movzx (%[array],rbx)

0X00000030   main:
0X00000031   call speculative
0X00000032        rdtscp
                  access
                  rdtscp
```



Software Stack

pollute

speculative

main

0x00000032

arguments

Local variables

ebp old value

0x00000080

arguments

RSB

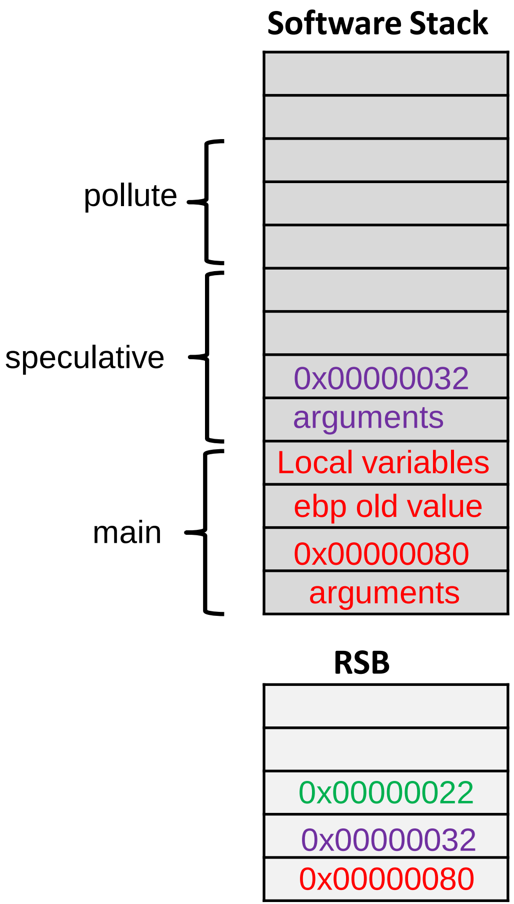
0x00000022

0x00000032

0x00000080

Attack 1: Basic Attack

```
0X00000010   pollute:
                  push  %rbp
                  mov  %rsp,%rbp
                  pop  %rdi
                  pop  %rdi
                  pop  %rdi
                  pop  %rdi
                  pop  %rbp
                  clflush  (%rsp)
0X00000019        retq


0X00000020   speculative:
0X00000021      call pollute
0X00000022☠     movzx  (%[array],rbx)

0X00000030   main:
0X00000031   call speculative
0X00000032        rdtscp
                  access
                  rdtscp
```



Software Stack

pollute

speculative

main

0x00000032

arguments

Local variables

ebp old value

0x00000080

arguments

RSB

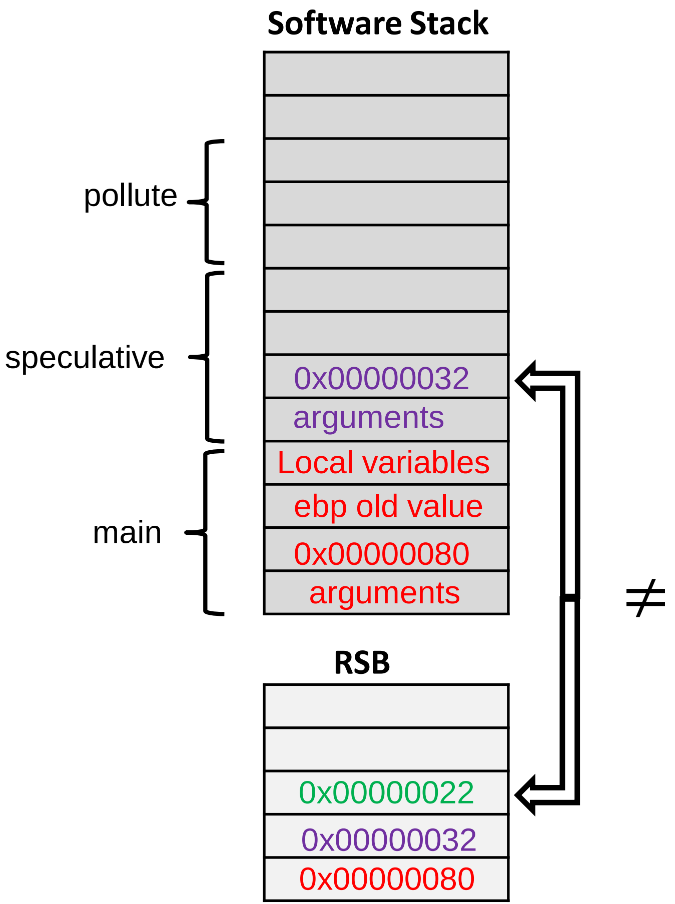
0x00000022

0x00000032

0x00000080

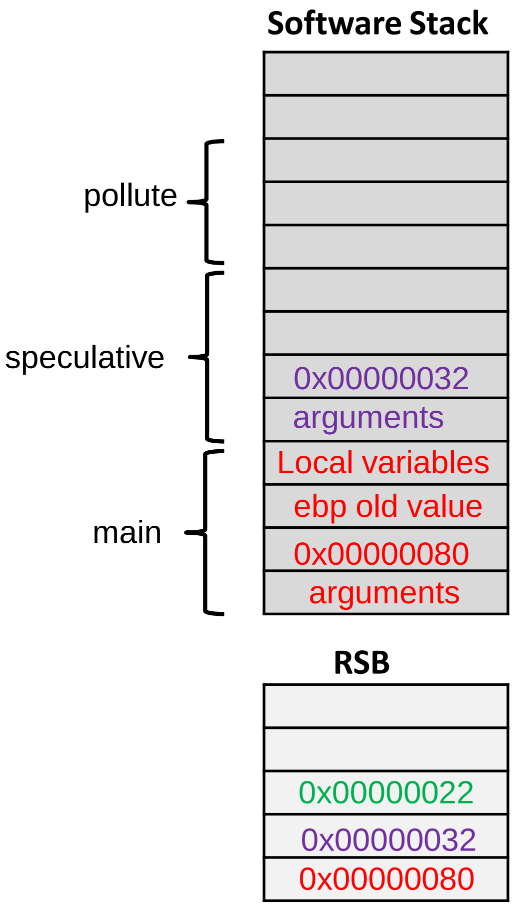
Attack 1: Basic Attack

```
0x00000010   pollute:
                 push %rbp
                 mov %rsp,%rbp
                 pop %rdi
                 pop %rdi
                 pop %rdi
                 pop %rdi
                 pop %rbp
                 clflush (%rsp)
0x00000019       retq


0x00000020   speculative:
0x00000021     call pollute
0x00000022☠    movzx (%[array],rbx)

0x00000030   main:
0x00000031   call speculative
0x00000032       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)
0x00000019        retq


0x00000020   speculative:
0x00000021     call pollute
0x00000022☠    movzx (%[array],rbx)

0x00000030   main:
0x00000031   call speculative
0x00000032        rdtscp
                  access
                  rdtscp
```



Software Stack

pollute

speculative

main

0x00000032

arguments

Local variables

ebp old value

0x00000080

arguments

RSB

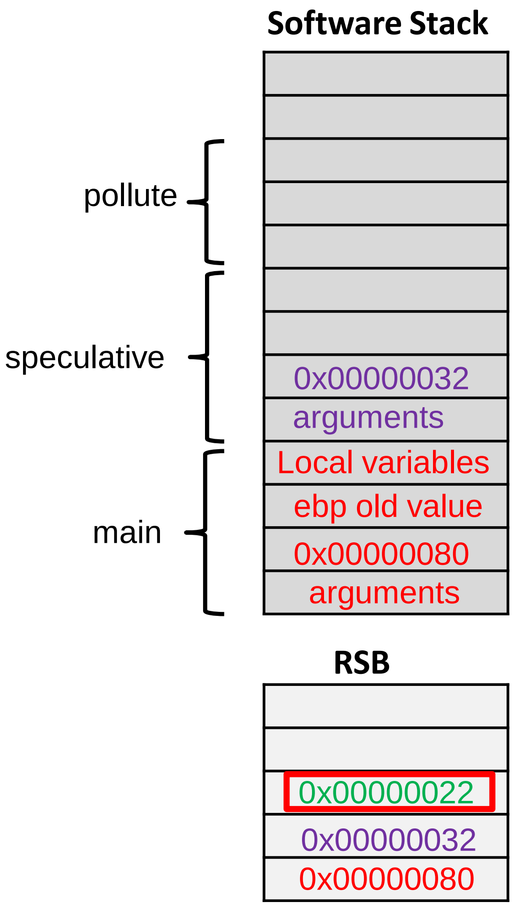
0x00000022

0x00000032

0x00000080

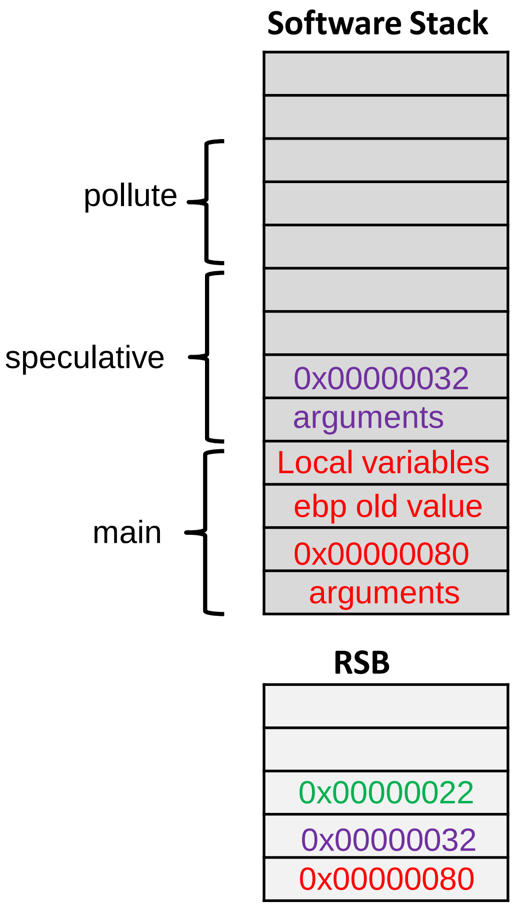
Attack 1: Basic Attack

```
0X00000010   pollute:
                   push %rbp
                   mov %rsp,%rbp
                   pop %rdi
                   pop %rdi
                   pop %rdi
                   pop %rdi
                   pop %rbp
                   clflush (%rsp)
0X00000019         retq


0X00000020   speculative:
0X00000021      call pollute
0X00000022☠    movzx (%[array],rbx)

0X00000030   main:
0X00000031   call speculative
0X00000032        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)
0X00000019              retq


0X00000020    speculative:
0X00000021         call pollute
0X00000022☠         movzx (%[array],rbx)

0X00000030    main:
0X00000031    call speculative
0X00000032              rdtscp
                        access
                        rdtscp
```



Software Stack

pollute

speculative

main

0x00000032

arguments

Local variables

ebp old value

0x00000080

arguments

RSB

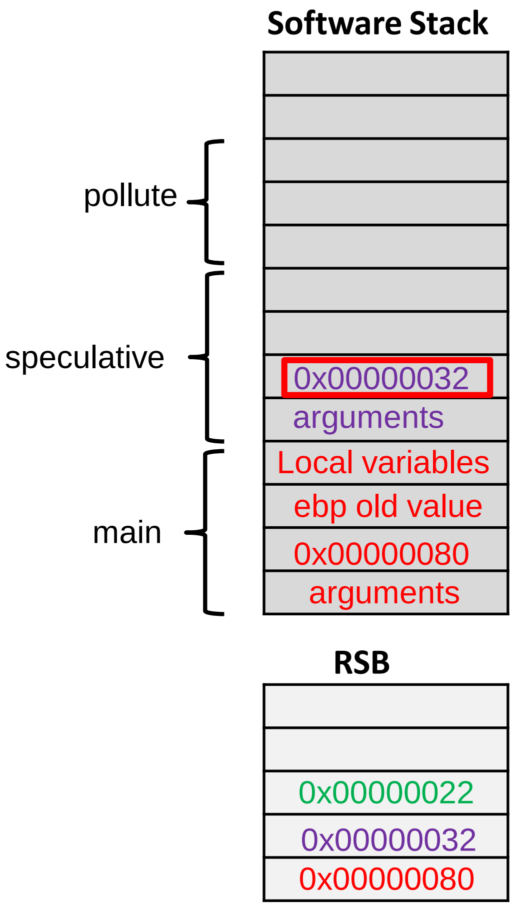
0x00000022

0x00000032

0x00000080

Attack 1: Basic Attack

```
  0X00000010    pollute:
                     push %rbp
                     mov %rsp,%rbp
                     pop %rdi
                     pop %rdi
                     pop %rdi
                     pop %rdi
                     pop %rbp
                     clflush (%rsp)
  0X00000019         retq


  0X00000020    speculative:
  0X00000021       call pollute
  0X00000022☠      movzx (%[array],rbx)

  0X00000030    main:
  0X00000031    call speculative
⇨ 0X00000032         rdtscp
                     access
                     rdtscp
```



Software Stack

pollute

speculative

main

0x00000032

arguments

Local variables

ebp old value

0x00000080

arguments

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+)

```
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

```
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
  - Enables attacker to bypass defense

```
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

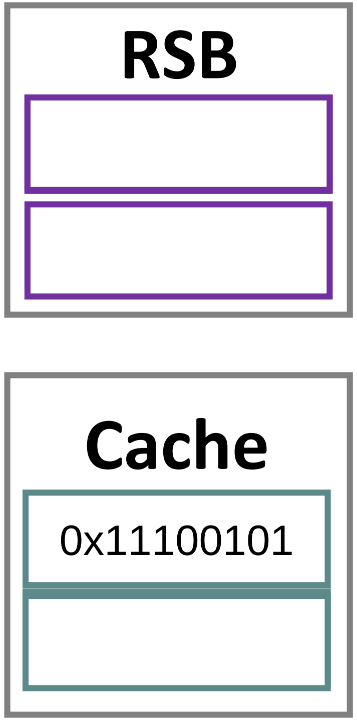
- RSB underfill (Skylake+)
  - RSB switch to BTB if RSB is empty
  - Enables attacker to bypass defense
  - Fill the RSB with a sequence of benign address

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

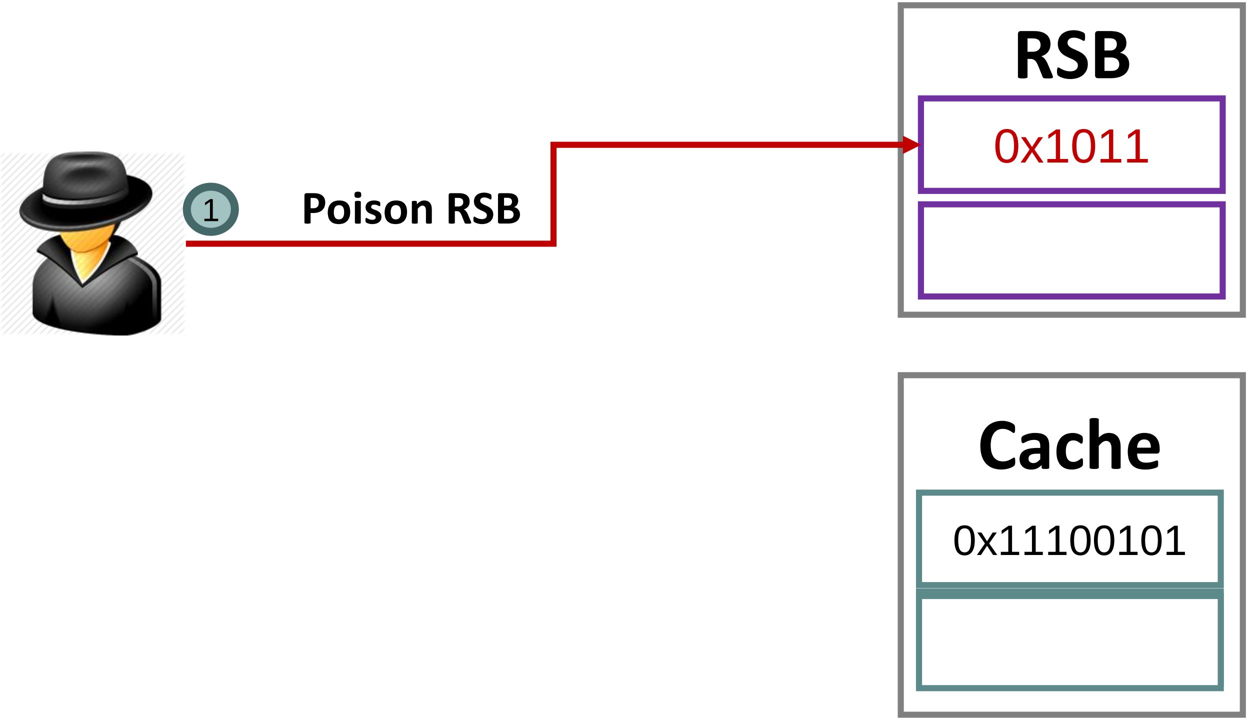
Attack 2: Across different threads/process

- **Attack setup:**
  - The attacker and victim run on a same core (Share RSB)
  - Synchronize threads using futex operations to control their interleaving

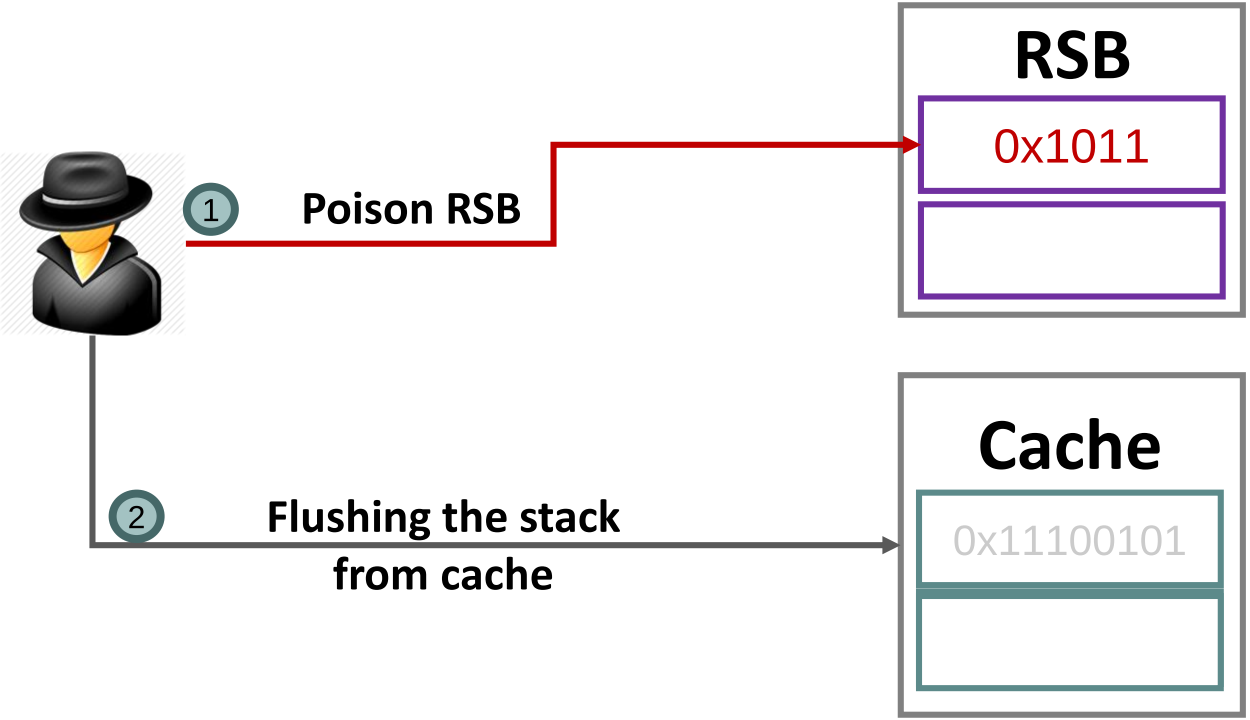
Attack 2.a: Colluding threads (User)



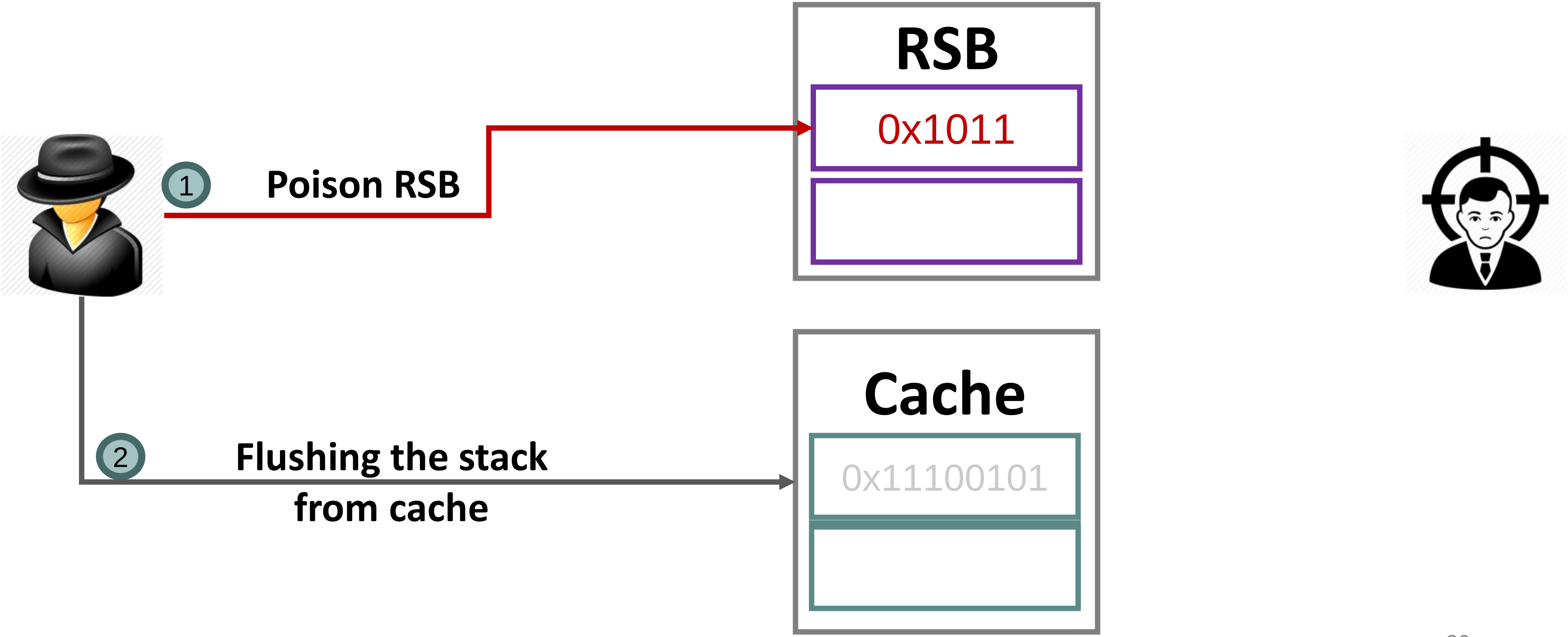
Attack 2.a: Colluding threads (User)



Attack 2.a: Colluding threads (User)



Attack 2.a: Colluding threads (User)



1 Poison RSB

RSB

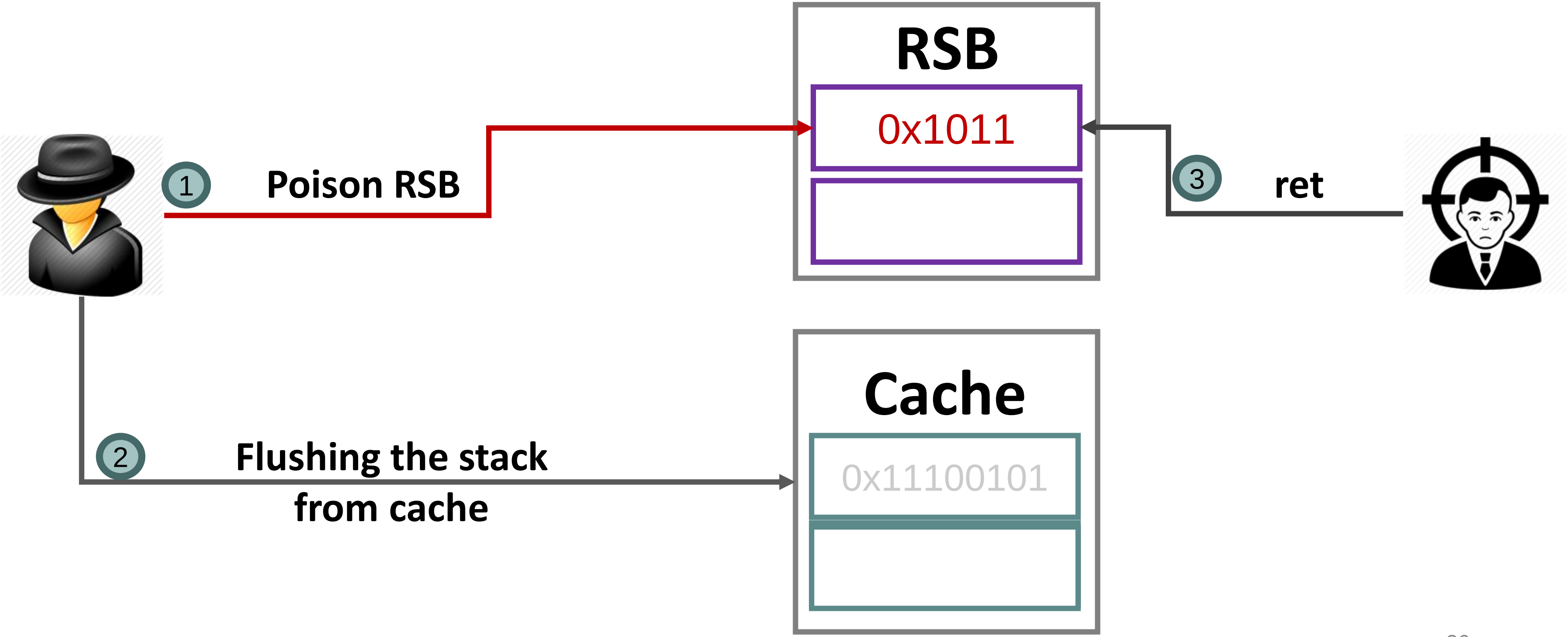
0x1011

2 Flushing the stack from cache

Cache

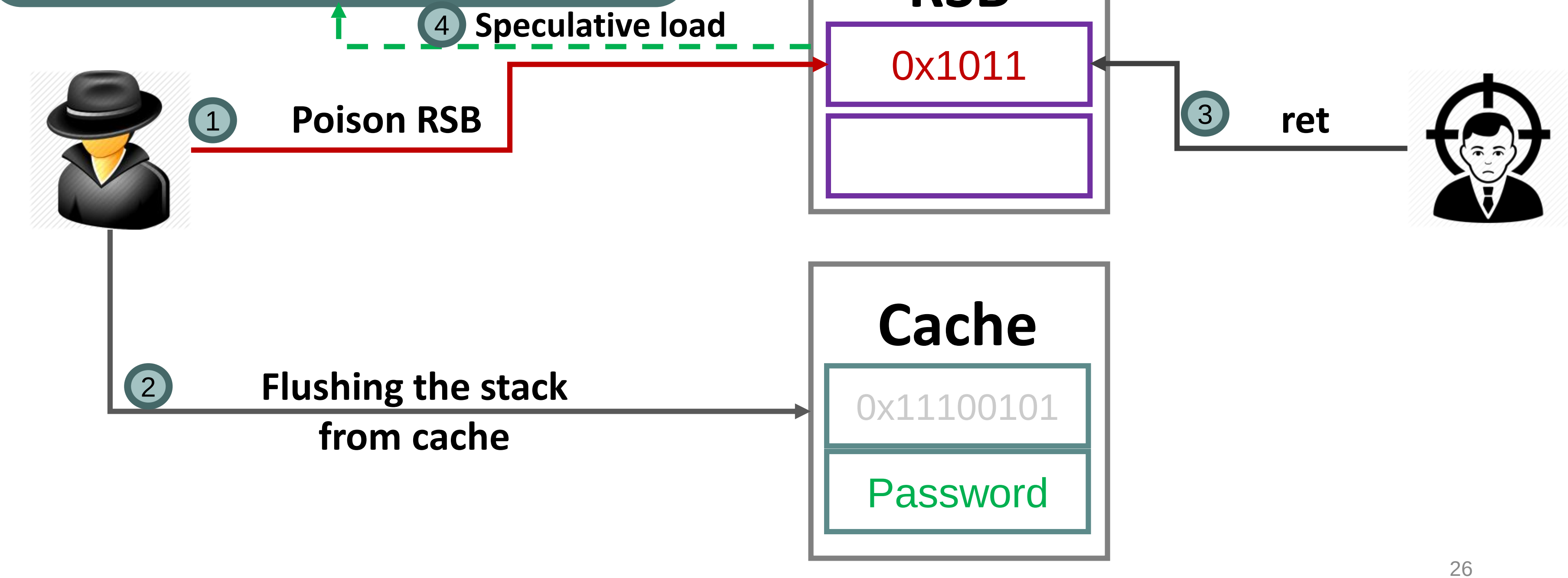
0x11100101

Attack 2.a: Colluding threads (User)



Attack 2.a: Colluding threads (User)

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



4 Speculative load

1 Poison RSB

2 Flushing the stack from cache

3 ret

RSB

0x1011

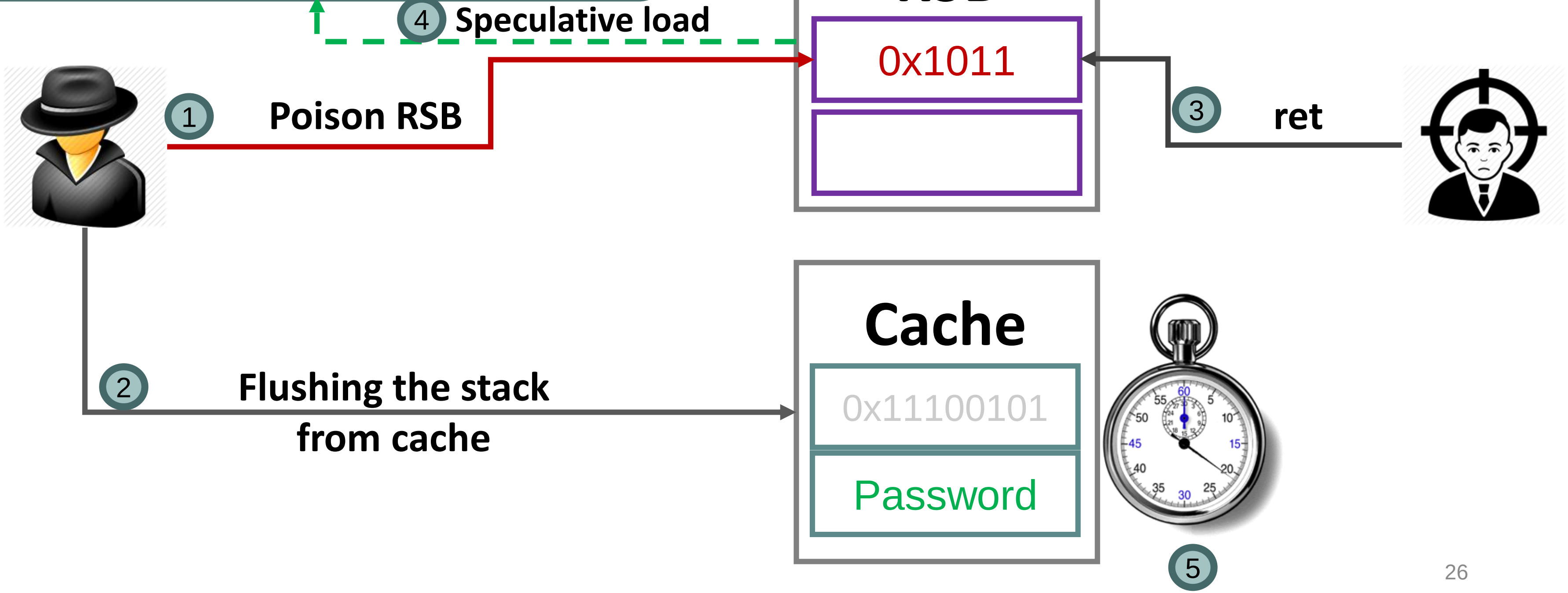
Cache

0x11100101

Password

Attack 2.a: Colluding threads (User)

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



4 Speculative load

RSB

0x1011

1 Poison RSB

3 ret

2 Flushing the stack from cache

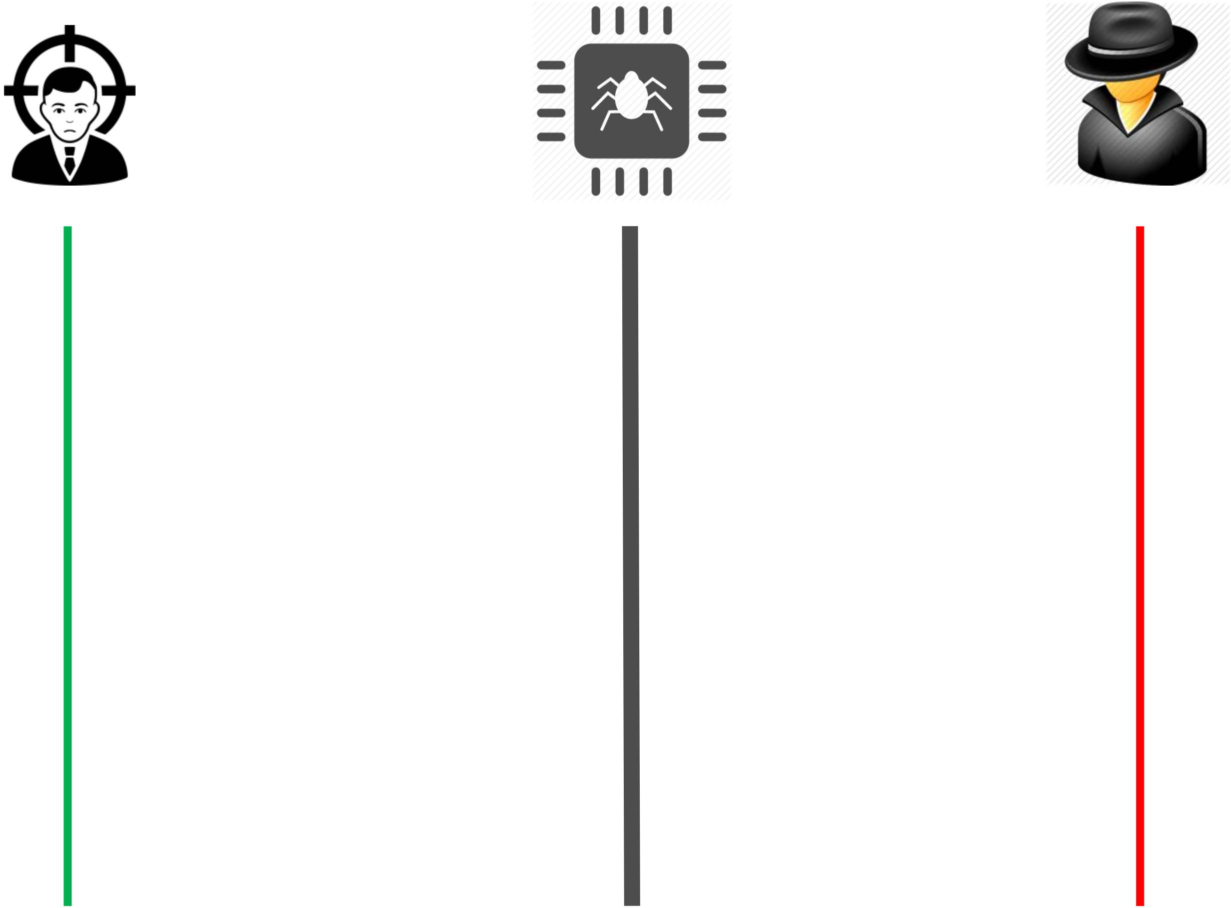
Cache

0x11100101

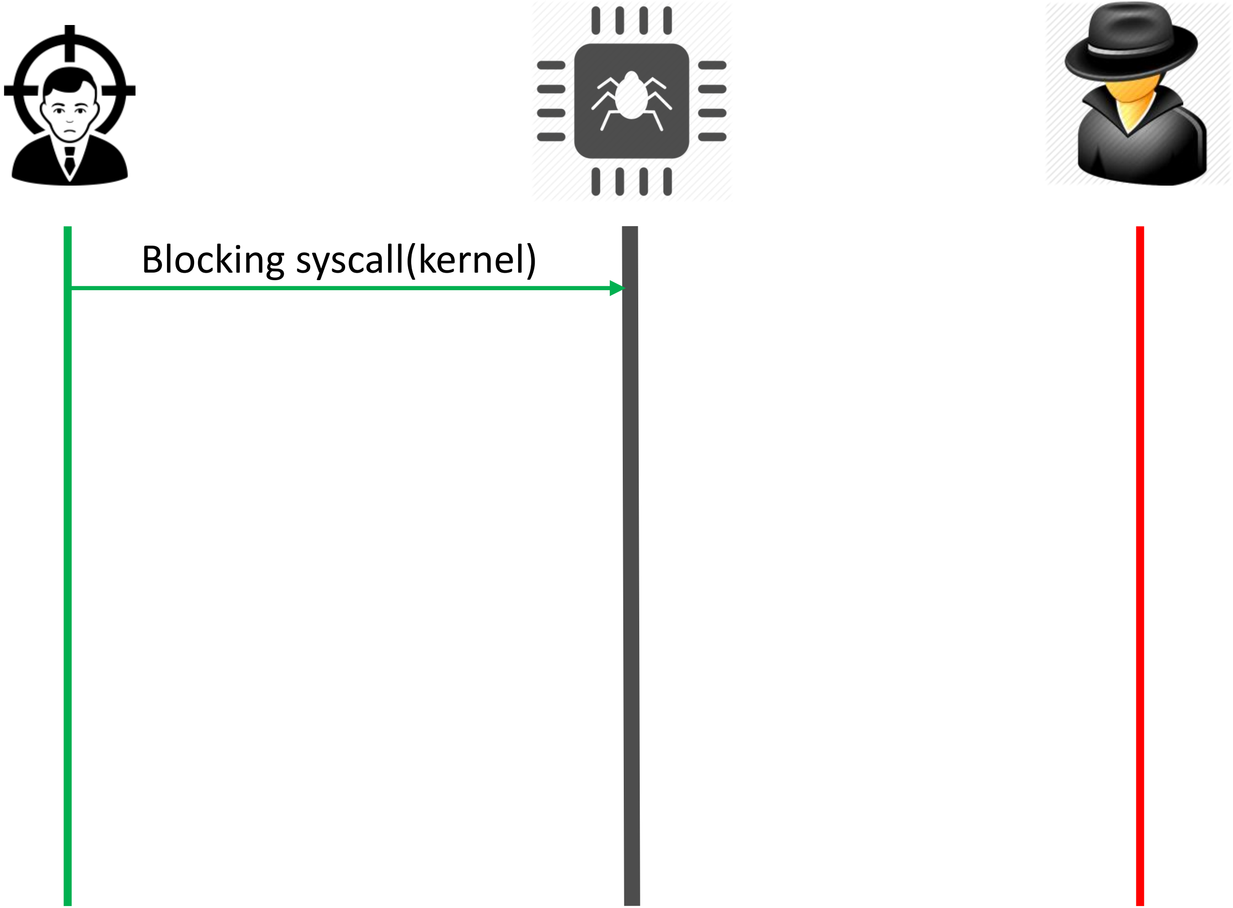
Password

5

Attack2.b: Colluding threads(kernel)

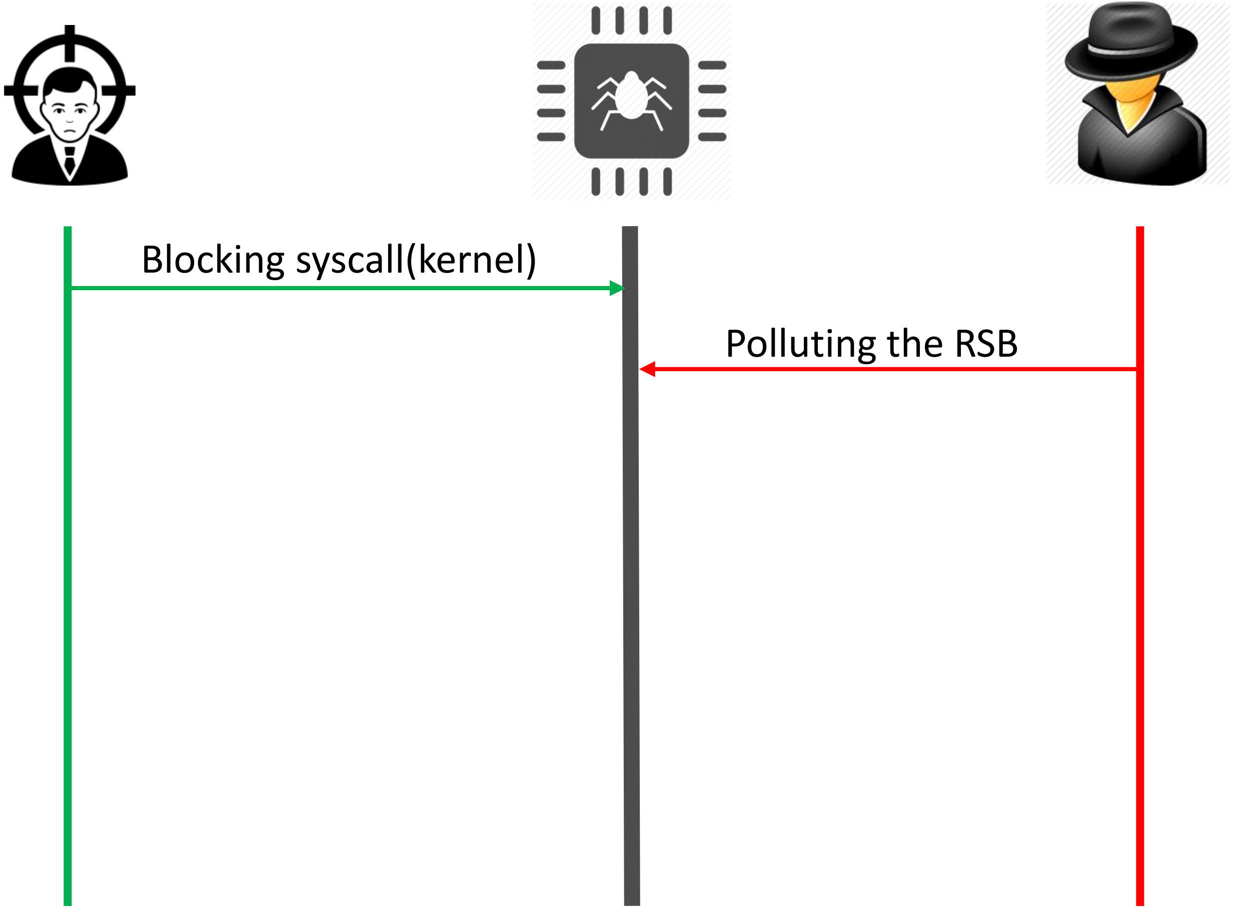


Attack2.b: Colluding threads(kernel)



Blocking syscall(kernel)

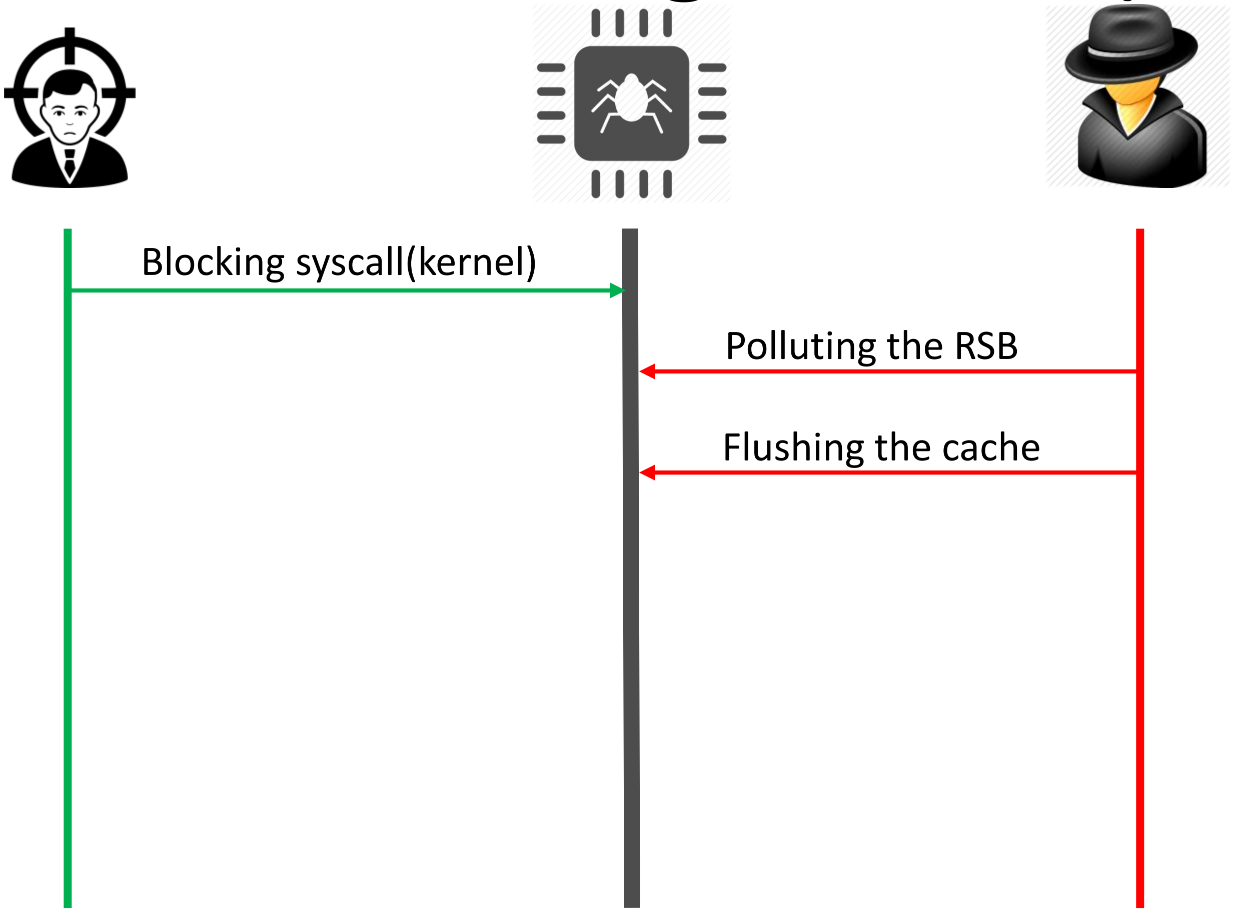
Attack2.b: Colluding threads(kernel)



Blocking syscall(kernel)

Polluting the RSB

Attack2.b: Colluding threads(kernel)

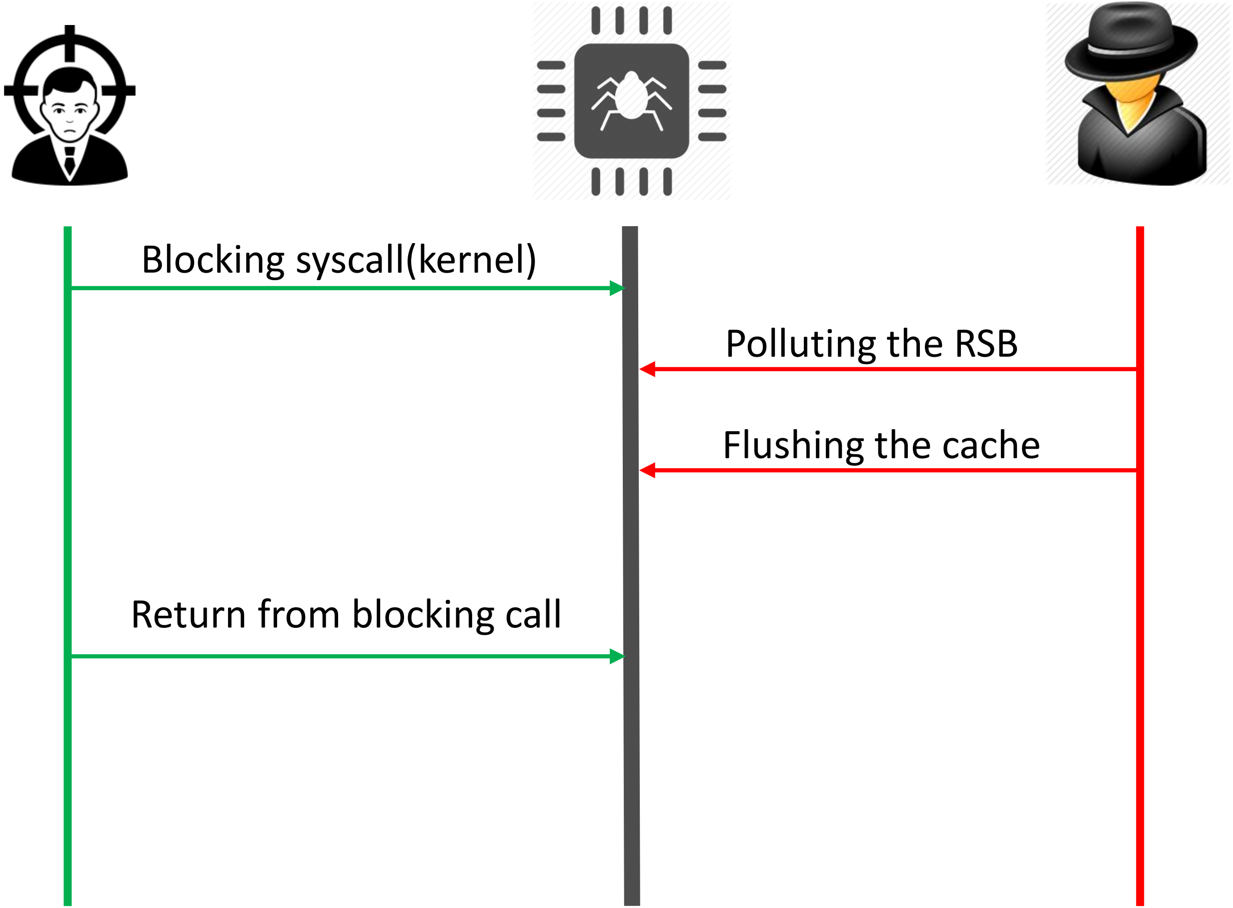


Blocking syscall(kernel)

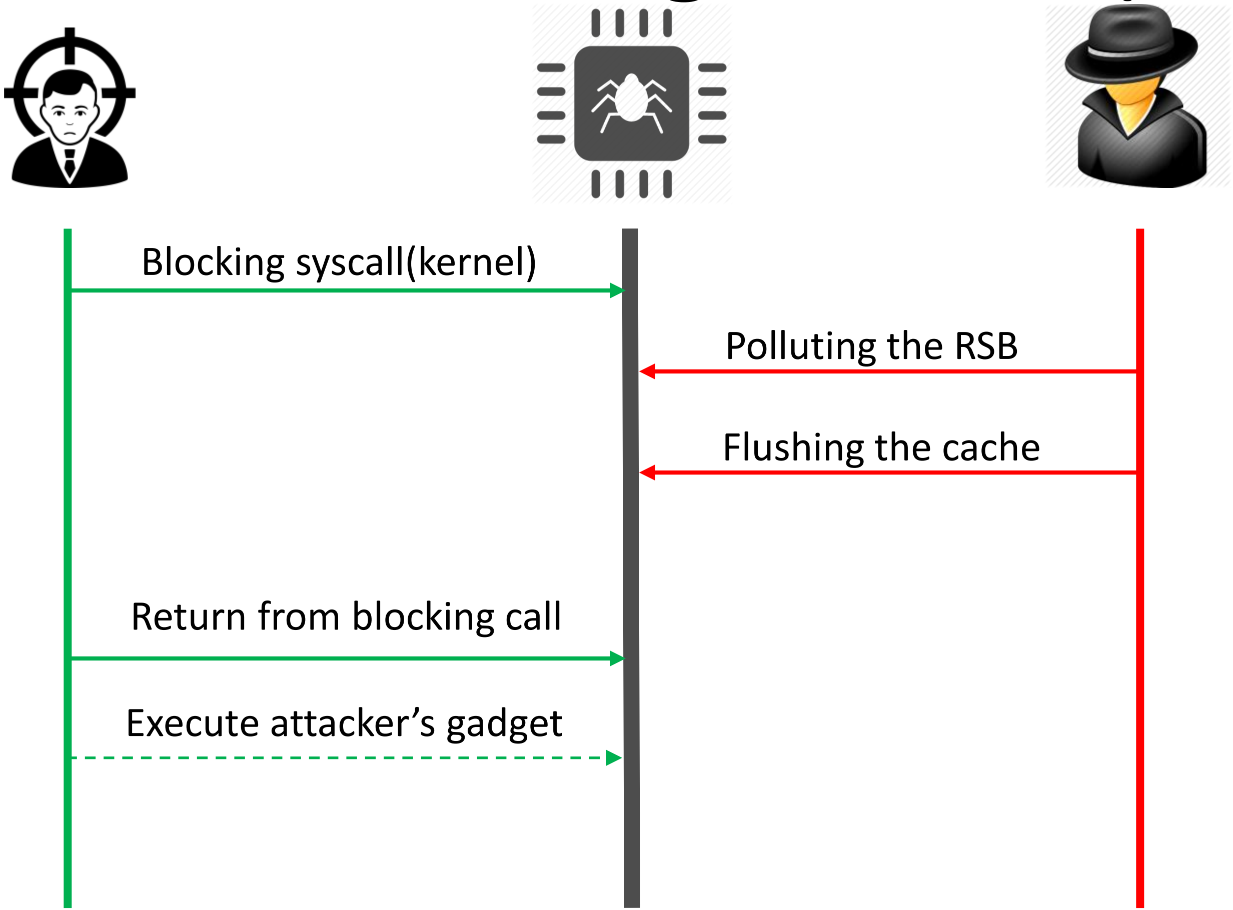
Polluting the RSB

Flushing the cache

Attack2.b: Colluding threads(kernel)



Attack2.b: Colluding threads(kernel)



Blocking syscall(kernel)

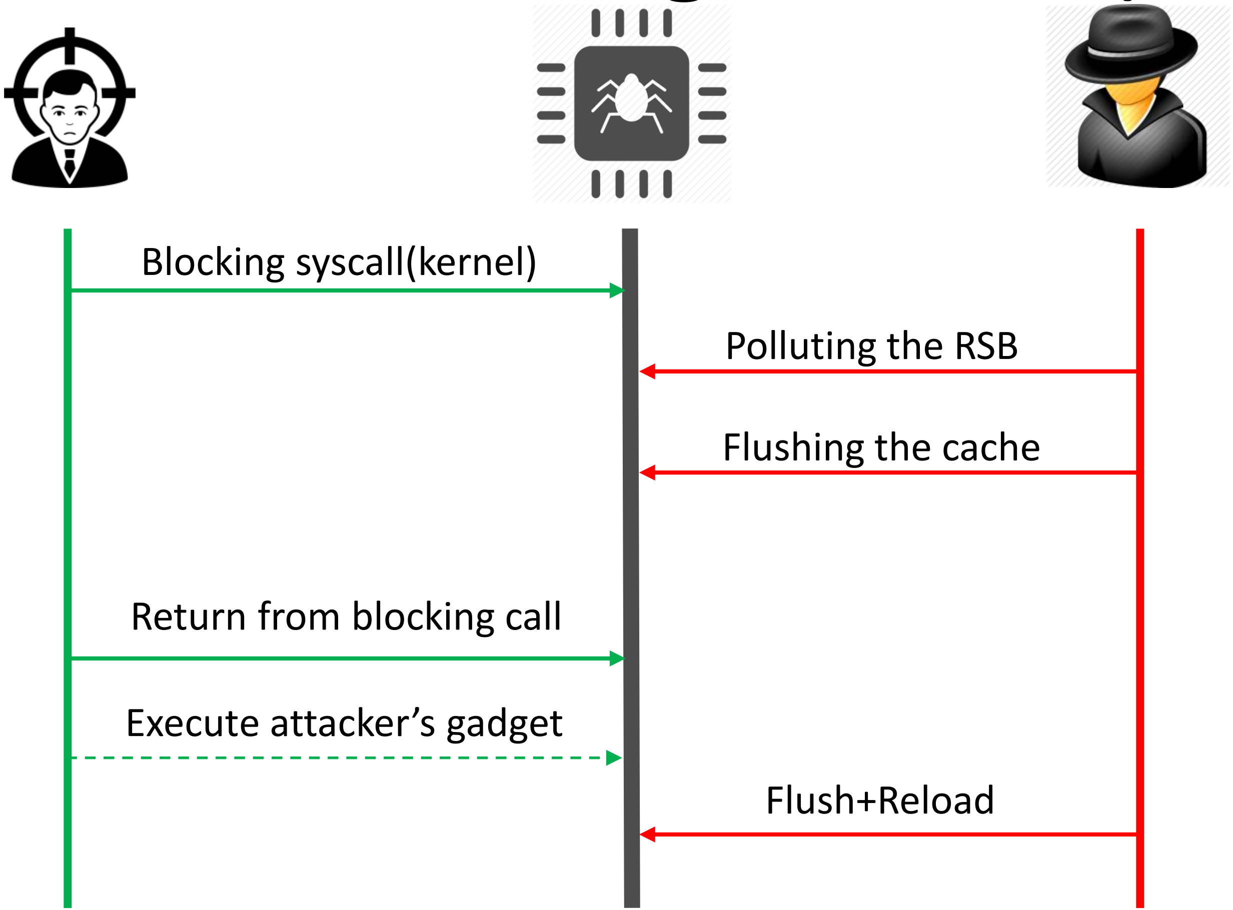
Polluting the RSB

Flushing the cache

Return from blocking call

Execute attacker’s gadget

Attack2.b: Colluding threads(kernel)



Blocking syscall(kernel)

Polluting the RSB

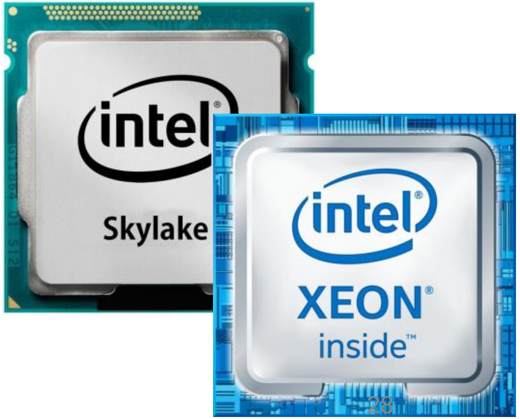
Flushing the cache

Return from blocking call

Execute attacker’s gadget

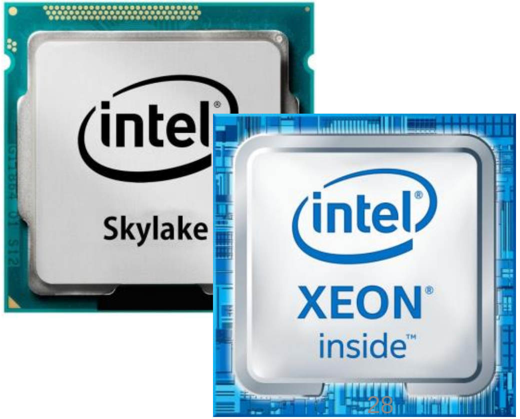
Flush+Reload

Discussion on Attack 2



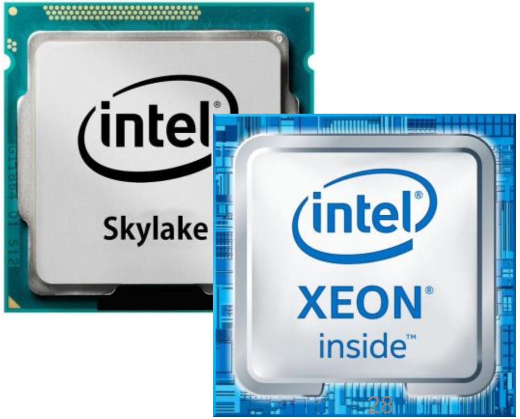
Discussion on Attack 2

- **RSB Refilling**



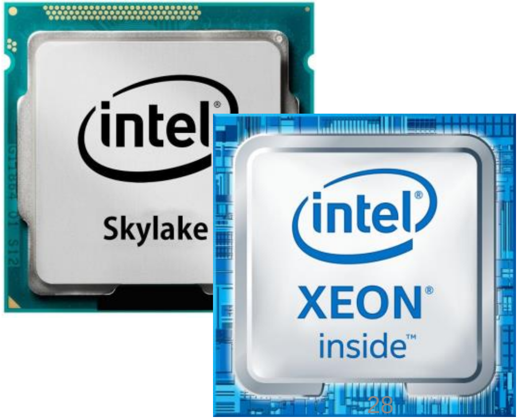
Discussion on Attack 2

- **RSB Refilling**
  - Linux has developed it for **Skylake+** processors



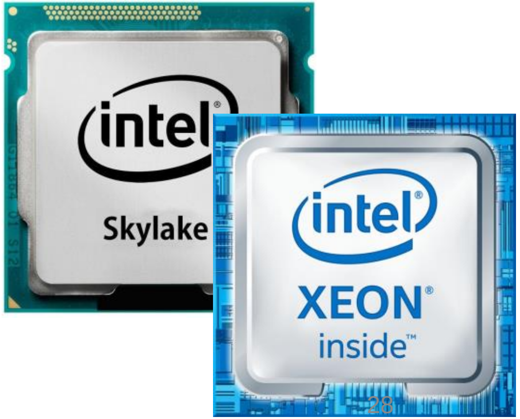
Discussion on Attack 2

- **RSB Refilling**
  - Linux has developed it for **Skylake+** processors ❌



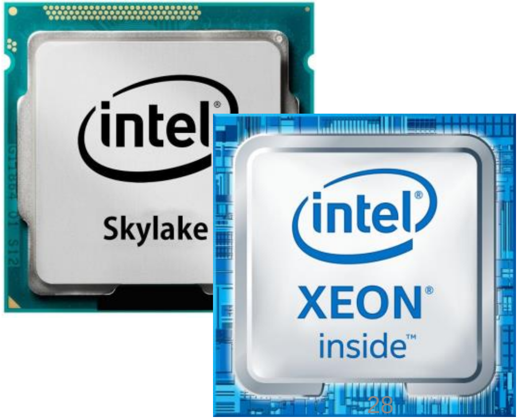
Discussion on Attack 2

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



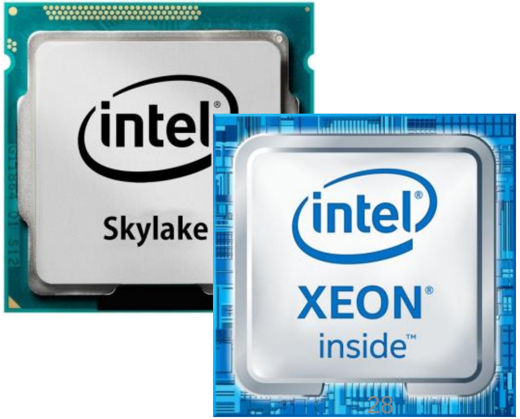
Discussion on Attack 2

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



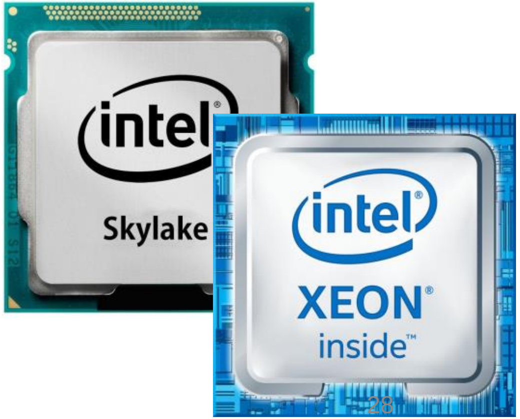
Discussion on Attack 2

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



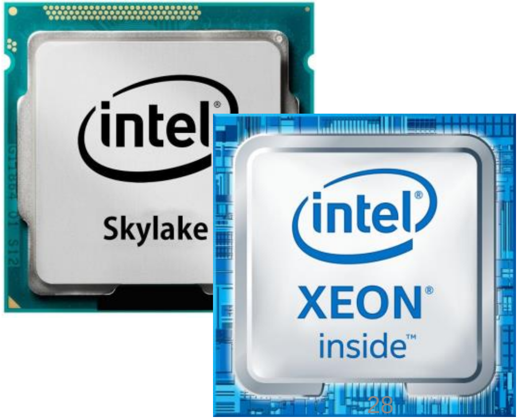
Discussion on Attack 2

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



Discussion on Attack 2

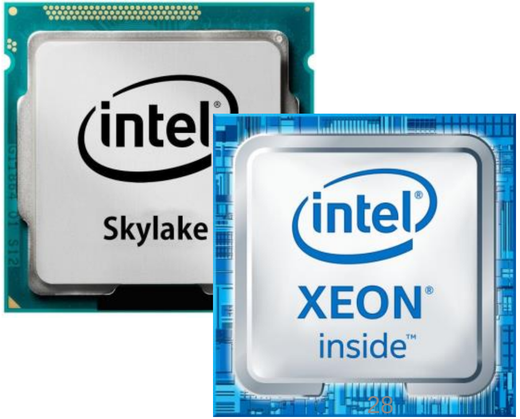
- **RSB Refilling**
  - Linux has developed it for **Skylake+** processors ✗
  - Xeon and older processor ✓
  - Microsoft windows does not implement it ✓



Discussion on Attack 2

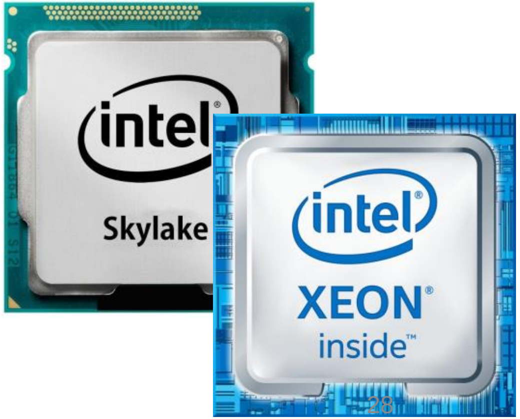
- **RSB Refilling**
  - Linux has developed it for **Skylake+** processors ✗
  - Xeon and older processor ✓
  - Microsoft windows does not implement it ✓

- 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**



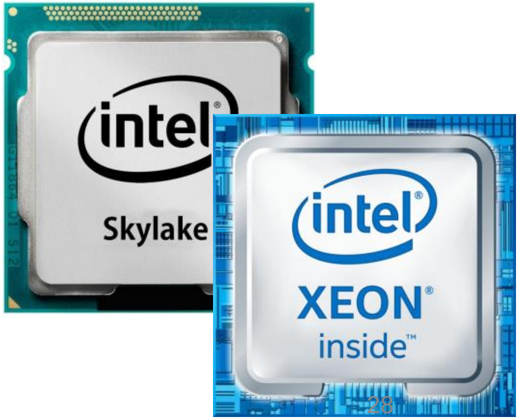
Discussion on Attack 2

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

Windows

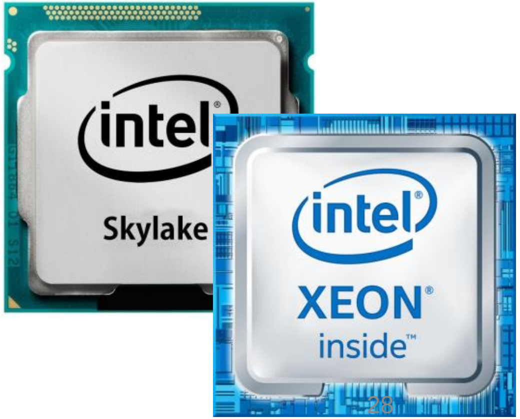
- Update: linux-mainline released a new patch based on our suggestion to refill RSB unconditionally

- **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

Attack on SGX

- Reveal Data from SGX enclave
- Triggering an unmatched return
- IBPB prevent it based on the new contact with Intel engineer.

Attack on other process

- Run on the same core
- Need to know the address of victim’s stack
- Bypassing ASLR
- RSB refilling/IBPB may stop the attack

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?