Spectre Attacks: Exploiting Speculative Execution

Paul Kocher [1] , Daniel Genkin [2] , Daniel Gruss [3] , Werner Haas [4] , Mike Hamburg [5] ,

Moritz Lipp [3] , Stefan Mangard [3] , Thomas Prescher [4] , Michael Schwarz [3] , Yuval Yarom [6]

1 Independent

2 University of Pennsylvania and University of Maryland

3 Graz University of Technology

4 Cyberus Technology

5 Rambus, Cryptography Research Division

6 University of Adelaide and Data61

40th IEEE Symposium on Security and Privacy

Outline

- Attack description
- Background
- Spectre attack
- Spectre variations
- Mitigation options
- Strengths
- Weaknesses
- Discussion

Attack description

Attack description

Attack description

- Spectre exploit gives the attacker the ability to read out the memory of a bug-free victim process

Attack description

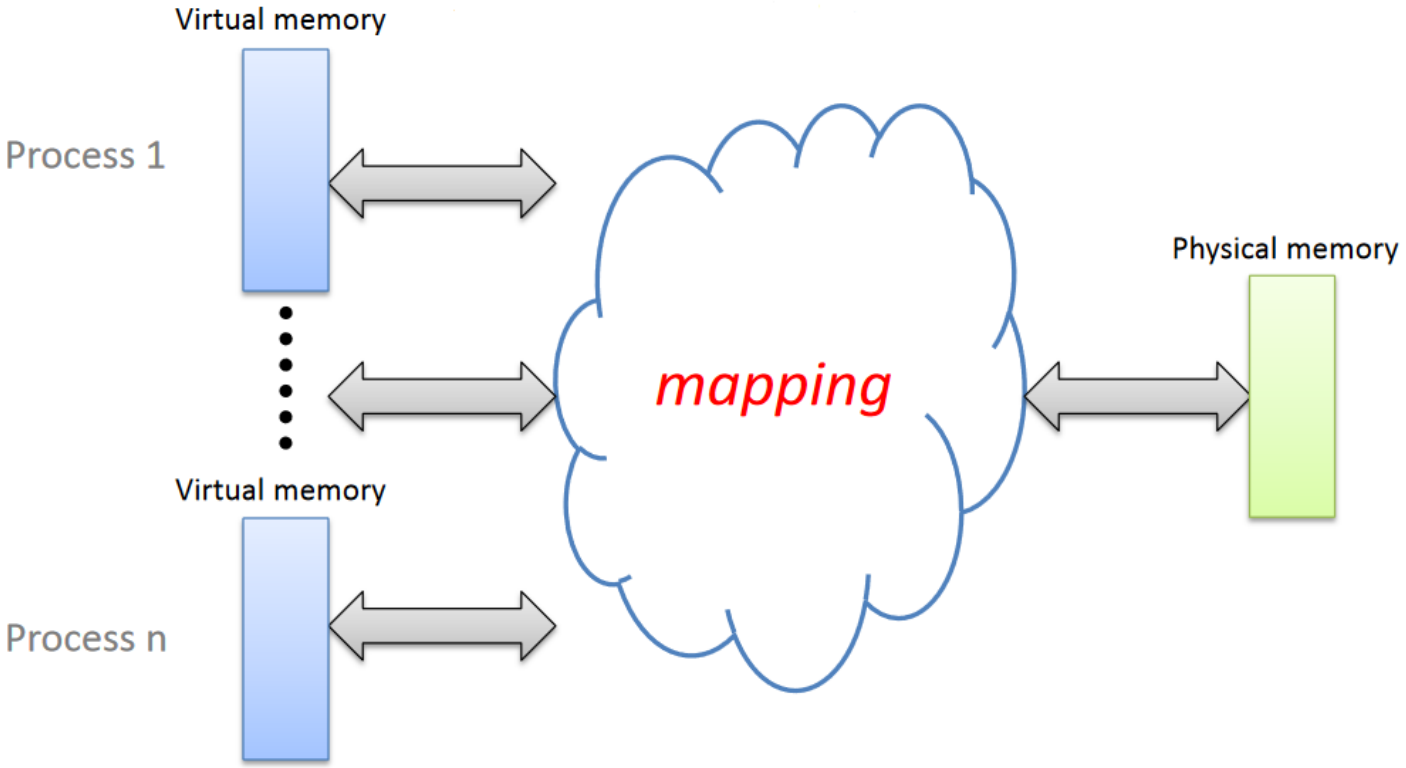
- Spectre exploit gives the attacker the ability to read out the memory of a bug-free victim process
- Works on Intel, AMD and ARM

Attack description

- Spectre exploit gives the attacker the ability to read out the memory of a bug-free victim process
- Works on Intel, AMD and ARM
- How?

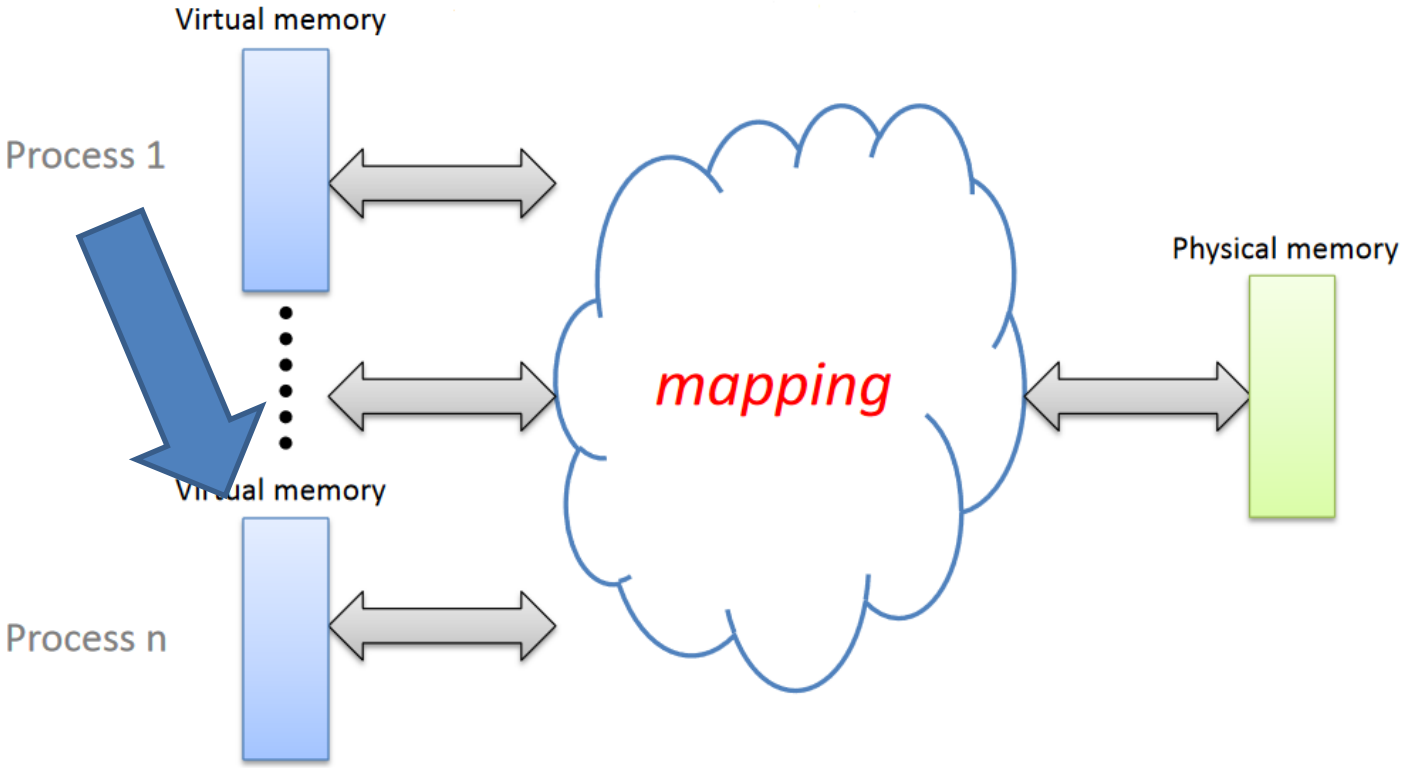
Background

Background: Virtual Memory



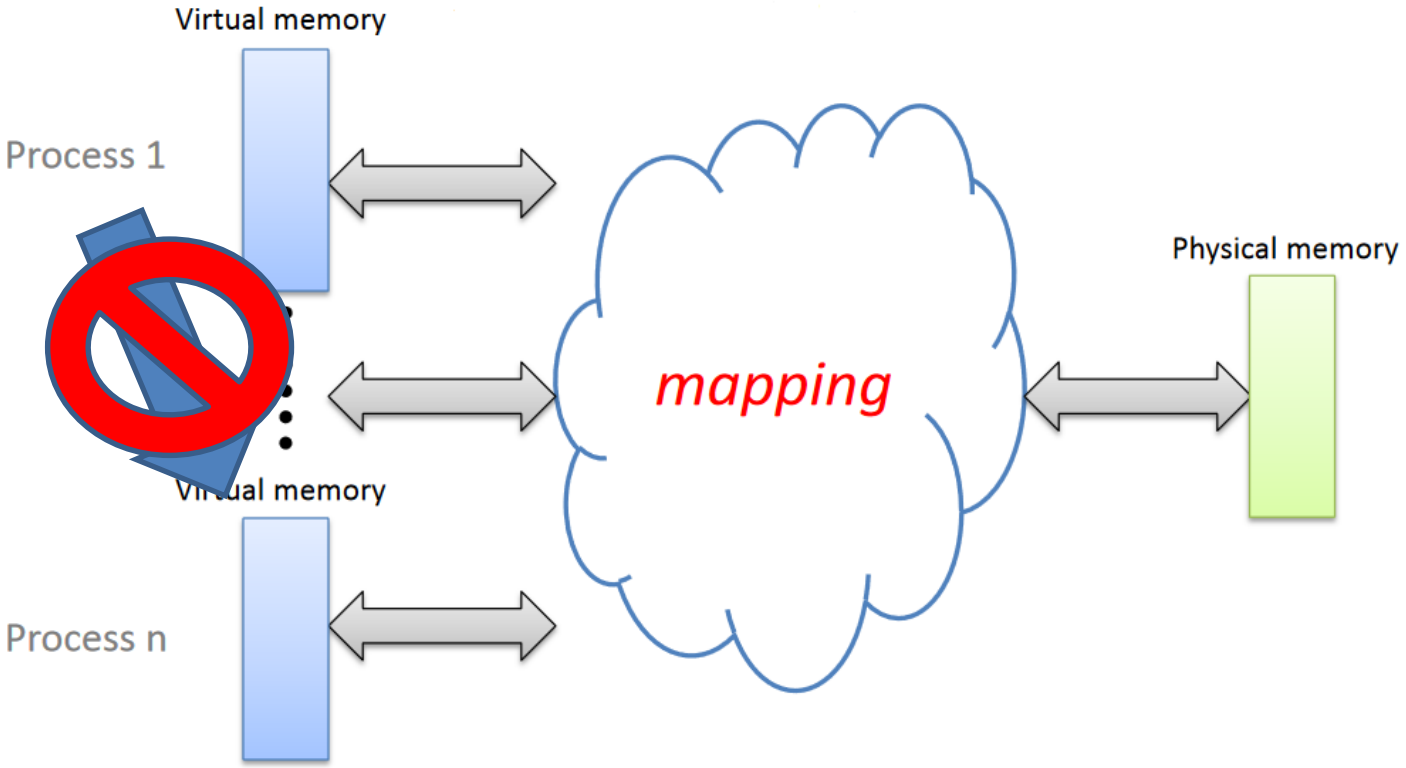
https://svn.inf.ethz.ch/svn/systems/SPCA2017/trunk/lectures/18-VirtualMemory-upload.pdf

Background: Virtual Memory



https://svn.inf.ethz.ch/svn/systems/SPCA2017/trunk/lectures/18-VirtualMemory-upload.pdf

Background: Virtual Memory



https://svn.inf.ethz.ch/svn/systems/SPCA2017/trunk/lectures/18-VirtualMemory-upload.pdf

Background: Covert channel

Background: Covert channel

- When two processes cooperate to communicate, not by architecturally defined means but by changing the microarchitectural state in a suitable way

Background: Covert channel

- When two processes cooperate to communicate, not by architecturally defined means but by changing the microarchitectural state in a suitable way
- Example of state that can be used:

Background: Covert channel

- When two processes cooperate to communicate, not by architecturally defined means but by changing the microarchitectural state in a suitable way
- Example of state that can be used:
  - Cache timing

Background: Covert channel

- When two processes cooperate to communicate, not by architecturally defined means but by changing the microarchitectural state in a suitable way
- Example of state that can be used:
  - Cache timing
  - Instruction timing

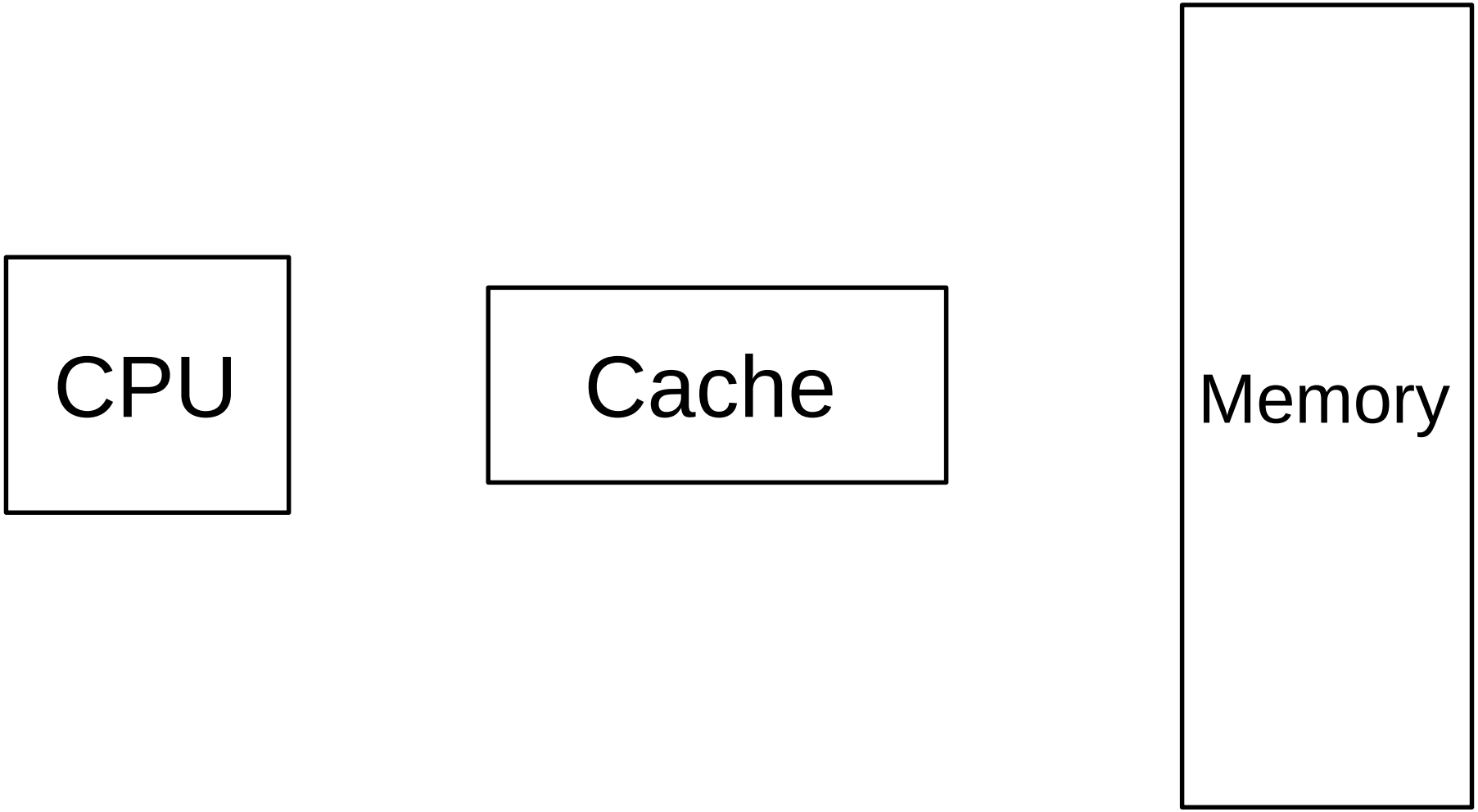
Background: Covert channel

- When two processes cooperate to communicate, not by architecturally defined means but by changing the microarchitectural state in a suitable way
- Example of state that can be used:
  - Cache timing
  - Instruction timing
  - ALU contention

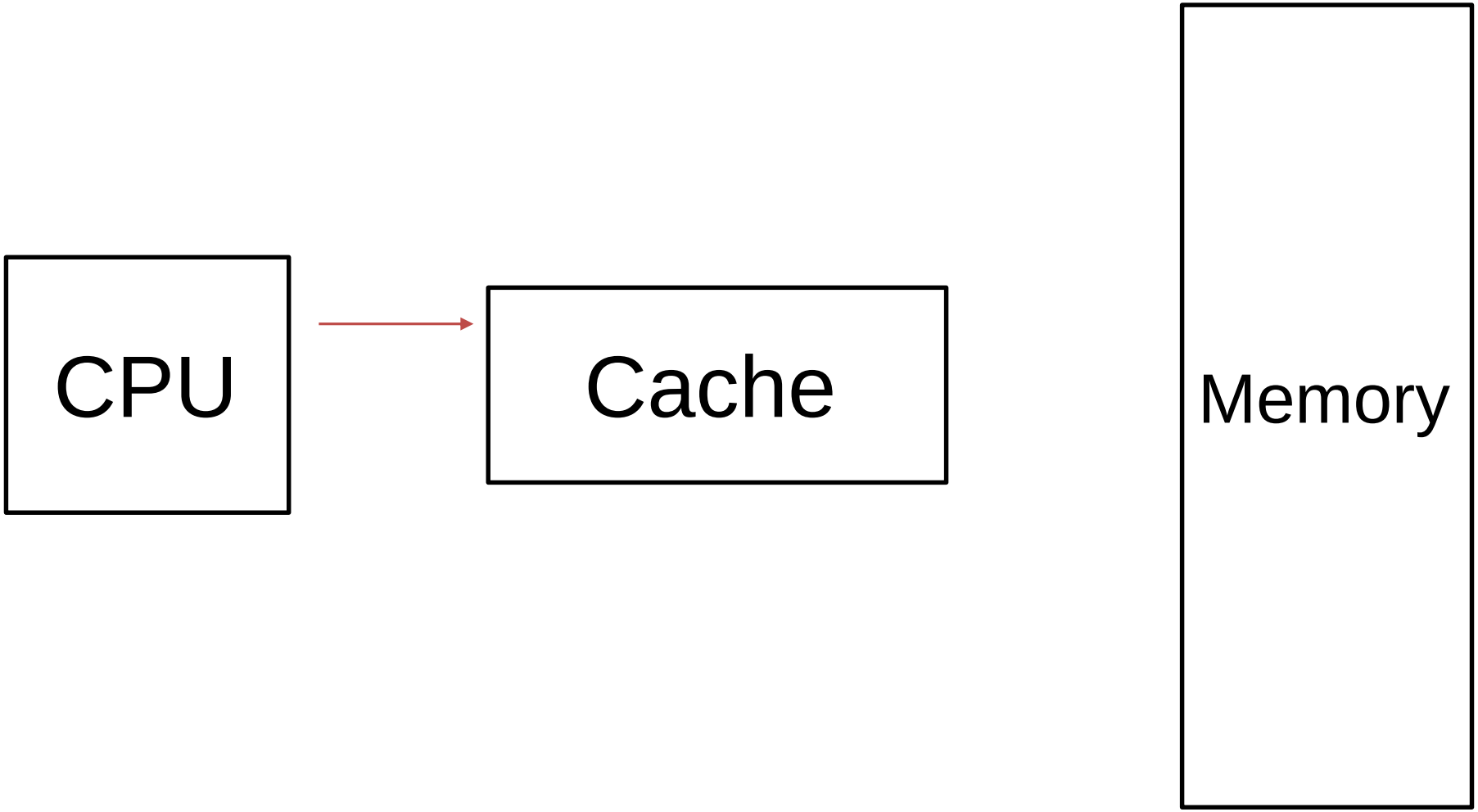
Background: Covert channel

- When two processes cooperate to communicate, not by architecturally defined means but by changing the microarchitectural state in a suitable way
- Example of state that can be used:
  - Cache timing
  - Instruction timing
  - ALU contention
  - Memory contention

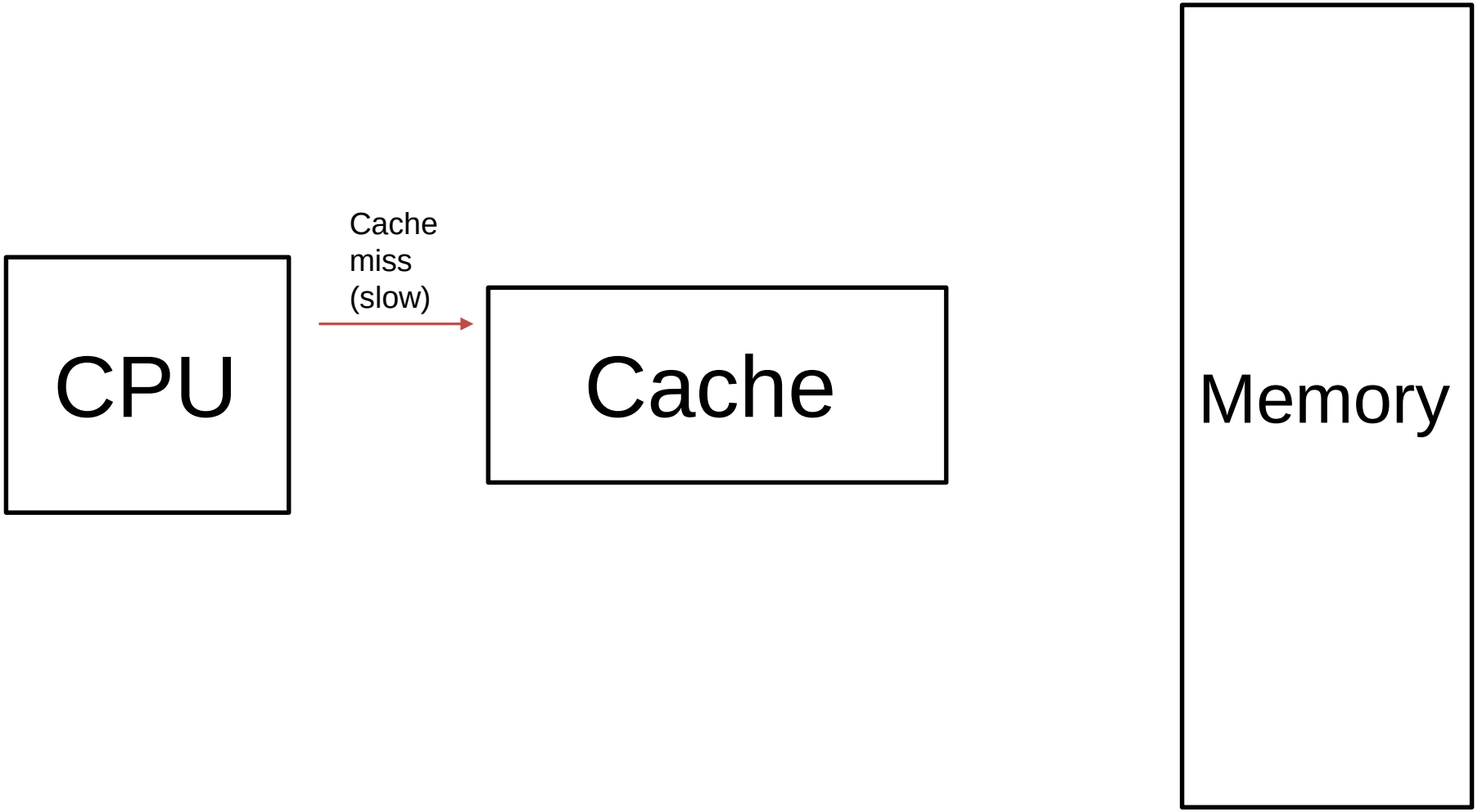
Background: Cache



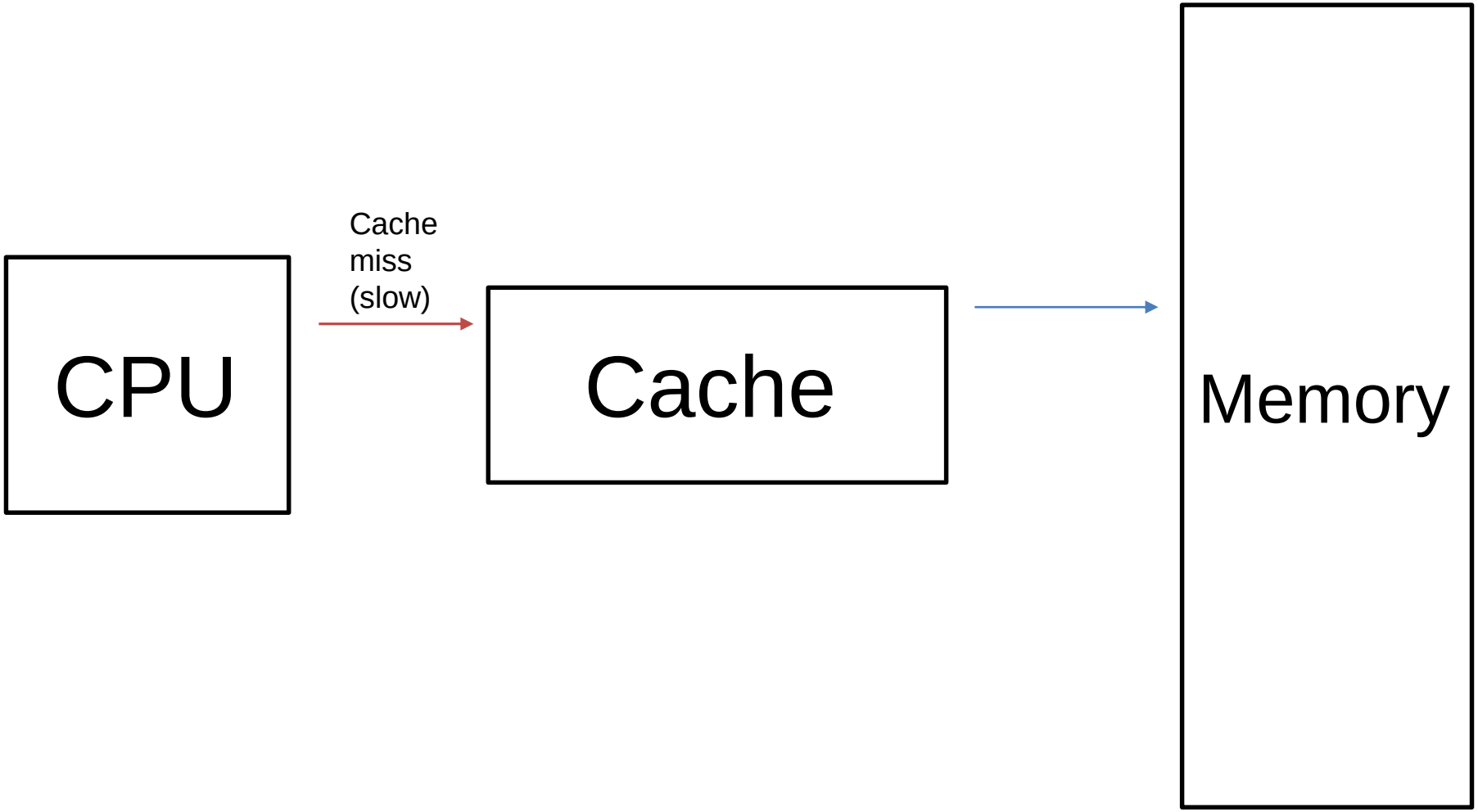
Background: Cache



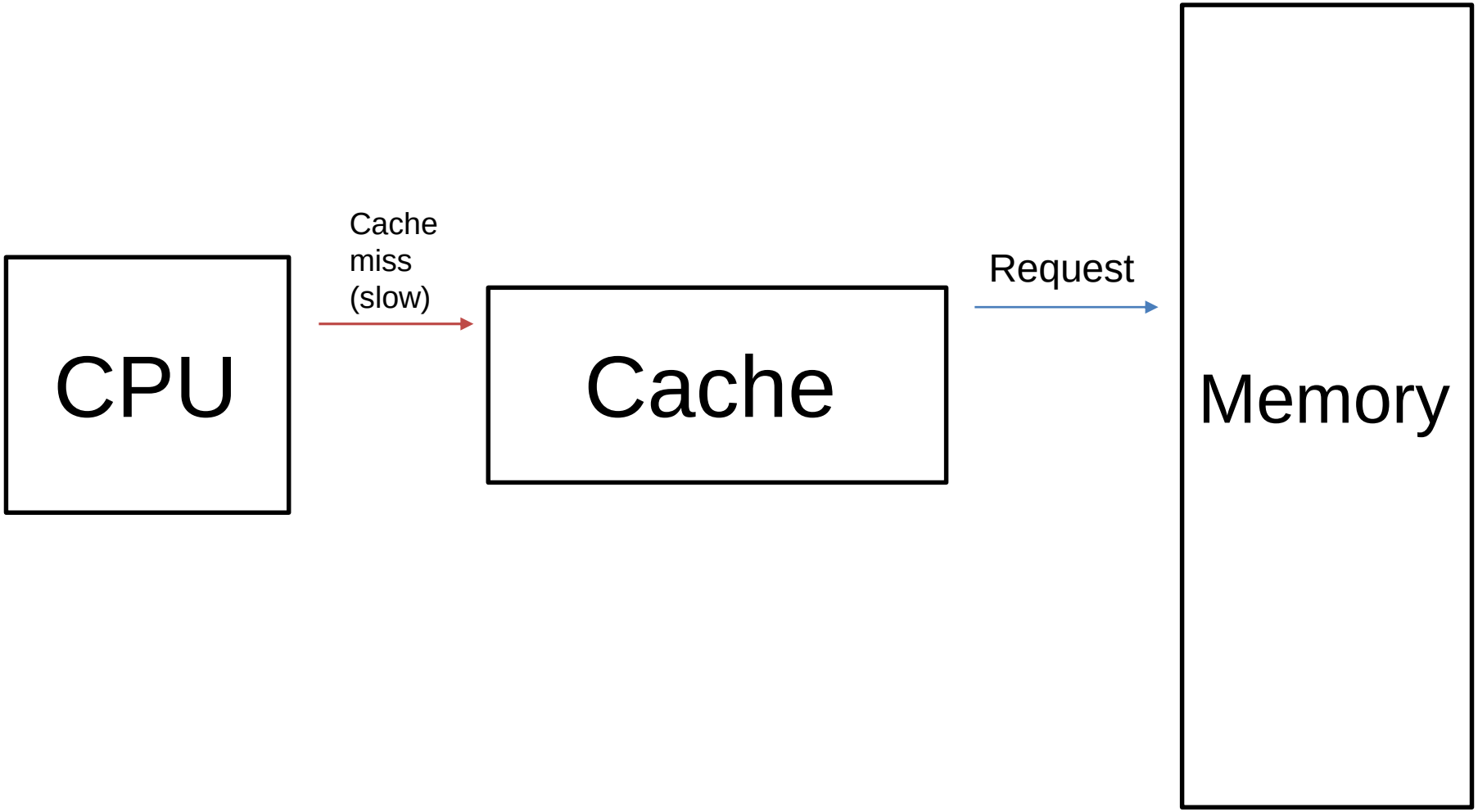
Background: Cache



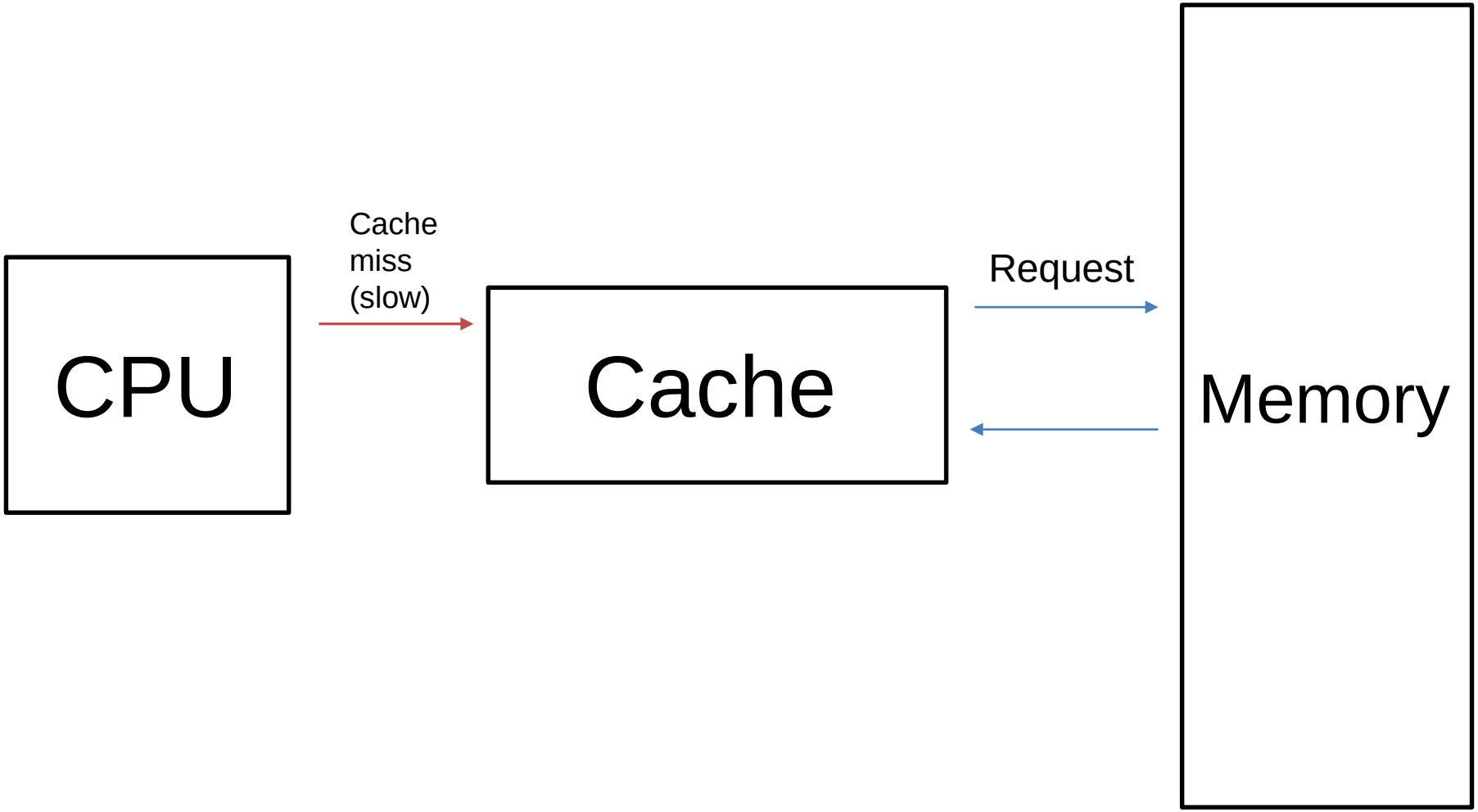
Background: Cache



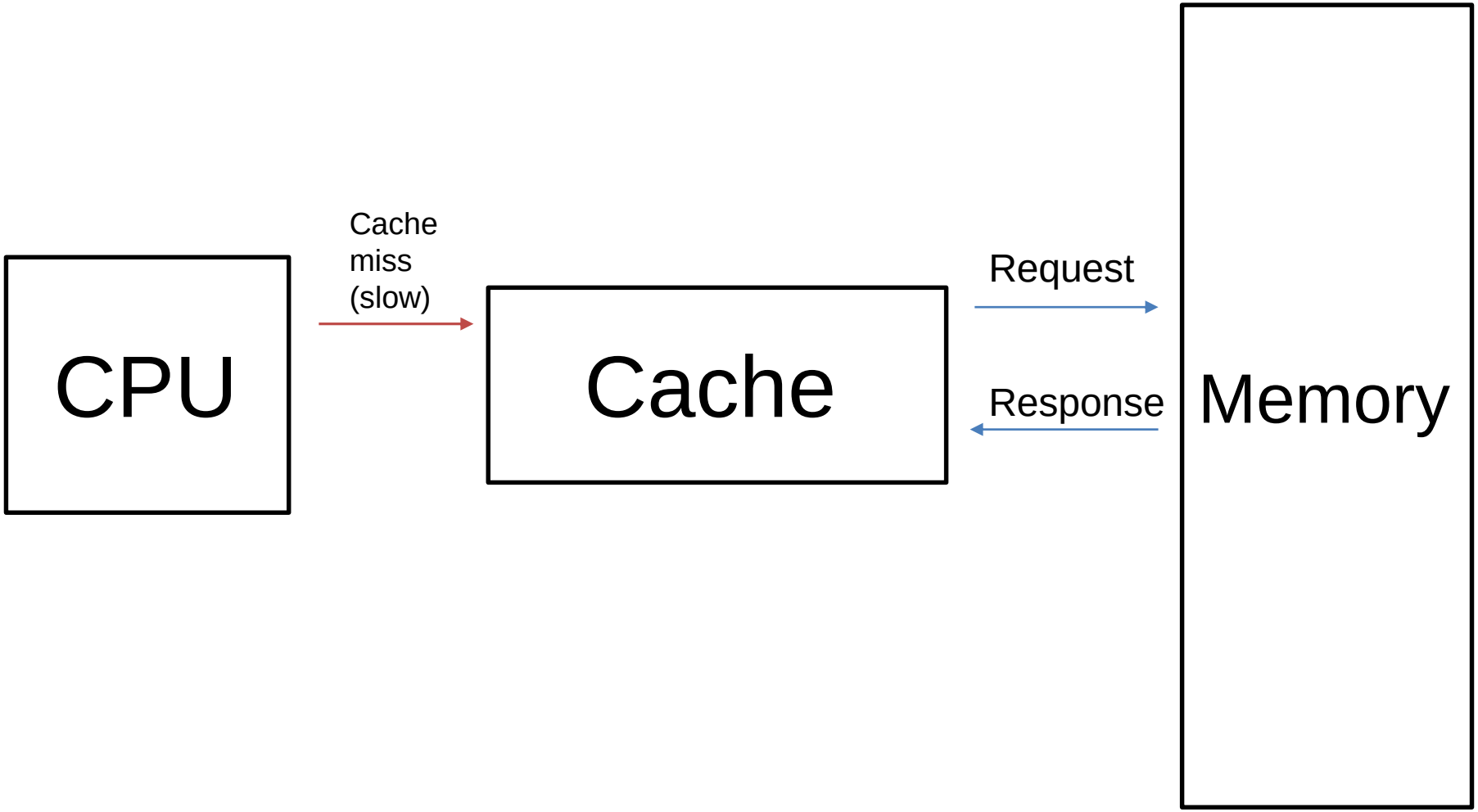
Background: Cache



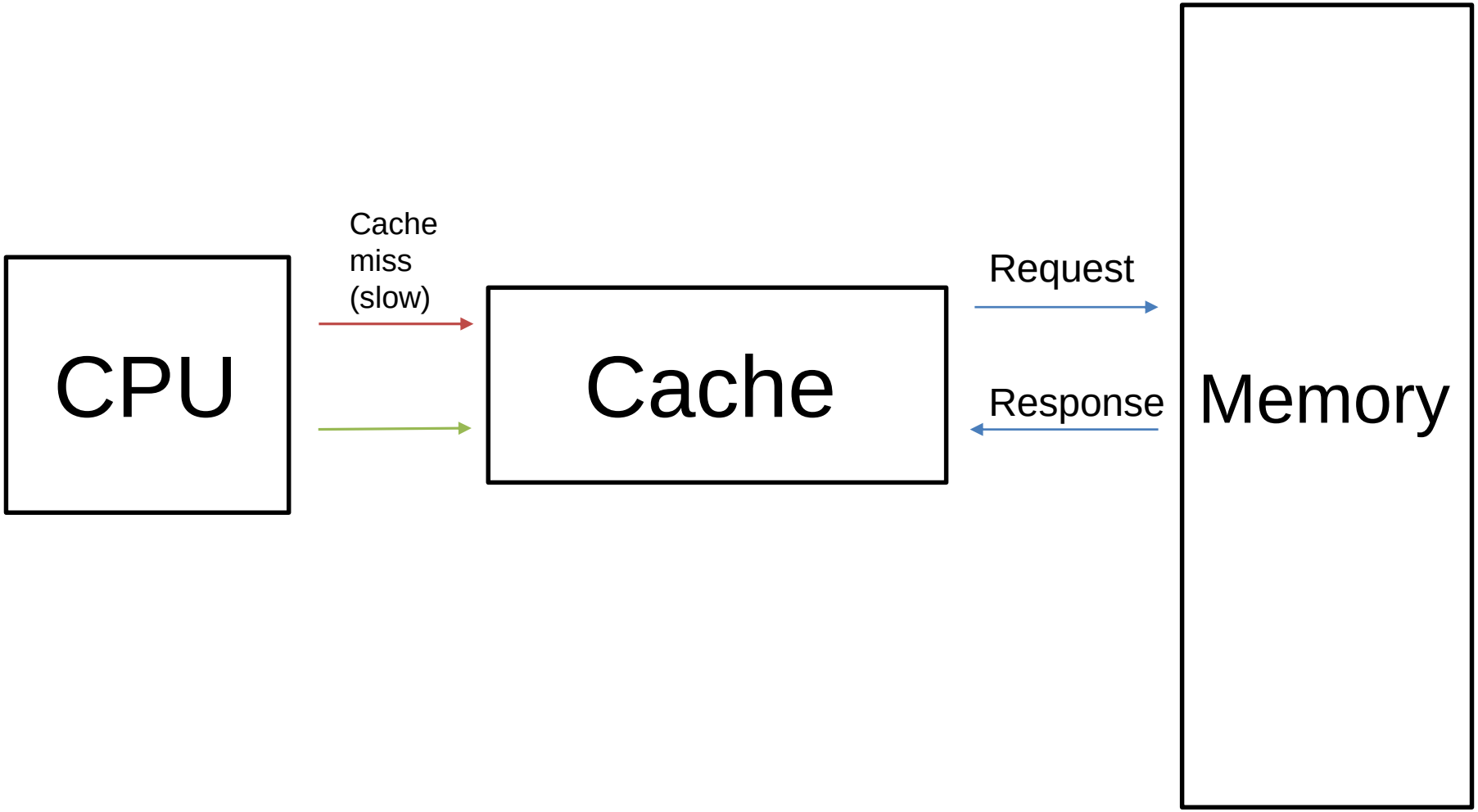
Background: Cache



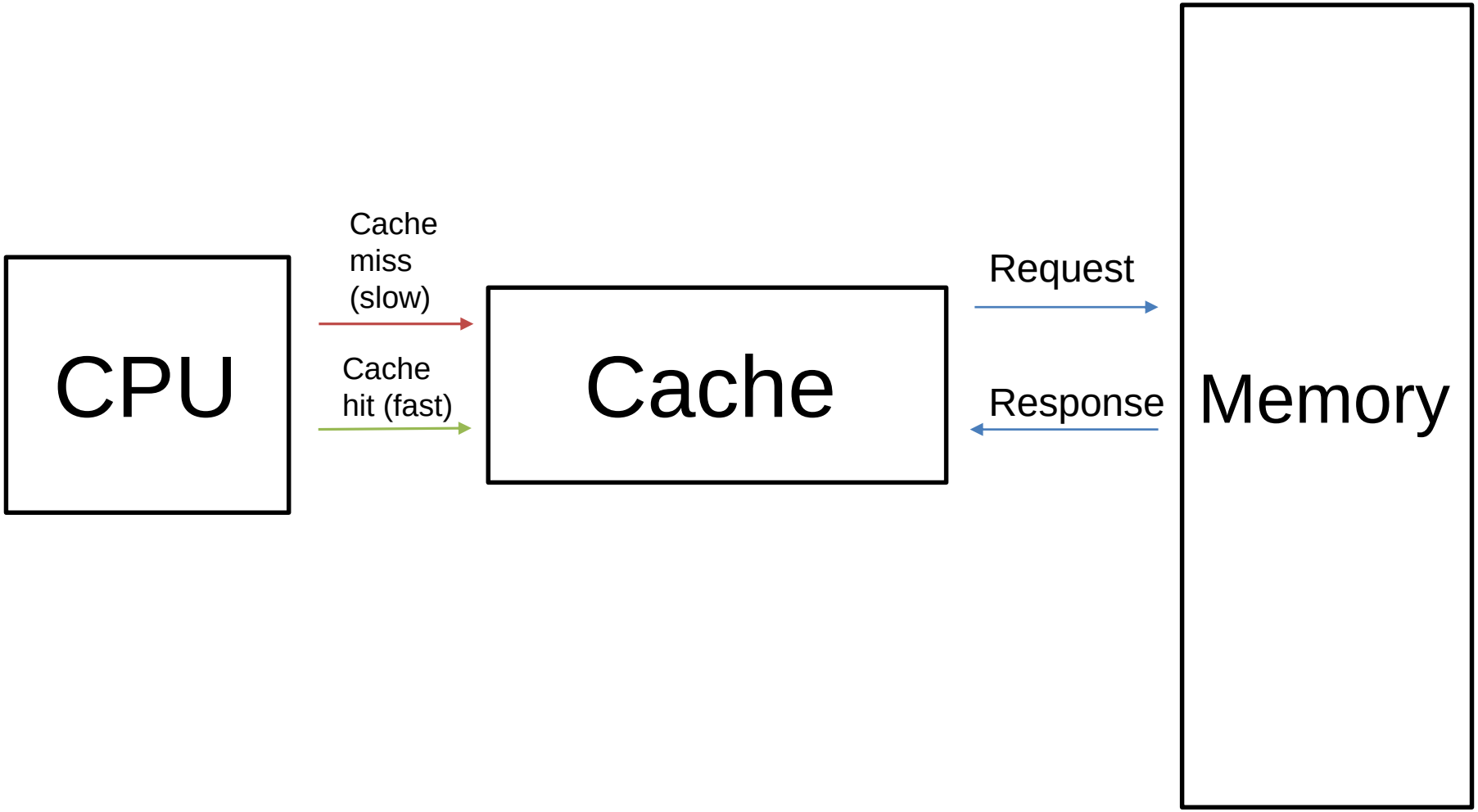
Background: Cache



Background: Cache



Background: Cache



Background: Covert channel

Background: Covert channel

- Example: cache timing as covert channel

Background: Covert channel

- Example: cache timing as covert channel
  - Sender process has a value it wants to transmit to the receiver process

Background: Covert channel

- Example: cache timing as covert channel
  - Sender process has a value it wants to transmit to the receiver process
  - Sender changes the cache (loading, evicting) in a value-dependent way

Background: Covert channel

- Example: cache timing as covert channel
  - Sender process has a value it wants to transmit to the receiver process
  - Sender changes the cache (loading, evicting) in a value-dependent way
  - Receiver can’t see the value in the cache directly but can time the cache and thus infer the value

Background: Speculative Execution

Background: Speculative Execution

- Predicting/Speculating for example:

Background: Speculative Execution

- Predicting/Speculating for example:
  - Prefetcher (what will be needed in the future)

Background: Speculative Execution

- Predicting/Speculating for example:
  - Prefetcher (what will be needed in the future)
  - Branch Predictor (Speculate if direct branch taken or not)

Background: Speculative Execution

- Predicting/Speculating for example:
  - Prefetcher (what will be needed in the future)
  - Branch Predictor (Speculate if direct branch taken or not)
  - Branch Target Buffer/BTB (Speculate what a value will be)

Background: Speculative Execution

- Predicting/Speculating for example:
  - Prefetcher (what will be needed in the future)
  - Branch Predictor (Speculate if direct branch taken or not)
  - Branch Target Buffer/BTB (Speculate what a value will be)
- Leads to improved Instruction Level Parallelism

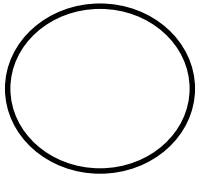
Background: Speculative Execution

- Predicting/Speculating for example:
  - Prefetcher (what will be needed in the future)
  - Branch Predictor (Speculate if direct branch taken or not)
  - Branch Target Buffer/BTB (Speculate what a value will be)
- Leads to improved Instruction Level Parallelism
- Otherwise CPU would have to sit idle while waiting for results

Background: Speculative Execution

- Branch predictor

```
if (slow condition){
    //do something
} else {
    //do something
}
```

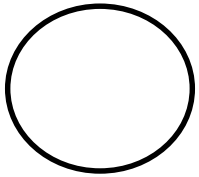


Branch predictor

Background: Speculative Execution

- Branch predictor

```
if (slow condition){
    //do something
} else {
    //do something
}
```



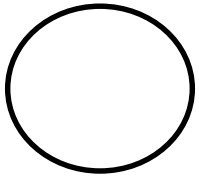
Branch predictor

Background: Speculative Execution

- Branch predictor

Start evaluating condition

```
if (slow condition){
    //do something
} else {
    //do something
}
```

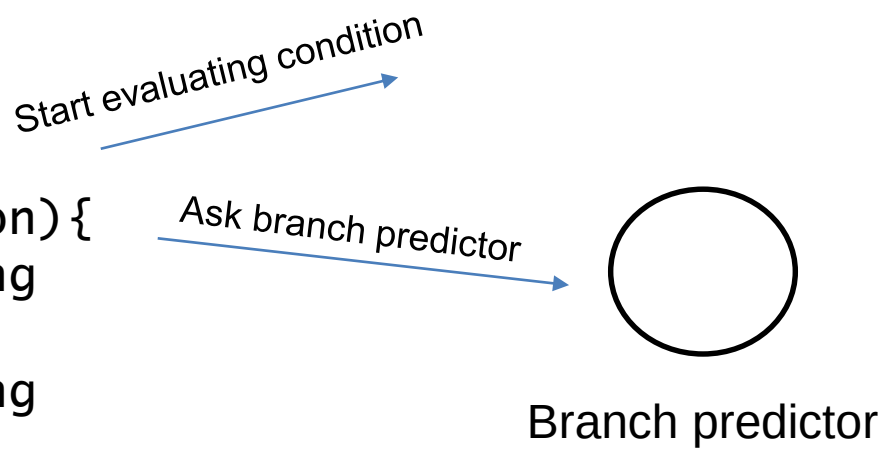


Branch predictor

Background: Speculative Execution

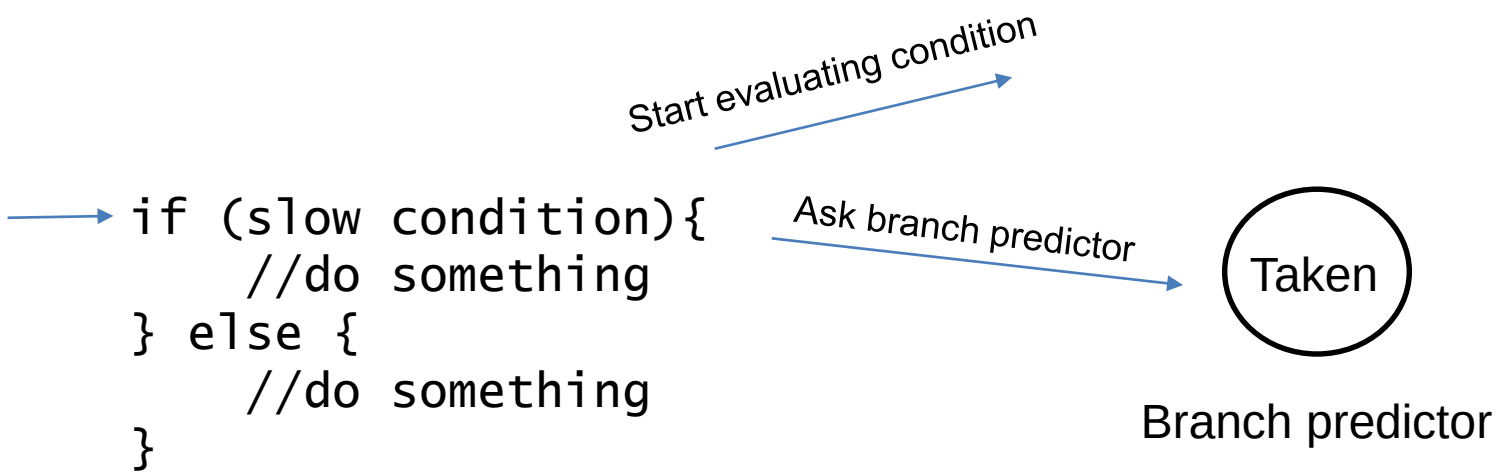
- Branch predictor

```
if (slow condition){
    //do something
} else {
    //do something
}
```



Background: Speculative Execution

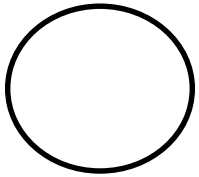
- Branch predictor



Background: Speculative Execution

- Branch predictor

```
if (slow condition){
    //do something
} else {
    //do something
}
```



Branch predictor

Background: Speculative Execution

Background: Speculative Execution

- Up to 200 instruction ahead

Background: Speculative Execution

- Up to 200 instruction ahead
- Revert the result of incorrect execution

Background: Speculative Execution

- Up to 200 instruction ahead
- Revert the result of incorrect execution
  - => No correctness issues?

Background: Speculative Execution

- Up to 200 instruction ahead
- Revert the result of incorrect execution
  - => No correctness issues?
- But speculative execution has measurable side effects

Spectre Attacks

Spectre Attacks

Spectre Attacks

- Locate sequence of instructions which act as covert channel transmitter

Spectre Attacks

- Locate sequence of instructions which act as covert channel transmitter
- Setup phase: Mistrain CPU into speculatively executing these instructions

Spectre Attacks

Spectre Attacks

- Second phase: speculatively execute instruction that leak information

Spectre Attacks

- Second phase: speculatively execute instruction that leak information
  - Via syscall/socket/file

Spectre Attacks

- Second phase: speculatively execute instruction that leak information
  - Via syscall/socket/file
  - Misexecute own code (e.g. sandbox, interpreter, JIT)

Spectre Attacks

Spectre Attacks

- Final phase: Recover data by retrieving over covert channel

Spectre Attacks

- Final phase: Recover data by retrieving over covert channel
  - Cache

Spectre Attacks

- Final phase: Recover data by retrieving over covert channel
  - Cache
  - Execution time

Spectre Attacks

- Final phase: Recover data by retrieving over covert channel
  - Cache
  - Execution time
  - ALU contention

Spectre Attacks

- Final phase: Recover data by retrieving over covert channel
  - Cache
  - Execution time
  - ALU contention
  - Memory contention

Spectre Attacks

- Final phase: Recover data by retrieving over covert channel
  - Cache
  - Execution time
  - ALU contention
  - Memory contention
  - Other microarchitectural state

Spectre Variants

Variant 1: Exploiting Conditional Branch Misprediction

Address	Value
000	4
001	13
010	9
011	2
100	12
101	75
110	24
111	87

k

Variant 1: Exploiting Conditional Branch Misprediction

- We want to find out what a certain byte in the virtual address of the victim is

Address	Value
000	4
001	13
010	9
011	2 ← k
100	12
101	75
110	24
111	87

Variant 1: Exploiting Conditional Branch Misprediction

- We want to find out what a certain byte in the virtual address of the victim is
- Let's call this secret byte k

Address	Value
000	4
001	13
010	9
011	2 ← k
100	12
101	75
110	24
111	87

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

- Locate a conditional which matches this pattern in the software you want to attack

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

- Locate a conditional which matches this pattern in the software you want to attack
- Setup phase:

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

- Locate a conditional which matches this pattern in the software you want to attack
- Setup phase:
  - call many times with some x < *array1_size* to mistrain branch predictor

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

- Locate a conditional which matches this pattern in the software you want to attack
- Setup phase:
  - call many times with some x < *array1_size* to mistrain branch predictor
  - Evict *array1_size* and *array2*, but leave secret byte k in cache

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

- Second phase: Choose x out-of-bounds such that *array1[x]* resolves to secret byte

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

Attacker

	Tag	Value	Tag	Value
Set 0				
Set 1	01	2		

Victim

	Address	Value
array1	000	4
	001	13
	010	9
	011	2 ← k
	100	12
array2	101	75
	110	24
	111	87

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

Attacker

Victim

	Tag	Value	Tag	Value
Set 0				
Set 1	01	2		

	Address	Value
array1	000	4
	001	13
	010	9
	011	2
	100	12
array2	101	75
	110	24
	111	87

k

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

Attacker

x = 4

Victim

	Tag	Value	Tag	Value
Set 0				
Set 1	01	2		

	Address	Value
array1	000	4
	001	13
	010	9
	011	2 ← k
	100	12
array2	101	75
	110	24
	111	87

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

Attacker

x = 4

Victim

	Tag	Value	Tag	Value
Set 0				
Set 1	01	2		

	Address	Value
array1	000	4
	001	13
	010	9
	011	2
	100	12
array2	101	75
	110	24
	111	87

k

Variant 1: Exploiting Conditional Branch Misprediction

Cache miss

```
if (x < array1_size)
    y = array2[array1[x]];
```

Attacker

x = 4

Victim

	Address	Value
array1	000	4
	001	13
	010	9
	011	2
	100	12
array2	101	75
	110	24
	111	87

k

	Tag	Value	Tag	Value
Set 0				
Set 1	01	2		

Variant 1: Exploiting Conditional Branch Misprediction

Cache miss

```
if (x < array1_size)
    y = array2[array1[x]];
```

Attacker

x = 4

Victim

	Tag	Value	Tag	Value
Set 0				
Set 1	01	2		

Address	Value
000	4
001	13
010	9
011	2
100	12
101	75
110	24
111	87

array1

array2

k

Variant 1: Exploiting Conditional Branch Misprediction

Cache miss

```
if (x < array1_size)
    y = array2[array1[x]];
```

Attacker

x = 4

Victim

	Tag	Value	Tag	Value
Set 0				
Set 1	01	2		

Address	Value
000	4
001	13
010	9
011	2
100	12
101	75
110	24
111	87

array1

array2

k

Variant 1: Exploiting Conditional Branch Misprediction

Cache miss

```
if (x < array1_size)
    y = array2[array1[x]];
```

Attacker

x = 4

Victim

	Tag	Value	Tag	Value
Set 0				
Set 1	01	2		

Address	Value
000	4
001	13
010	9
011	2
100	12
101	75
110	24
111	87

array1

array2

k

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```


[1] OSVIK, D. A., SHAMIR, A.,ANDTROMER, E. Cache attacks and countermeasures: The case of AES

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

- Final phase:

[1] OSVIK, D. A., SHAMIR, A.,ANDTROMER, E. Cache attacks and countermeasures: The case of AES

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

- Final phase:
  - If *array2* is readable by attacker, load *array2[n]* for all n, will be fast for n==k

[1] OSVIK, D. A., SHAMIR, A.,ANDTROMER, E. Cache attacks and countermeasures: The case of AES

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

- Final phase:
  - If *array2* is readable by attacker, load *array2[n]* for all n, will be fast for n==k
  - Otherwise detect eviction

[1] OSVIK, D. A., SHAMIR, A.,ANDTROMER, E. Cache attacks and countermeasures: The case of AES

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

- Final phase:
  - If *array2* is readable by attacker, load *array2[n]* for all n, will be fast for n==k
  - Otherwise detect eviction
    - Prime & Probe [1]

[1] OSVIK, D. A., SHAMIR, A.,ANDTROMER, E. Cache attacks and countermeasures: The case of AES

Variant 1: Exploiting Conditional Branch Misprediction

```
if (x < array1_size)
    y = array2[array1[x]];
```

- Final phase:
  - If *array2* is readable by attacker, load *array2[n]* for all n, will be fast for n==k
  - Otherwise detect eviction
    - Prime & Probe [1]
  - Call method again with in bounds value of x, if array1[x'] == k, will be fast

[1] OSVIK, D. A., SHAMIR, A.,ANDTROMER, E. Cache attacks and countermeasures: The case of AES

Variant 1: Implementations

Variant 1: Implementations

- In C:

Variant 1: Implementations

- In C:
  - Wrote a victim function which has the pattern

Variant 1: Implementations

- In C:
  - Wrote a victim function which has the pattern
  - Attacker function

Variant 1: Implementations

- In C:
  - Wrote a victim function which has the pattern
  - Attacker function
  - Were able to read out address space at 10 kB/second

Variant 1: Implementations

- In C:
  - Wrote a victim function which has the pattern
  - Attacker function
  - Were able to read out address space at 10 kB/second
- In JS:

Variant 1: Implementations

- In C:
  - Wrote a victim function which has the pattern
  - Attacker function
  - Were able to read out address space at 10 kB/second
- In JS:
  - JS gets JITed and bounds checks inserted

Variant 1: Implementations

- In C:
  - Wrote a victim function which has the pattern
  - Attacker function
  - Were able to read out address space at 10 kB/second
- In JS:
  - JS gets JITed and bounds checks inserted
  - Works even though no high res. timer available

Variant 1: Implementations

- In C:
  - Wrote a victim function which has the pattern
  - Attacker function
  - Were able to read out address space at 10 kB/second
- In JS:
  - JS gets JITed and bounds checks inserted
  - Works even though no high res. timer available
  - Able to read out browser's address space

Variant 2: Poisoning Indirect Branches

Variant 2: Poisoning Indirect Branches

- Locate gadgets whose execution will leak the chosen memory in the process you want to attack (either source code or in shared library)

Variant 2: Poisoning Indirect Branches

- Locate gadgets whose execution will leak the chosen memory in the process you want to attack (either source code or in shared library)
- Example gadget:

Variant 2: Poisoning Indirect Branches

- Locate gadgets whose execution will leak the chosen memory in the process you want to attack (either source code or in shared library)
- Example gadget:
  - add R2, [R1]

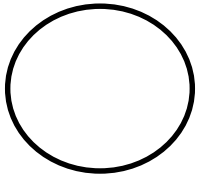
Variant 2: Poisoning Indirect Branches

- Locate gadgets whose execution will leak the chosen memory in the process you want to attack (either source code or in shared library)
- Example gadget:
  - add R2, [R1]
  - mov R3, [R2]

Background: Speculative Execution

- Branch Target Buffer

```
jmp //slow computation
```

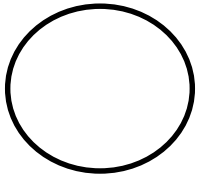


Branch Target Buffer

Background: Speculative Execution

- Branch Target Buffer

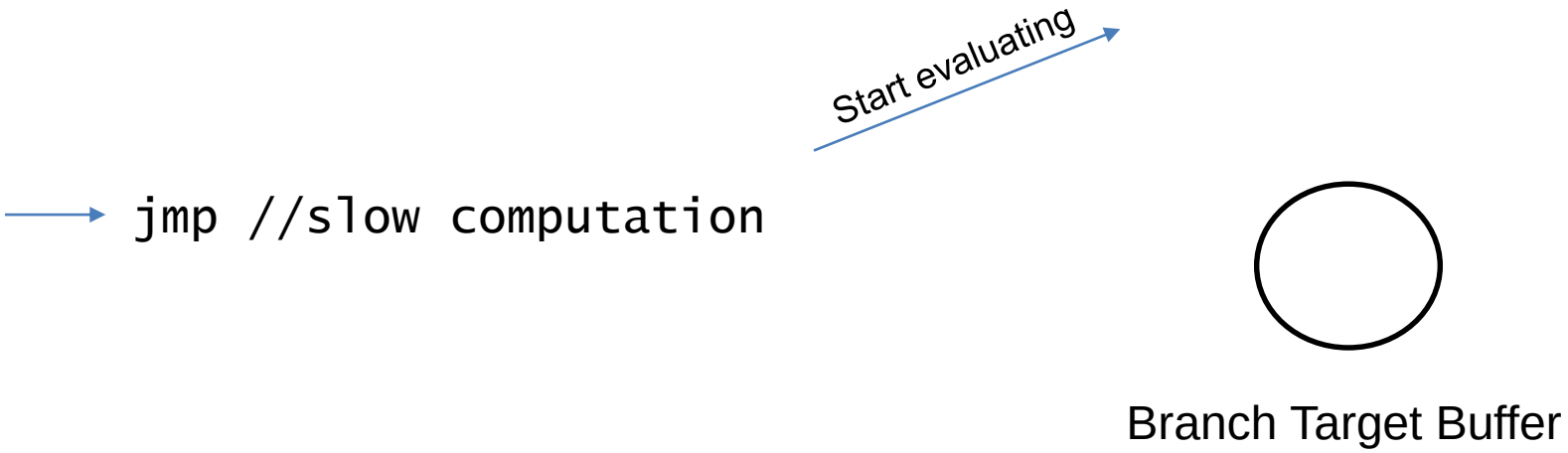
```
jmp //slow computation
```



Branch Target Buffer

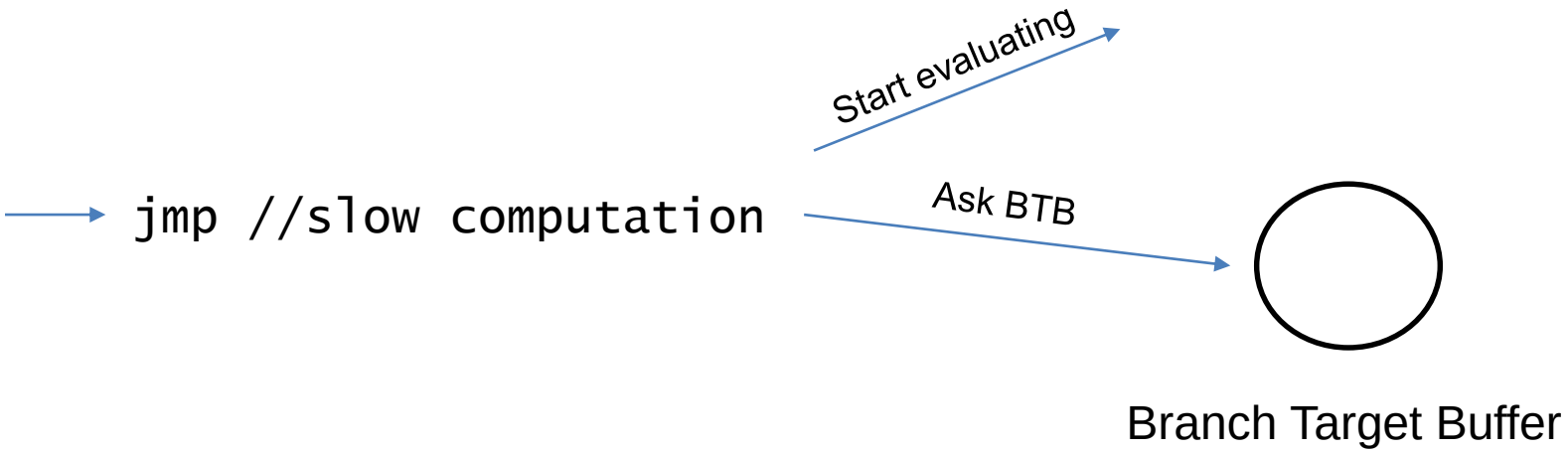
Background: Speculative Execution

- Branch Target Buffer



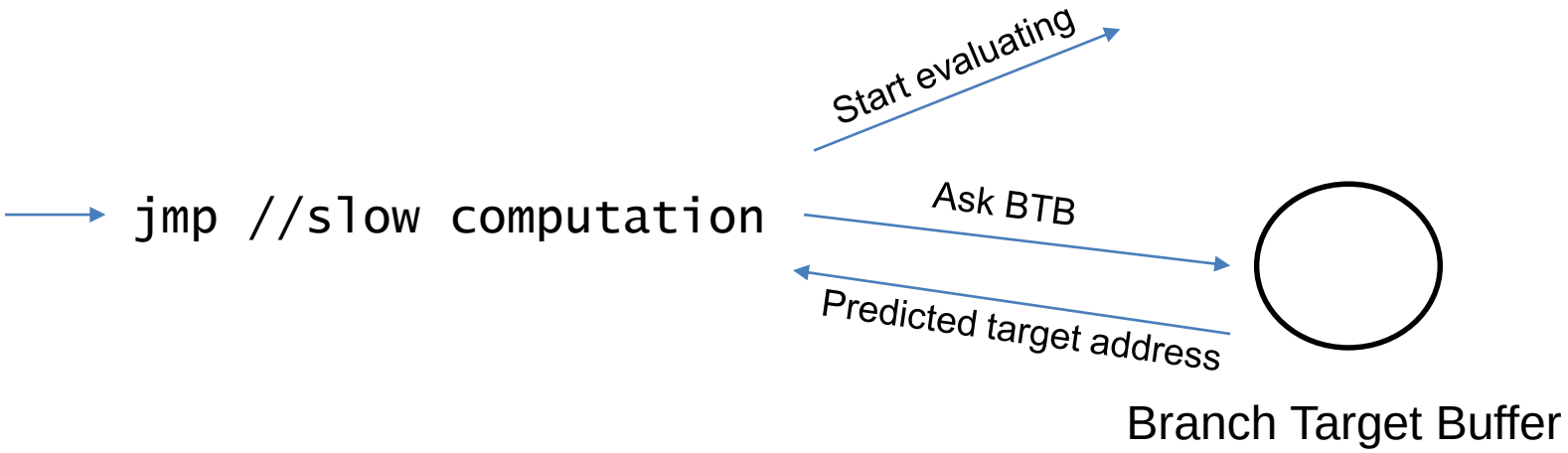
Background: Speculative Execution

- Branch Target Buffer



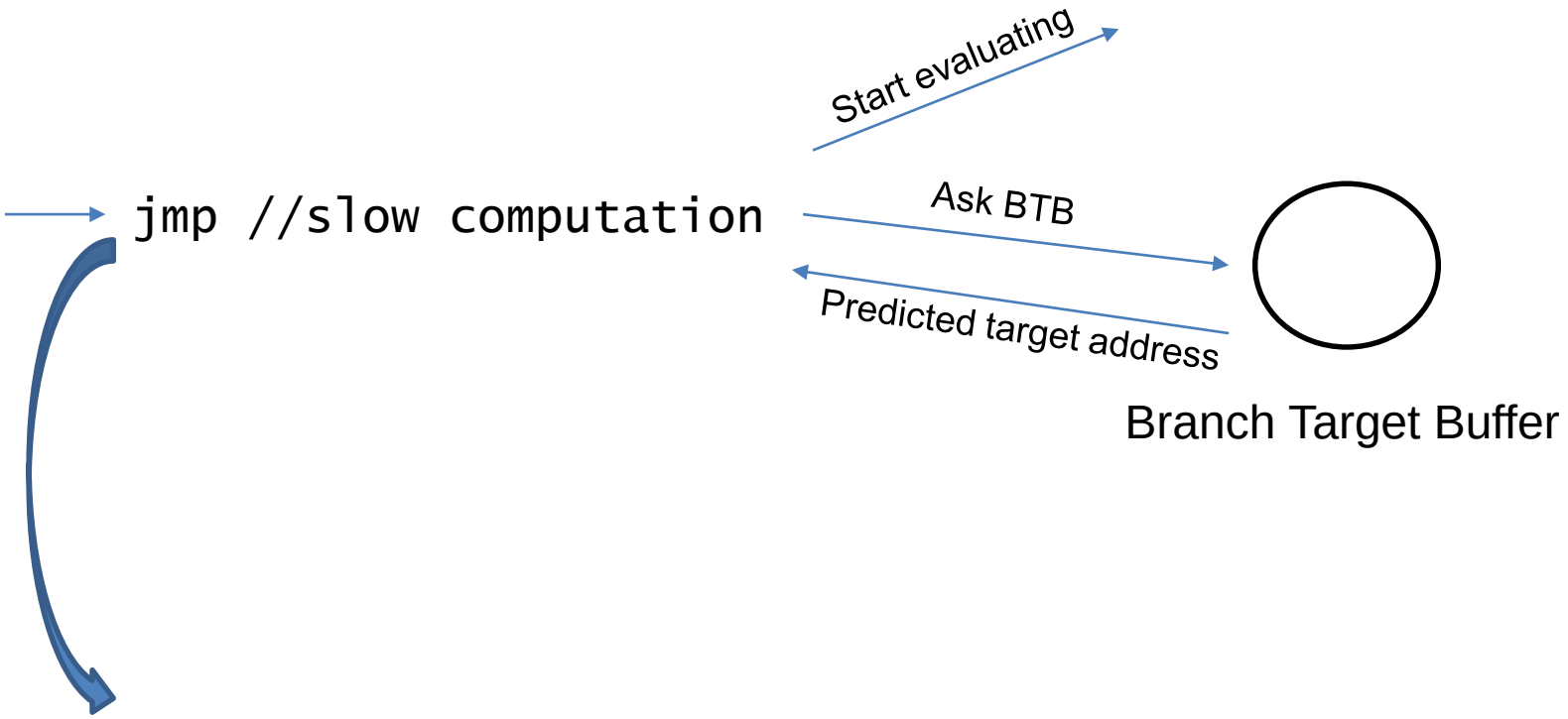
Background: Speculative Execution

- Branch Target Buffer



Background: Speculative Execution

- Branch Target Buffer



Variant 2: Poisoning Indirect Branches

Variant 2: Poisoning Indirect Branches

- Setup phase:

Variant 2: Poisoning Indirect Branches

- Setup phase:
  - Train branch target buffer in attacker thread to jump to the chosen gadget's virtual address

Variant 2: Poisoning Indirect Branches

- Setup phase:
  - Train branch target buffer in attacker thread to jump to the chosen gadget's virtual address
    - This works because the BTB is unaware/doesn't care about process ids

Variant 2: Poisoning Indirect Branches

- Setup phase:
  - Train branch target buffer in attacker thread to jump to the chosen gadget's virtual address
    - This works because the BTB is unaware/doesn't care about process ids
- Example of indirect branch:

Variant 2: Poisoning Indirect Branches

- Setup phase:
  - Train branch target buffer in attacker thread to jump to the chosen gadget's virtual address
    - This works because the BTB is unaware/doesn't care about process ids
- Example of indirect branch:
  - jmp eax

Variant 2: Poisoning Indirect Branches

Variant 2: Poisoning Indirect Branches

- Second phase:

Variant 2: Poisoning Indirect Branches

- Second phase:
  - Victim speculatively jumps to gadget, which then leaks information

Variant 2: Poisoning Indirect Branches

- Second phase:
  - Victim speculatively jumps to gadget, which then leaks information
- Final phase: Recover over covert channel

Variant 2: Implementation in Windows

Variant 2: Implementation in Windows

- Creates random key, calls sleep, reads from file, calls crypto

Variant 2: Implementation in Windows

- Creates random key, calls sleep, reads from file, calls crypto
- When compiled with optimization,Sleep() gets made with file data in registers ebx and edi

Variant 2: Implementation in Windows

- Creates random key, calls sleep, reads from file, calls crypto
- When compiled with optimization,Sleep() gets made with file data in registers ebx and edi
- Found in ntdll.dll

Variant 2: Implementation in Windows

- Creates random key, calls sleep, reads from file, calls crypto
- When compiled with optimization,Sleep() gets made with file data in registers ebx and edi
- Found in ntdll.dll
  - adc edi,dword ptr [ebx+edx+13BE13BDh]

Variant 2: Implementation in Windows

- Creates random key, calls sleep, reads from file, calls crypto
- When compiled with optimization,Sleep() gets made with file data in registers ebx and edi
- Found in ntdll.dll
  - adc edi,dword ptr [ebx+edx+13BE13BDh]
  - adc dl,byte ptr [edi]

Variant 2: Implementation in Windows

Variant 2: Implementation in Windows

- Branch to mistrain found in Sleep()

Variant 2: Implementation in Windows

- Branch to mistrain found in Sleep()
  - “jmp dword ptr ds:[76AE0078h]”

Variant 2: Implementation in Windows

- Branch to mistrain found in Sleep()
  - “jmp dword ptr ds:[76AE0078h]”
- Speed: 41 B/s

Other Methods of achieving speculative execution

[1] Spectre Returns! Speculation Attacks using the Return Stack Buffer
By Esmaiel Mohammadian Koruyeh, Khaled Khasawneh,
Chengyu Song and Nael Abu-Ghazaleh

Other Methods of achieving speculative execution

- Mistraining return instructions [1]

[1] Spectre Returns! Speculation Attacks using the Return Stack Buffer
By Esmaiel Mohammadian Koruyeh, Khaled Khasawneh,
Chengyu Song and Nael Abu-Ghazaleh

Other Methods of achieving speculative execution

- Mistraining return instructions [1]
- Return from interrupts

[1] Spectre Returns! Speculation Attacks using the Return Stack Buffer
By Esmaiel Mohammadian Koruyeh, Khaled Khasawneh,
Chengyu Song and Nael Abu-Ghazaleh

Method of leaking information

Method of leaking information

- Evict+Time

Method of leaking information

- Evict+Time

```
if (false but mispredicts as true)
   read array1[R1]
read [R2]
```

Method of leaking information

- Evict+Time

```
if (false but mispredicts as true)
   read array1[R1]
read [R2]
```

- Instruction Timing

Method of leaking information

- Evict+Time

```
if (false but mispredicts as true)
   read array1[R1]
read [R2]
```

- Instruction Timing

```
if (false but mispredicts as true)
   multiply R1, R2
multiply R3, R4
```

Method of leaking information

- Evict+Time

```
if (false but mispredicts as true)
   read array1[R1]
read [R2]
```

- Instruction Timing

```
if (false but mispredicts as true)
    multiply R1, R2
 multiply R3, R4
```

- Contention on the Register File

Mitigation Options

[1] https://support.google.com/faqs/answer/7625886

[2] https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/compiler-support-for-mitigations

Mitigation Options

- Turn off speculation

[1] https://support.google.com/faqs/answer/7625886

[2] https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/compiler-support-for-mitigations

Mitigation Options

- Turn off speculation
  - =>very large performance impact

[1] https://support.google.com/faqs/answer/7625886

[2] https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/compiler-support-for-mitigations

Mitigation Options

- Turn off speculation
  - =>very large performance impact
  - Itanium, Mill architectures not vulnerable

[1] https://support.google.com/faqs/answer/7625886

[2] https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/compiler-support-for-mitigations

Mitigation Options

- Turn off speculation
  - =>very large performance impact
  - Itanium, Mill architectures not vulnerable
- Retpoline [1]

[1] https://support.google.com/faqs/answer/7625886

[2] https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/compiler-support-for-mitigations

Mitigation Options

- Turn off speculation
  - =>very large performance impact
  - Itanium, Mill architectures not vulnerable
- Retpoline [1]
  - swaps indirect branches for returns, to avoid using predictions which come from the BTB

[1] https://support.google.com/faqs/answer/7625886

[2] https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/compiler-support-for-mitigations

Mitigation Options

- Turn off speculation
  - =>very large performance impact
  - Itanium, Mill architectures not vulnerable
- Retpoline [1]
  - swaps indirect branches for returns, to avoid using predictions which come from the BTB
- Masking, etc.

[1] https://support.google.com/faqs/answer/7625886

[2] https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/compiler-support-for-mitigations

Mitigation Options

- Turn off speculation
  - =>very large performance impact
  - Itanium, Mill architectures not vulnerable
- Retpoline [1]
  - swaps indirect branches for returns, to avoid using predictions which come from the BTB
- Masking, etc.
  - Addressing Spectre Variant 1 (CVE-2017-5753) in Software [2]

[1] https://support.google.com/faqs/answer/7625886

[2] https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/compiler-support-for-mitigations

Mitigation Options

[1] oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis
Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, Abhik Roychoudhury

Mitigation Options

- Halt speculative execution on potentially sensitive execution paths

[1] oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis
Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, Abhik Roychoudhury

Mitigation Options

- Halt speculative execution on potentially sensitive execution paths
  - Not enough only on security-critical code because non-security-critical code in same process.

[1] oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis
Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, Abhik Roychoudhury

Mitigation Options

- Halt speculative execution on potentially sensitive execution paths
  - Not enough only on security-critical code because non-security-critical code in same process.
  - Compiler can’t find automatically [1]

[1] oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis
Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, Abhik Roychoudhury

Mitigation Options

- Halt speculative execution on potentially sensitive execution paths
  - Not enough only on security-critical code because non-security-critical code in same process.
  - Compiler can’t find automatically [1]
  - Need to recompile (what about legacy software?)

[1] oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis
Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, Abhik Roychoudhury

Mitigation Options

- Halt speculative execution on potentially sensitive execution paths
  - Not enough only on security-critical code because non-security-critical code in same process.
  - Compiler can’t find automatically [1]
  - Need to recompile (what about legacy software?)
- Flush branch prediction state on context switch

[1] oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis
Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, Abhik Roychoudhury

Mitigation Options

Mitigation Options

- Countermeasures limited to cache likely insufficient

Mitigation Options

- Countermeasures limited to cache likely insufficient
  - => different microarchitectural state can be used to leak information

Novelty

Novelty

Novelty

- First to exploit speculative execution

Novelty

- First to exploit speculative execution
- Developers now need to know about microarchitecture to code non-vulnerable software!

Strengths

Strengths of the Paper

Strengths of the Paper

- Hits at fundamentals of modern CPU

Strengths of the Paper

- Hits at fundamentals of modern CPU
- No good mitigations

Strengths of the Paper

- Hits at fundamentals of modern CPU
- No good mitigations
- Non-buggy software affected

Strengths of the Paper

- Hits at fundamentals of modern CPU
- No good mitigations
- Non-buggy software affected
- Can be used by JS

Strengths of the Paper

- Hits at fundamentals of modern CPU
- No good mitigations
- Non-buggy software affected
- Can be used by JS
  - And even remotely:

Strengths of the Paper

- Hits at fundamentals of modern CPU
- No good mitigations
- Non-buggy software affected
- Can be used by JS
  - And even remotely:
    - NetSpectre: Read Arbitrary Memory over Network by Michael Schwarz, Martin Schwarzl, Moritz Lipp, Daniel Gruss

Strengths of the Paper

- Hits at fundamentals of modern CPU
- No good mitigations
- Non-buggy software affected
- Can be used by JS
  - And even remotely:
    - NetSpectre: Read Arbitrary Memory over Network by Michael Schwarz, Martin Schwarzl, Moritz Lipp, Daniel Gruss
- Very well written paper

Strengths of the Paper

- Hits at fundamentals of modern CPU
- No good mitigations
- Non-buggy software affected
- Can be used by JS
  - And even remotely:
    - NetSpectre: Read Arbitrary Memory over Network by Michael Schwarz, Martin Schwarzl, Moritz Lipp, Daniel Gruss
- Very well written paper
- Gives a refresher on virtual memory, caches, CPU architecture

Weaknesses

Weaknesses/Limitations of the Paper

Weaknesses/Limitations of the Paper

- Have to target specific application and find gadgets in those

Weaknesses/Limitations of the Paper

- Have to target specific application and find gadgets in those
  - => But one can search in shared libraries

Weaknesses/Limitations of the Paper

- Have to target specific application and find gadgets in those
  - => But one can search in shared libraries
- Doesn’t tell us the speed of the Javascript implementation

Related papers

Related papers

- MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols by Caroline Trippel, Daniel Lustig, Margaret Martonosi

Related papers

- MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols by Caroline Trippel, Daniel Lustig, Margaret Martonosi
- Trusted Browsers for Uncertain Times by David Kohlbrenner and Hovav Shacham

Related papers

- MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols by Caroline Trippel, Daniel Lustig, Margaret Martonosi
- Trusted Browsers for Uncertain Times by David Kohlbrenner and Hovav Shacham
- Foreshadow: Extracting the Keys to the {Intel SGX} Kingdom with Transient Out-of-Order Execution by Van Bulck, Jo and Minkin, Marina and Weisse, Ofir and Genkin, Daniel and Kasikci, Baris and Piessens, Frank and Silberstein, Mark and Wenisch, Thomas F. and Yarom, Yuval and Strackx, Raoul

Related papers

- MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols by Caroline Trippel, Daniel Lustig, Margaret Martonosi
- Trusted Browsers for Uncertain Times by David Kohlbrenner and Hovav Shacham
- Foreshadow: Extracting the Keys to the {Intel SGX} Kingdom with Transient Out-of-Order Execution by Van Bulck, Jo and Minkin, Marina and Weisse, Ofir and Genkin, Daniel and Kasikci, Baris and Piessens, Frank and Silberstein, Mark and Wenisch, Thomas F. and Yarom, Yuval and Strackx, Raoul
- A Systematic Evaluation of Transient Execution Attacks and Defenses by Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, Daniel Gruss

Discussion Starters

Discussion Starters

- Is there a fundamental tradeoff between security and speed?

Discussion Starters

- Is there a fundamental tradeoff between security and speed?
- Can Spectre be fixed in hardware?