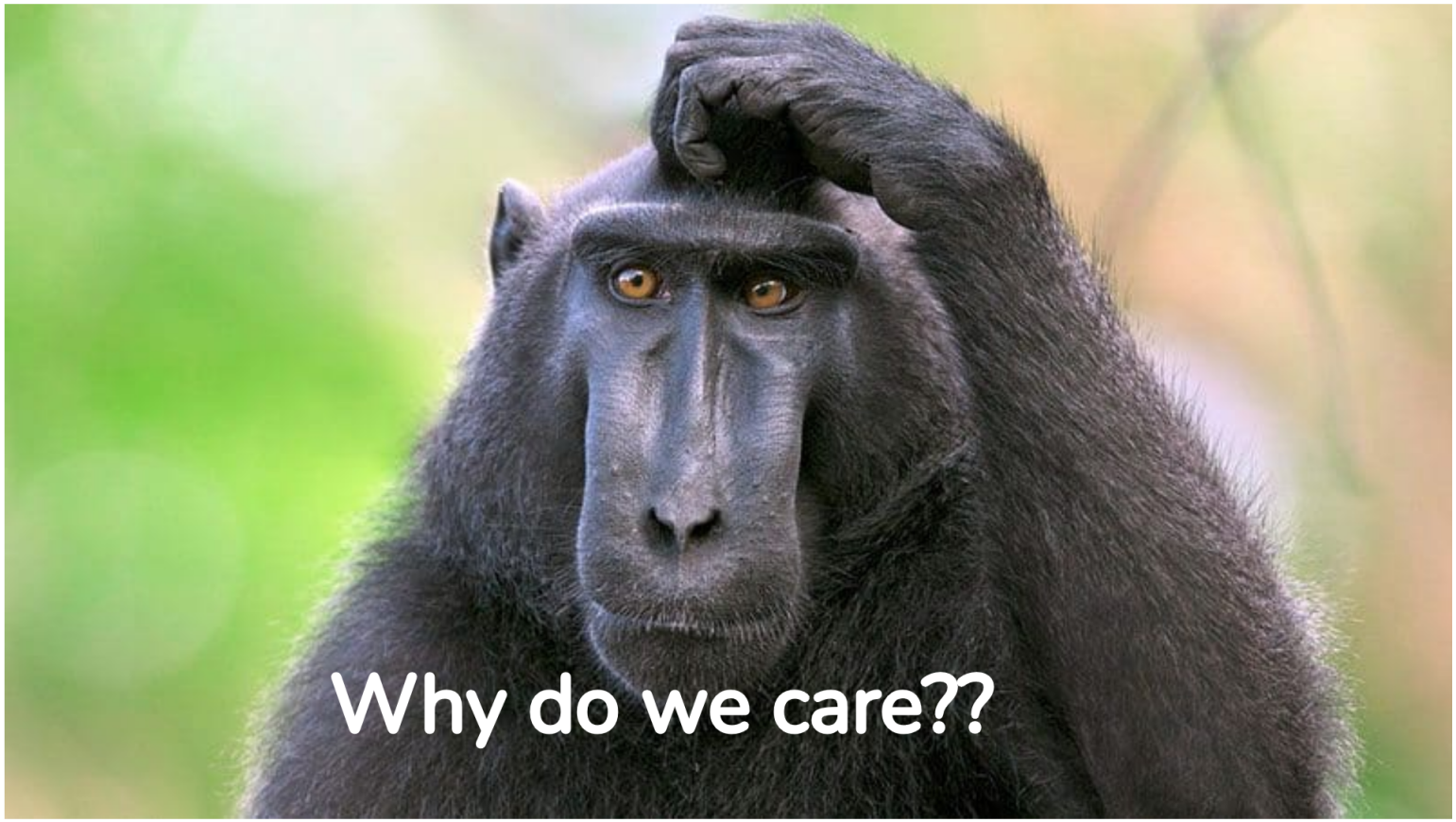
Reverse Engineering of Intel's Branch Prediction

Nick Mahling

Institute of IT-Security

17.11.2023

UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT



Branch Prediction

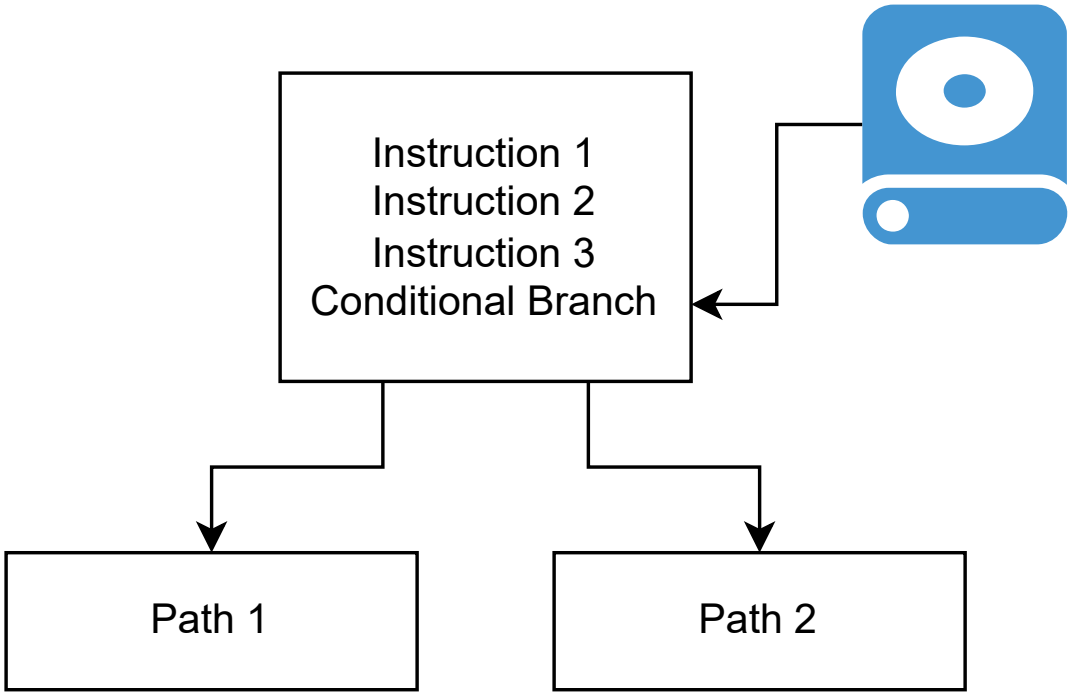
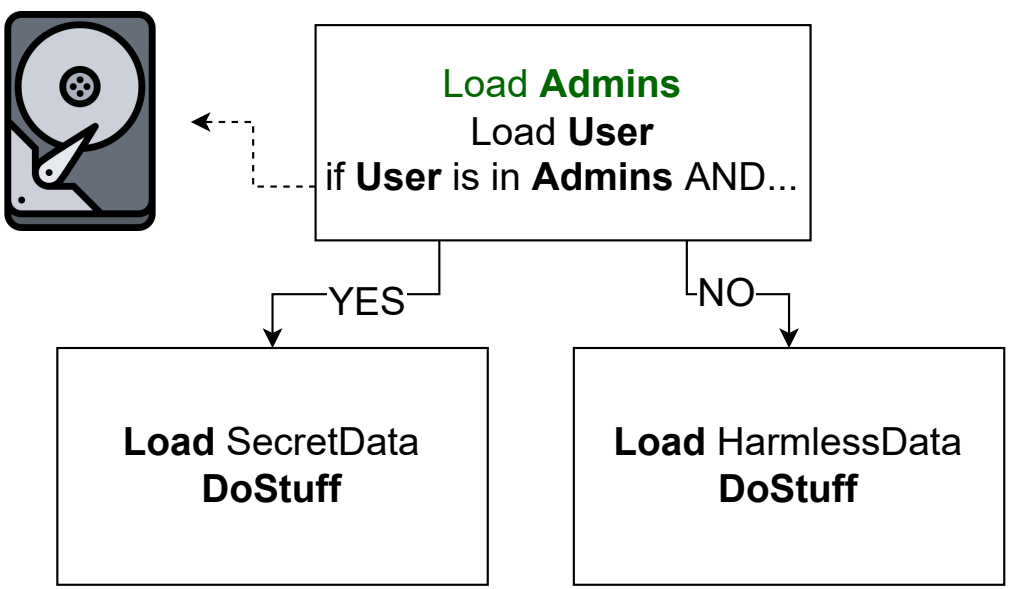
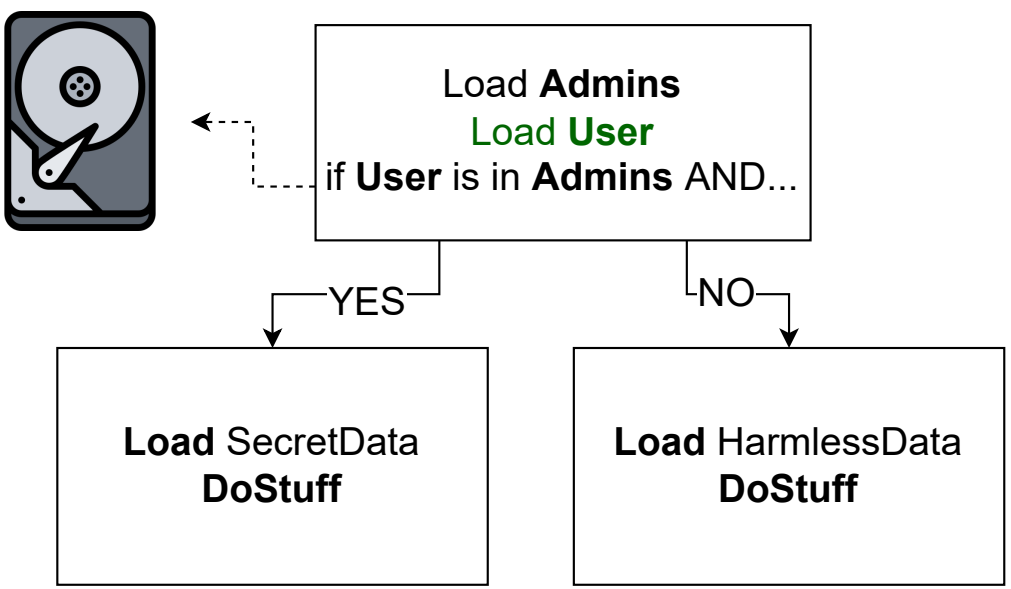


Figure: High level graph representation of a conditional branch.

Speculative Execution

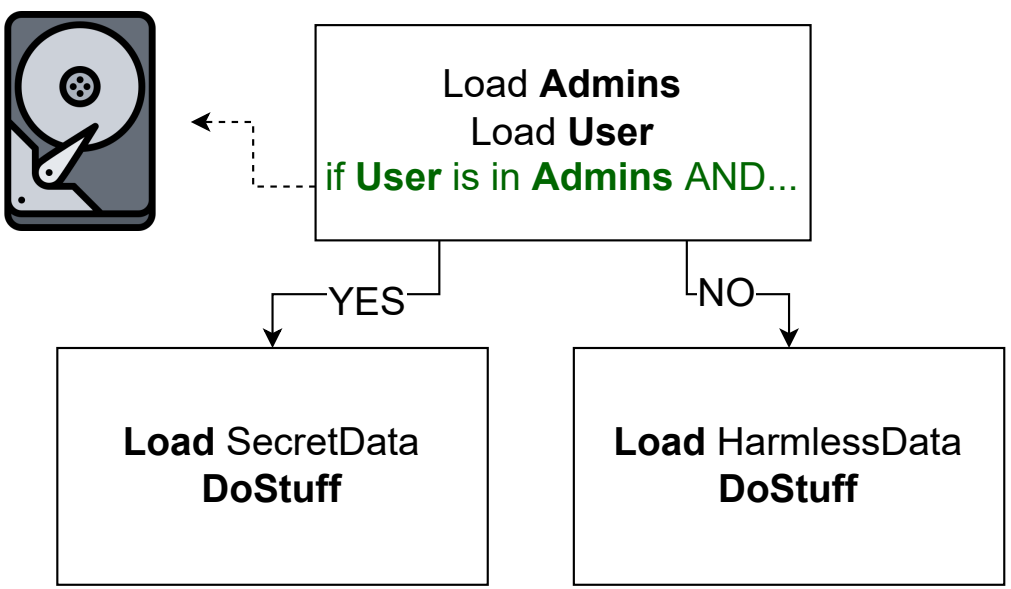


Speculative Execution



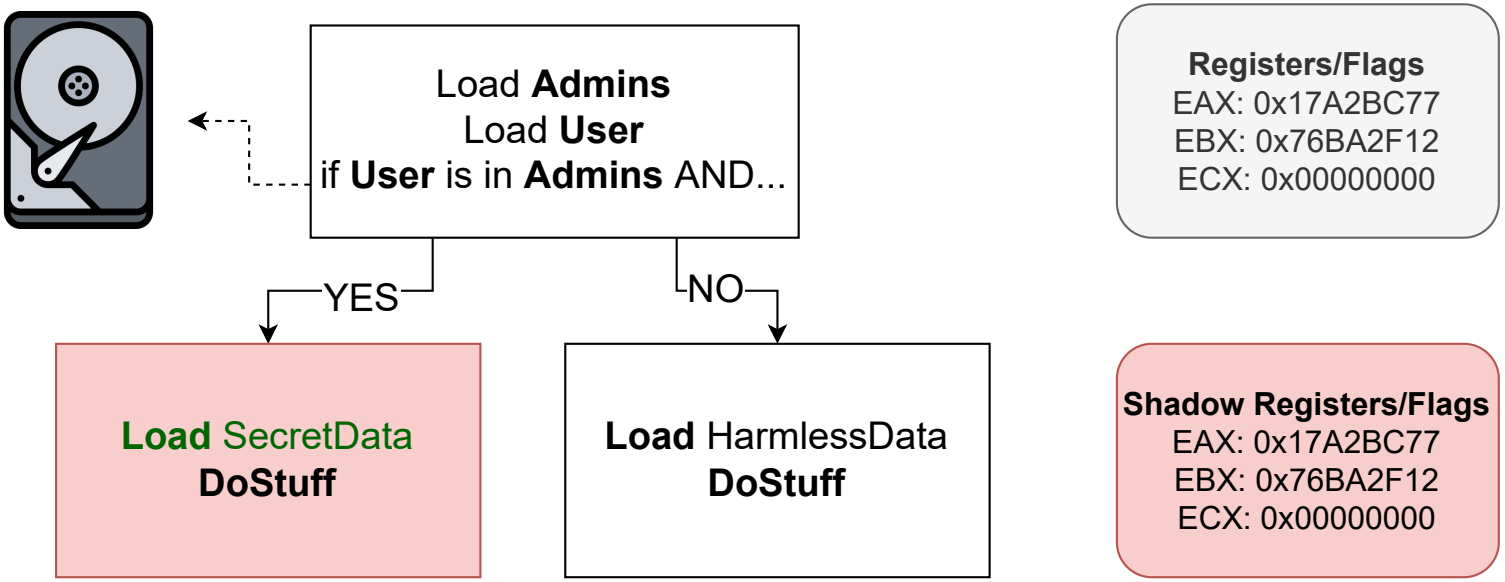
Registers/Flags
EAX: 0x17A2BC77
EBX: 0x00000000
ECX: 0x00000000

Speculative Execution

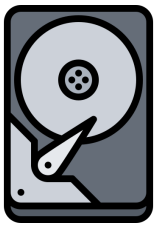


Registers/Flags
EAX: 0x17A2BC77
EBX: 0x76BA2F12
ECX: 0x00000000

Speculative Execution



Speculative Execution



Load **Admins**
Load **User**
if **User** is in **Admins** AND...

YES

Load SecretData
DoStuff

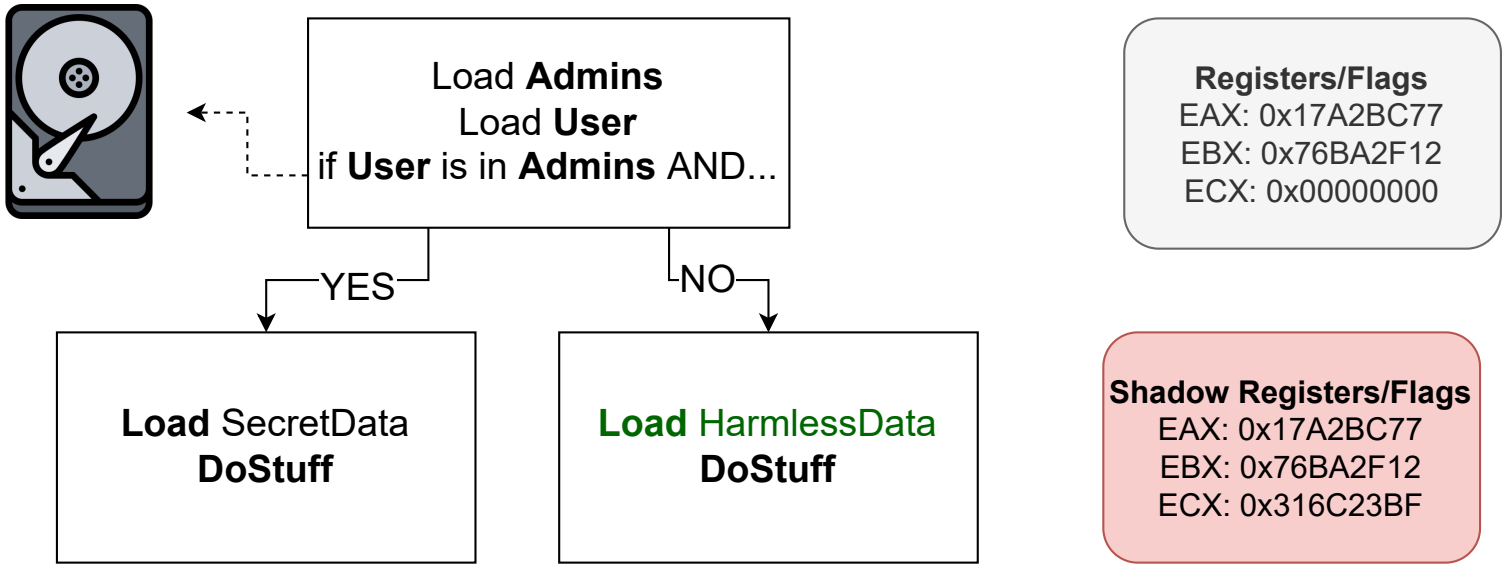
NO

Load HarmlessData
DoStuff

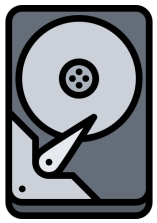
Registers/Flags
EAX: 0x17A2BC77
EBX: 0x76BA2F12
ECX: 0x00000000

Shadow Registers/Flags
EAX: 0x17A2BC77
EBX: 0x76BA2F12
ECX: 0x316C23BF

Speculative Execution



Speculative Execution



Load **Admins**
Load **User**
if **User** is in **Admins** AND...

YES

NO

Load SecretData
DoStuff

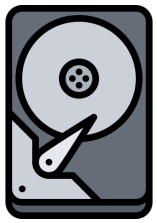
Load HarmlessData
DoStuff

Registers/Flags
EAX: 0x17A2BC77
EBX: 0x76BA2F12
ECX: 0x00000000

Shadow Registers/Flags
EAX: 0x17A2BC77
EBX: 0x76BA2F12
ECX: 0x00000000

Cache
Admins
User

Speculative Execution



Load **Admins**
Load **User**
if **User** is in **Admins** AND...

YES

NO

Load SecretData
DoStuff

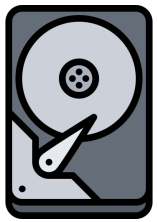
Load HarmlessData
DoStuff

Registers/Flags
EAX: 0x17A2BC77
EBX: 0x76BA2F12
ECX: 0x00000000

Shadow Registers/Flags
EAX: 0x17A2BC77
EBX: 0x76BA2F12
ECX: 0x316C23BF

Cache
Admins
User
SecretData

Speculative Execution



Load **Admins**
Load **User**
if **User** is in **Admins** AND...

YES

Load SecretData
DoStuff

NO

Load HarmlessData
DoStuff

Registers/Flags
EAX: 0x17A2BC77
EBX: 0x76BA2F12
ECX: 0x00000000

Shadow Registers/Flags
EAX: 0x17A2BC77
EBX: 0x76BA2F12
ECX: 0x316C23BF

Cache
Admins
User
SecretData

Attacker Scenario

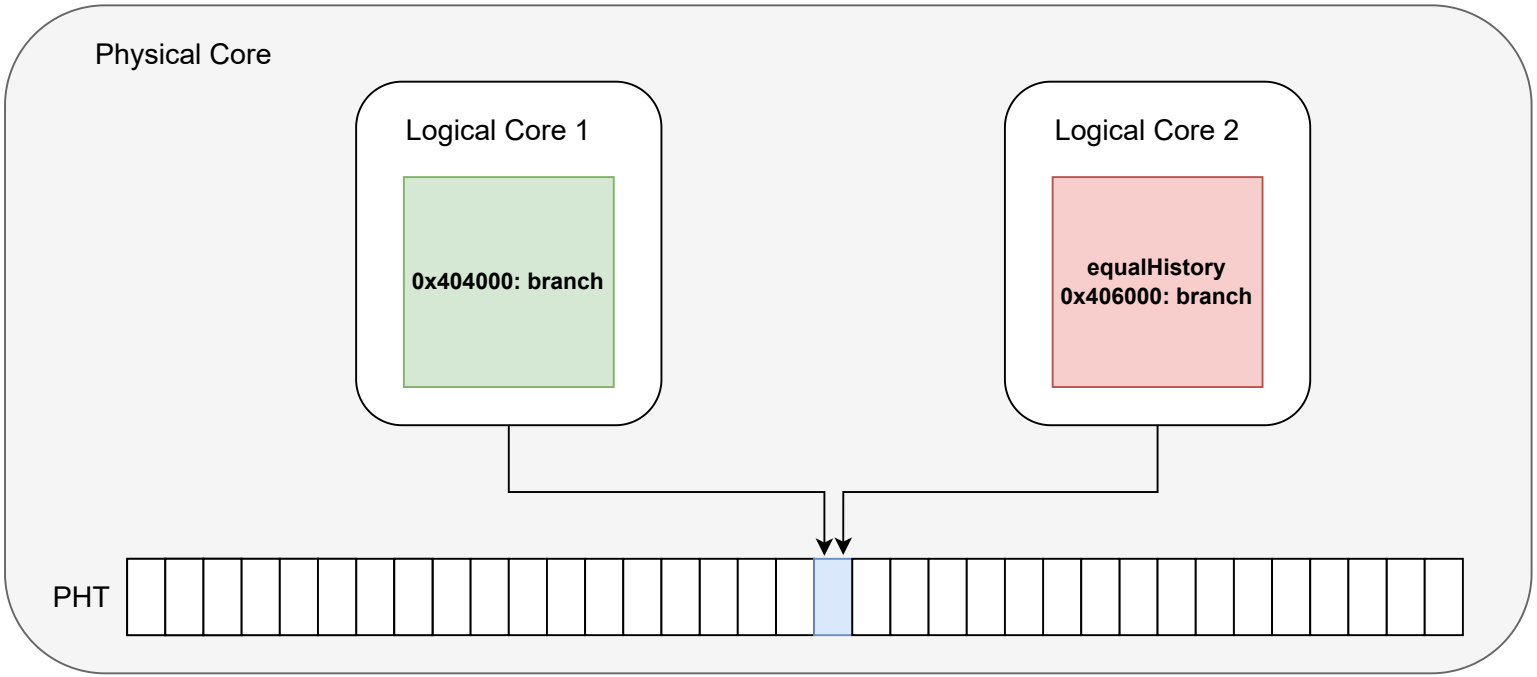


Figure: Attack scenario [CBS+19, KHF+19].

PHT Indexing

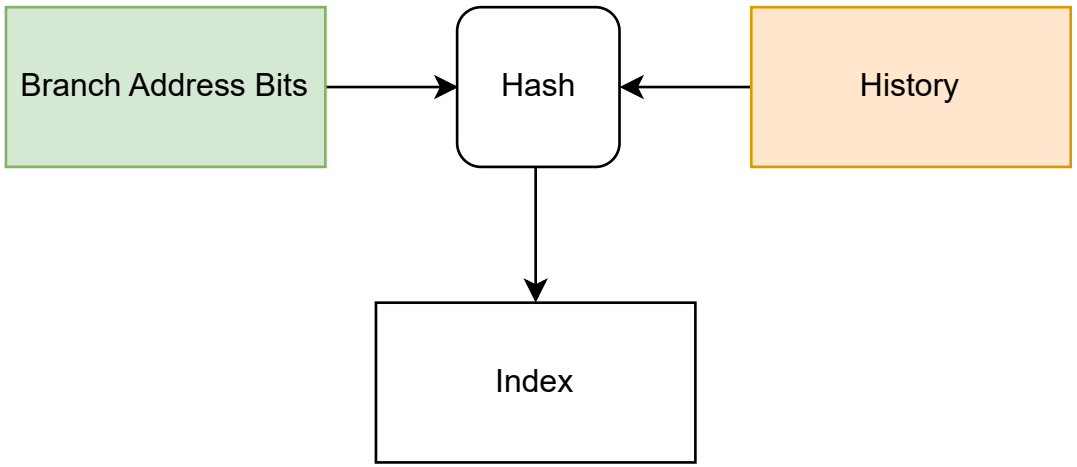


Figure: High level hash function for PHT indexing

Injector & Target Setup

Algorithm 1: *Target* program that gets mistrained by the *injector*

```
1 setCore(6)
2 while experimentOn do
3     normalizeHistory()
4     //Not taken
5     if True then
6         nop
```

Algorithm 2: *Injector* program

```
1 setCore(14)
2 while experimentOn do
3     normalizeHistory()
4     //Taken
5     if False then
6         nop
```

Result

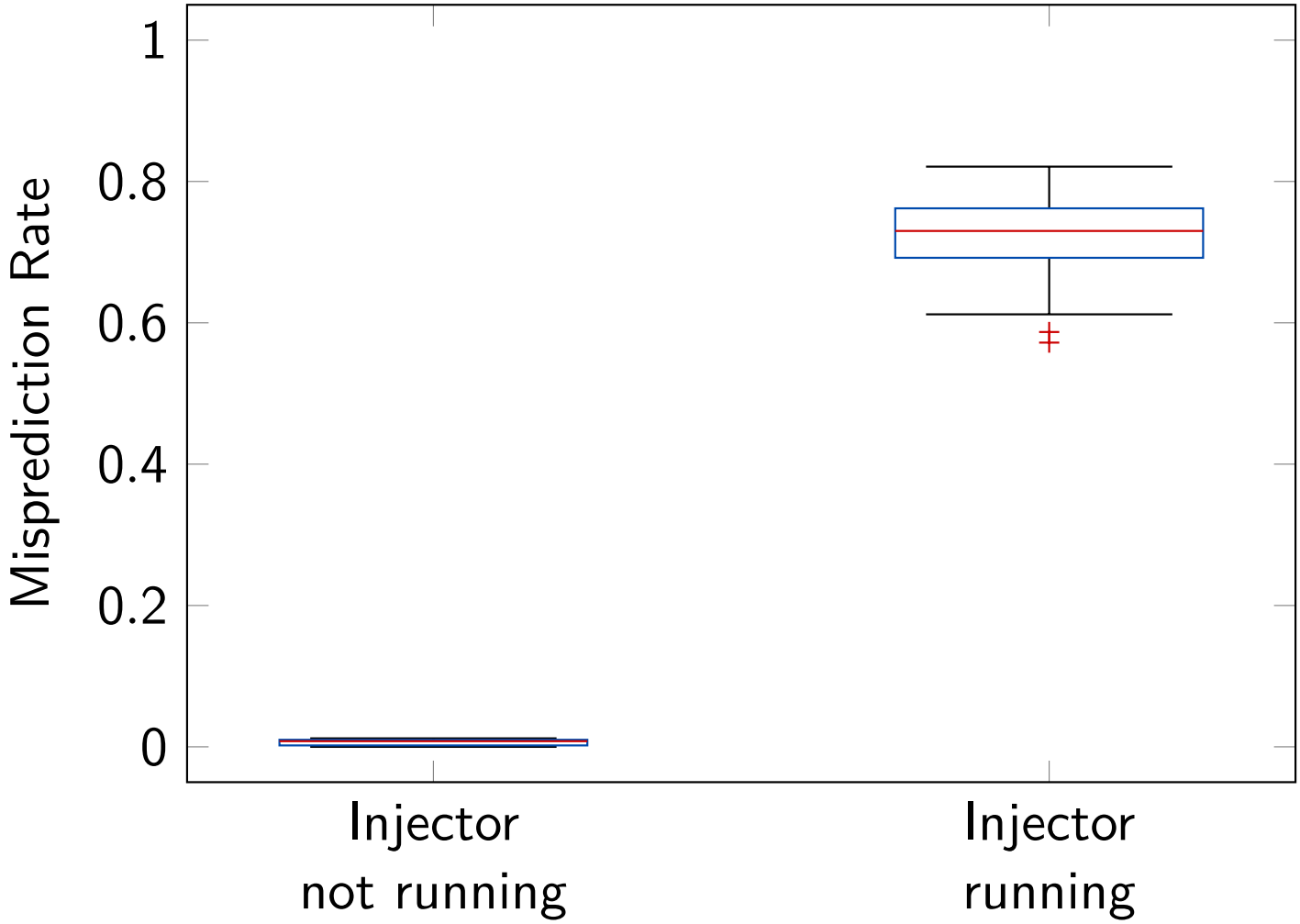


Figure: The mispredicts of the spy branch in the *target* go up when the *injector* runs. Thus, it successfully mistrains the *target's* branch.

What we know:

- Each PHT entry has a 2-bit/3-bit saturating counter
- Logical cores on the same physical core share branch prediction units
- A history impacts PHT indexing
- All branches affect the history except for not taken conditional branches
- The history stores the last 93 branches [YTN+23]

Reverse the Hash Function

- Normalize history
- Increase address of injector branch by one each iteration
- Collect all colliding branches

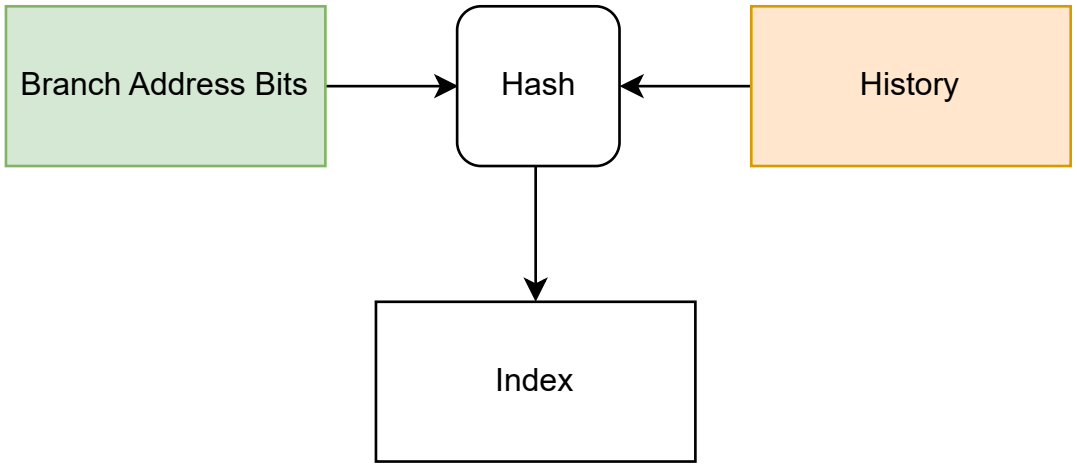
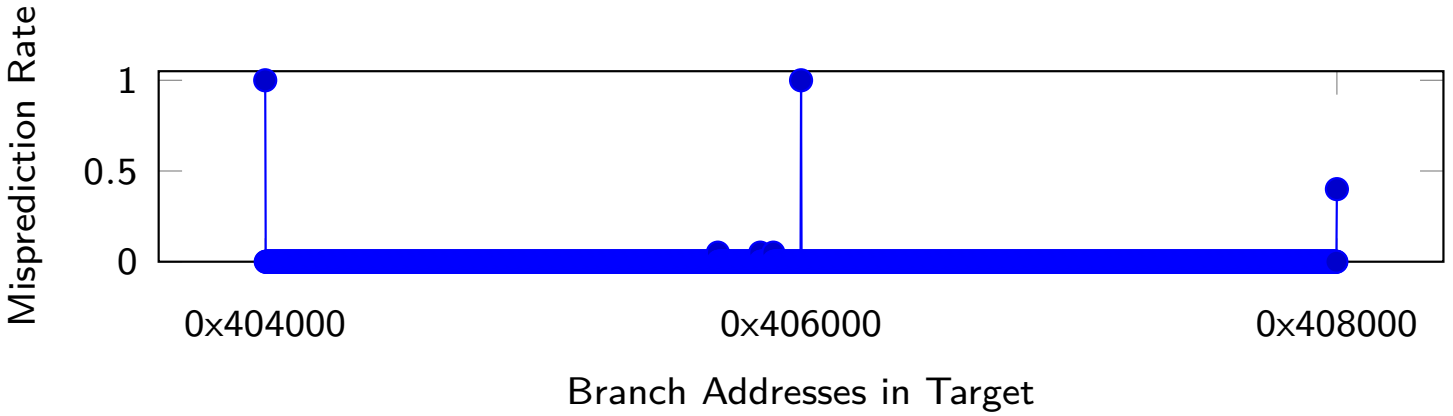
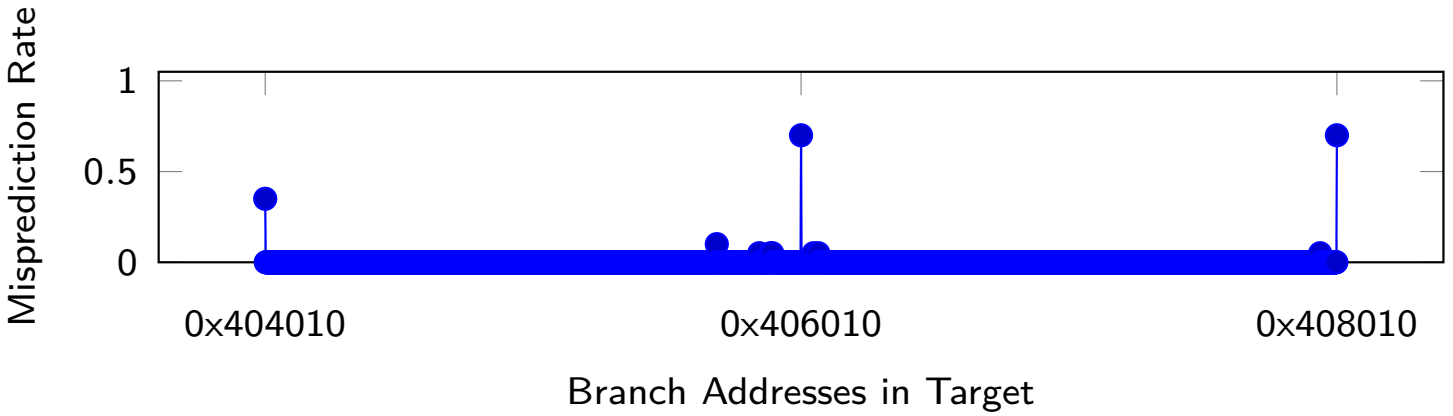
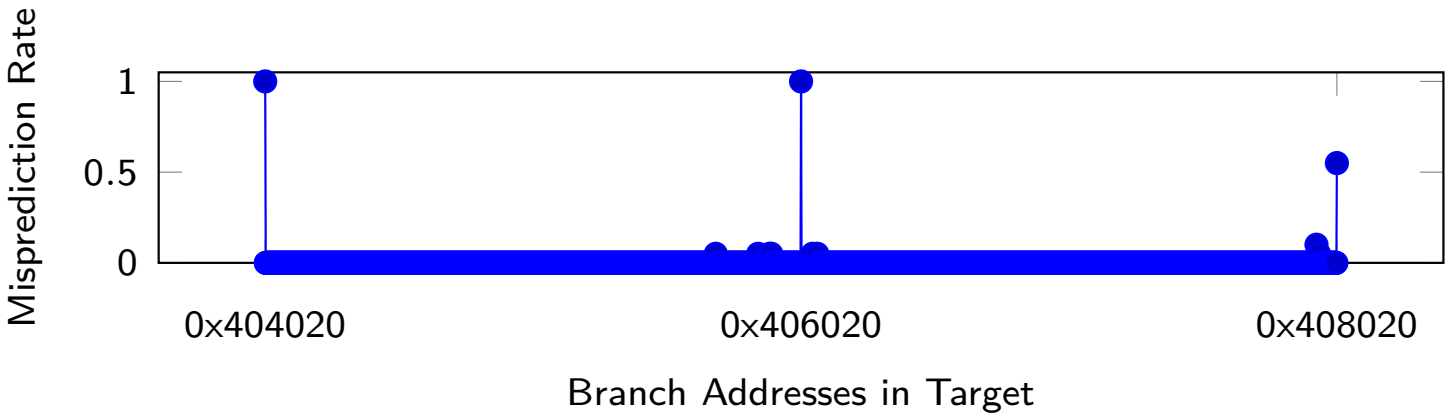


Figure: High level hash function for PHT indexing

Reverse the Hash Function







Spectre PoC (Proof of Concept)

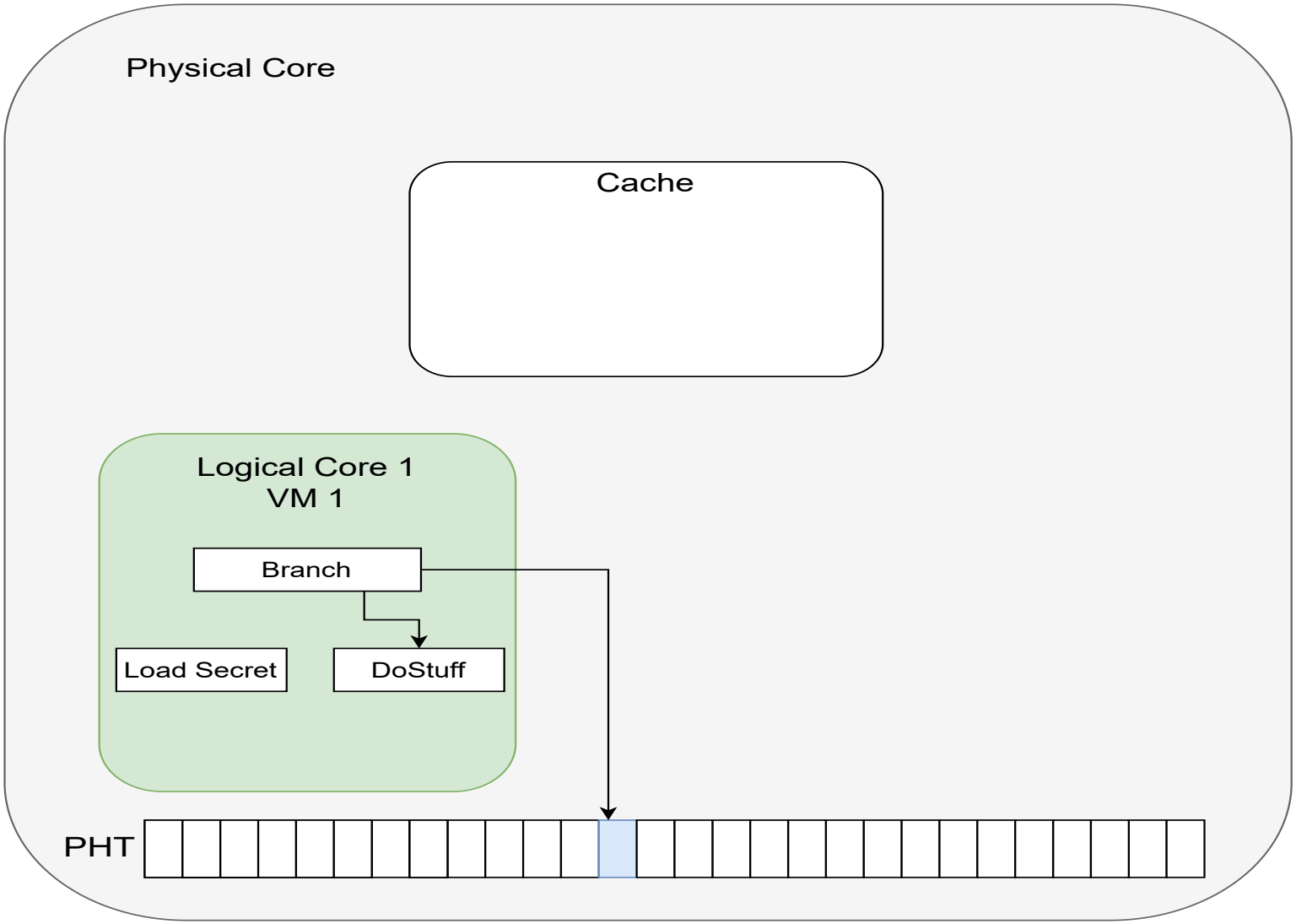


Figure: Spectre PoC (Active)

Spectre PoC (Proof of Concept)

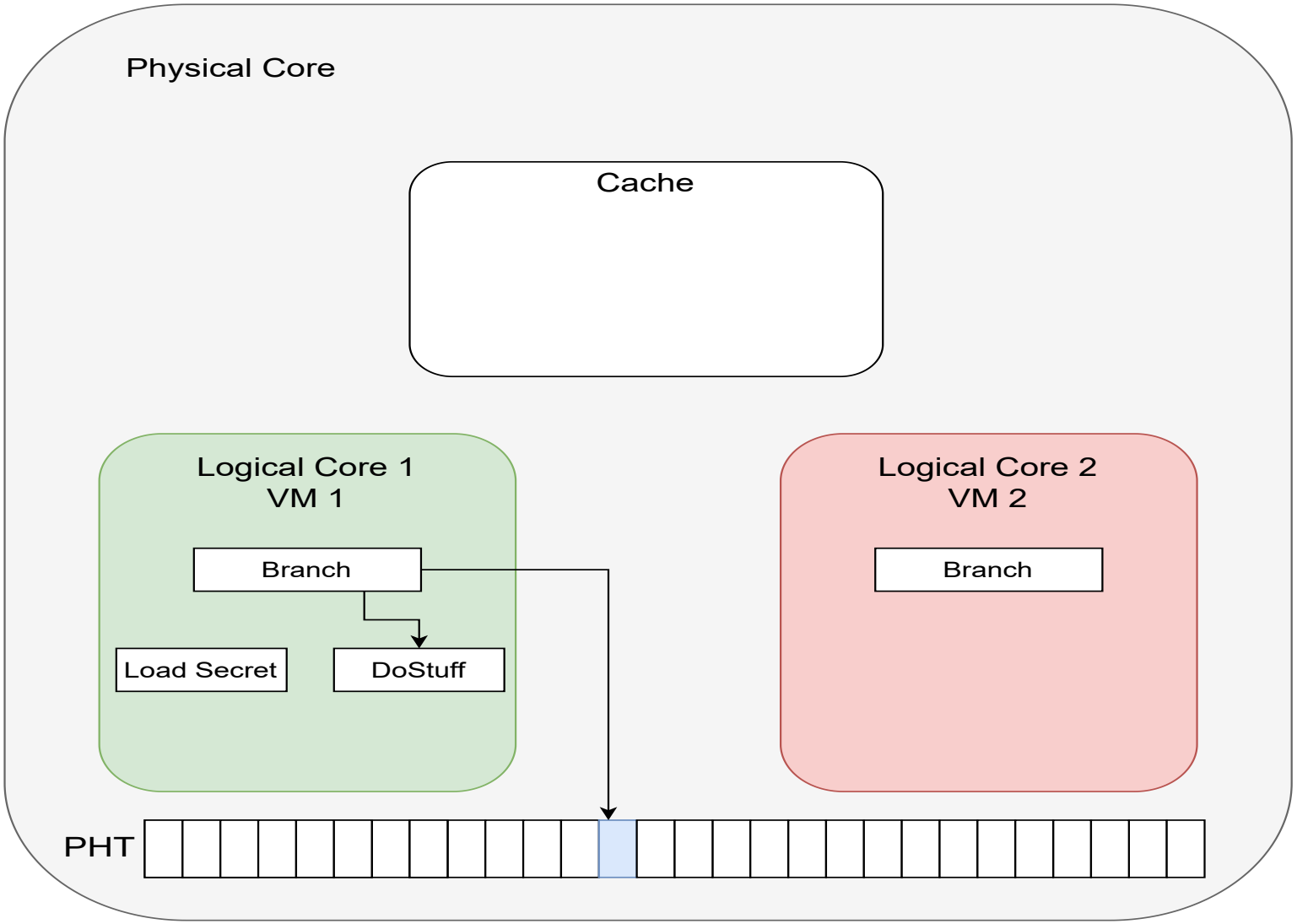


Figure: Spectre PoC (Active)

Spectre PoC (Proof of Concept)

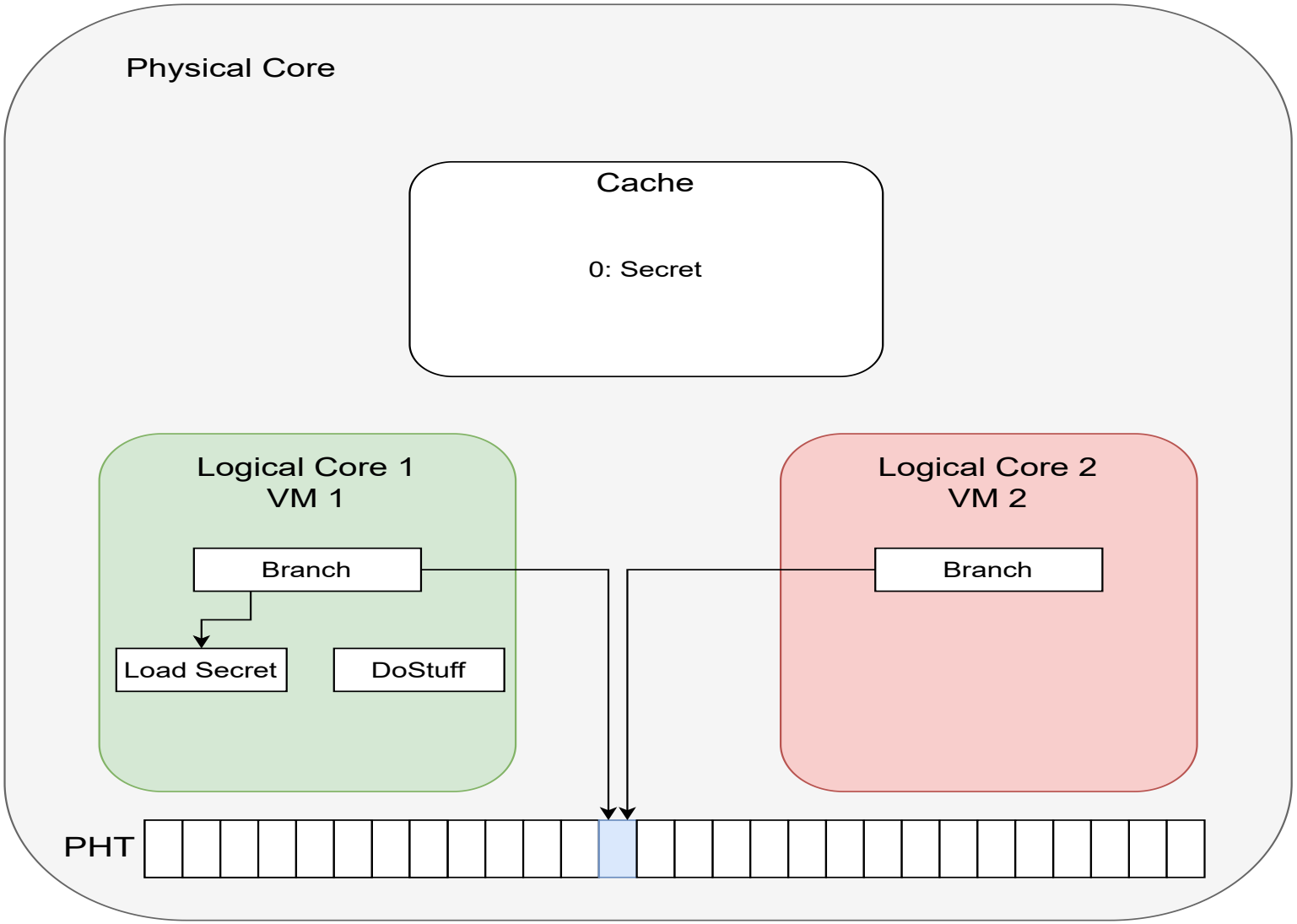


Figure: Spectre PoC (Active)

Leakage on the same logical core

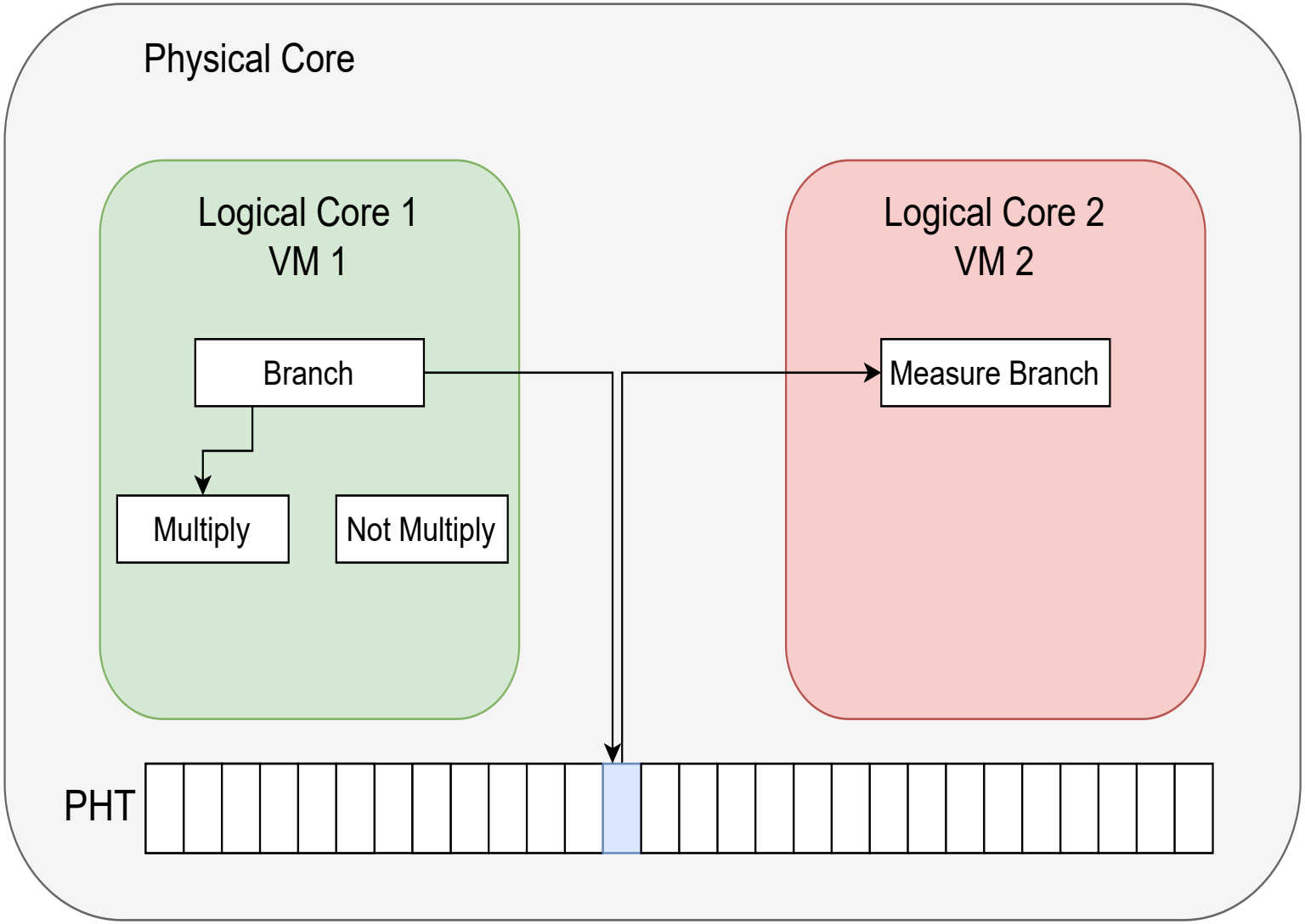


Figure: Spectre PoC (Passive)

Conclusion

- Attacks mostly theoretical yet
- Learning: Avoid shared resources
- Knowledge on branch prediction can help with defense measures

Conclusion

- Attacks mostly theoretical yet
- Learning: Avoid shared resources
- Knowledge on branch prediction can help with defense measures

References I

Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss.
A systematic evaluation of transient execution attacks and defenses.
In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 249–266. USENIX Association, 2019.

Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom.
Spectre attacks: Exploiting speculative execution.

References II

In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019.

André Seznec.
The L-TAGE branch predictor.
J. Instr. Level Parallelism, 9, 2007.

André Seznec.
A new case for the TAGE branch predictor.
In Carlo Galuzzi, Luigi Carro, Andreas Moshovos, and Milos Prvulovic, editors, *44rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2011, Porto Alegre, Brazil, December 3-7, 2011*, pages 117–127. ACM, 2011.

References III

André Seznec and Pierre Michaud.
A case for (partially) tagged geometric history length branch prediction.
J. Instr. Level Parallelism, 8, 2006.

Hosein Yavarzadeh, Mohammadkazem Taram, Shravan Narayan, Deian Stefan, and Dean Tullsen.
Half&Half: Demystifying Intel's directional branch predictors for fast, secure partitioned execution.
In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 22-26, 2023*, pages 1220–1237. IEEE, 2023.

Non-branch Instructions

Algorithm 3: *Target* program

```
1  setCore(6)
2  i ← 0
3  while experimentOn do
4      normalizeHistory()
5      instructions_1
6      //Always not taken
7      if True then
8          nop
9      i ← i + 1
10     nop_pattern...
```

Algorithm 4: *Injector* program

```
1  setCore(14)
2  i ← 0
3  while experimentOn do
4      normalizeHistory()
5      instructions_2
6      //Always taken
7      if False then
8          nop
9      i ← i + 1
```

Content of a PHT entry

Algorithm 5: Experiment pseudocode to test what kind of saturating counter is used.

```
1  param_1 ← args[0] //Assign first passed argument
2  i ← 0
3  while experimentOn do
4      normalizeHistory() //Ensures same history each iteration
5      //Cycling around the saturating counter
6      if ⌊i/param_1⌋ mod 2 then
7          nop
8      i ← i + 1
```

Content of a PHT entry

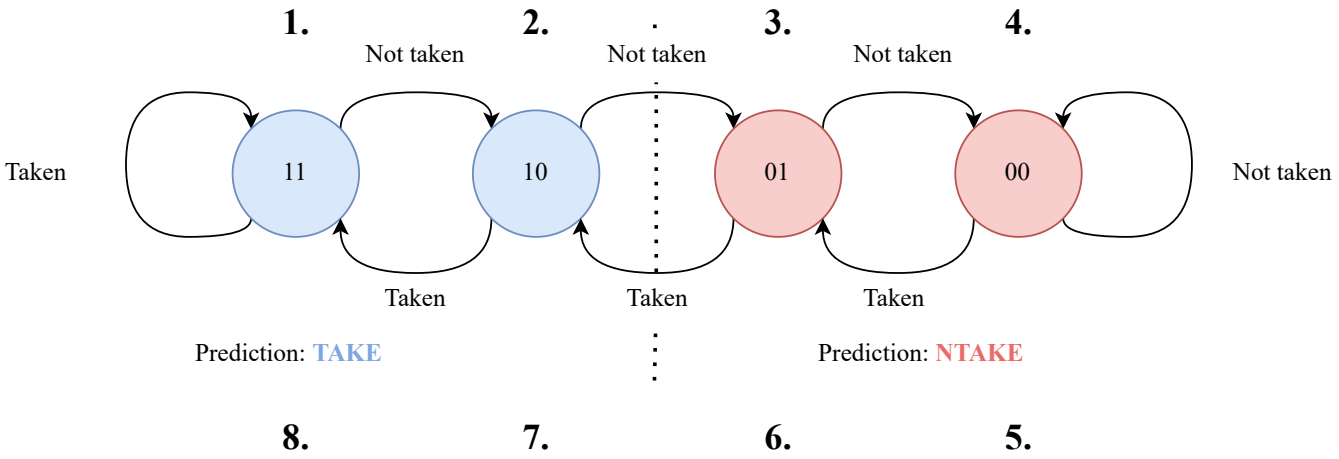


Figure: 2-bit saturating that we cycle around

Content of a PHT entry

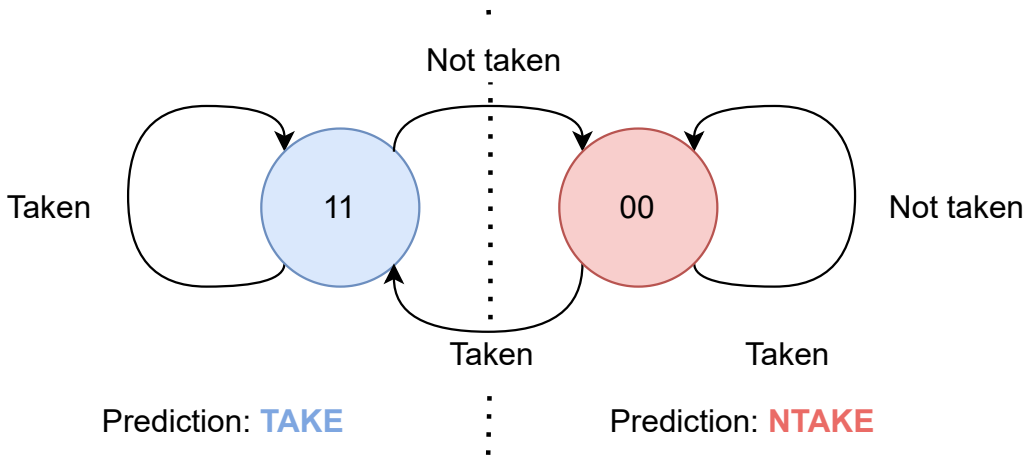


Figure: 1 Bit Counter

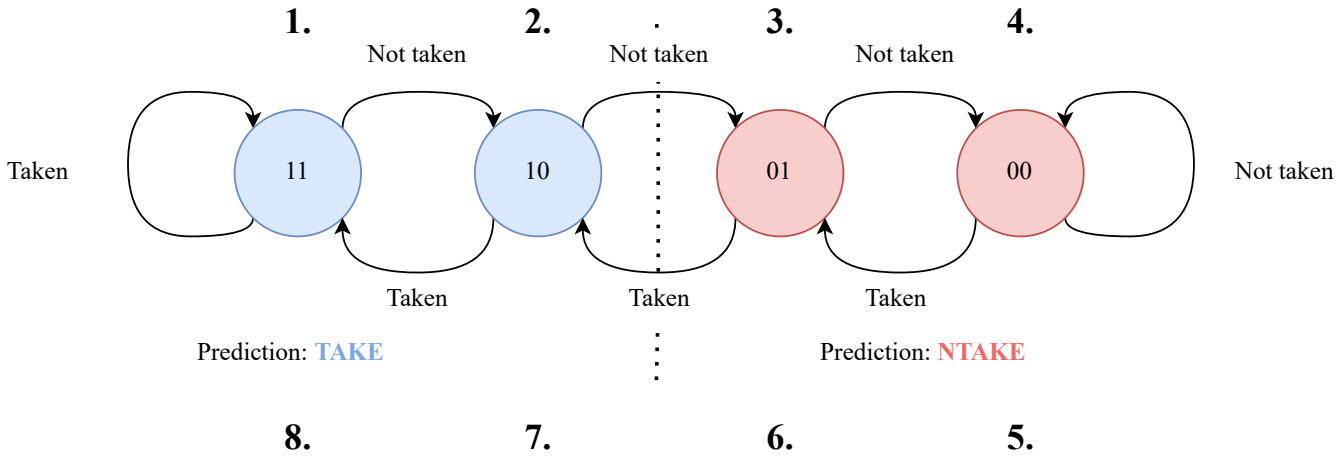


Figure: 2-bit saturating that we cycle around

Result

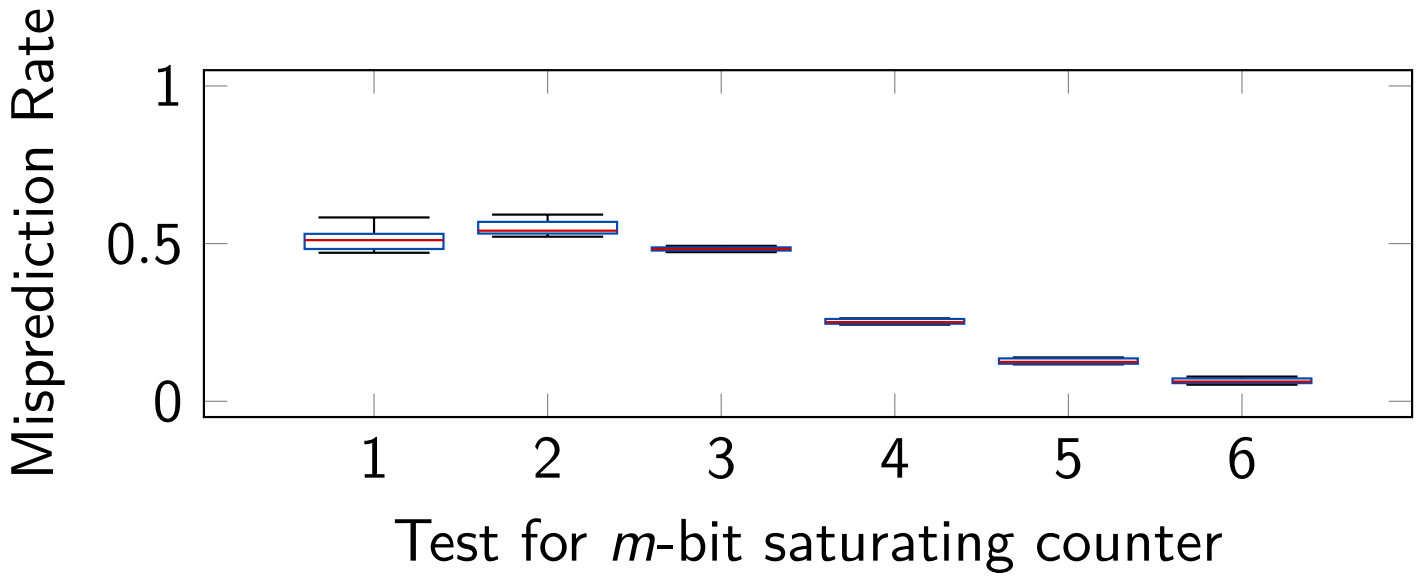


Figure: Results to show which saturating counter is used. They indicate a 3-bit saturating counter as results have a misprediction rate of around 0.5 for the first three values and then halve.

TAGE Predictor

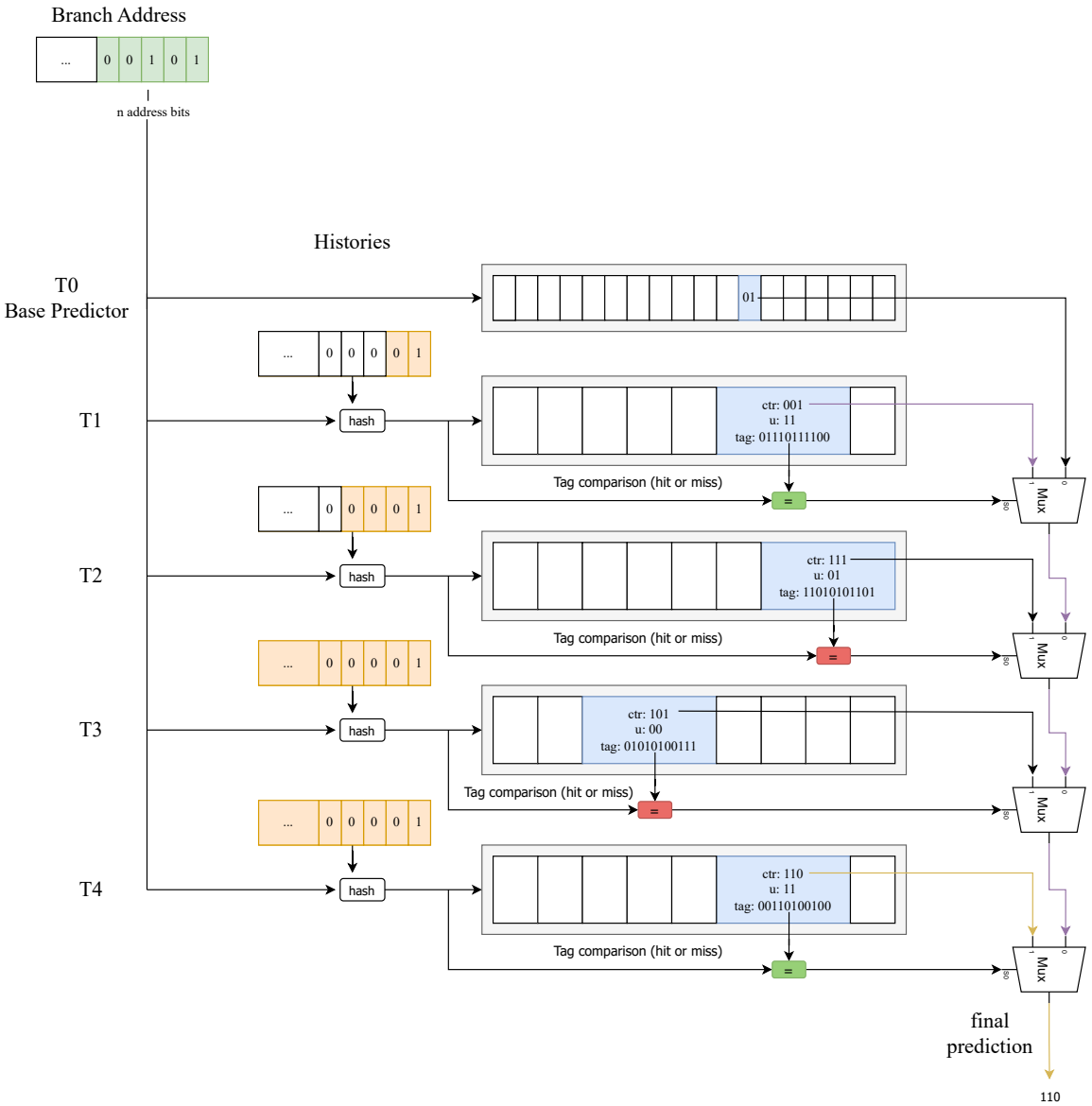


Figure: TAGE predictor illustration. Used by AMD Zen 2 processors (2019) e.g. [SM06, Sez07, Sez11]