# OCCUPY+PROBE

## Cross-Privilege Branch Target Buffer Side-Channel Attacks at Instruction Granularity

NDSS *Best paper award 2026*

Duzés Florian

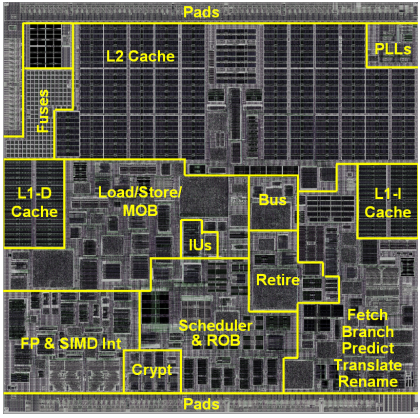
27/06/2026

# Summary

# 01
# Introduction

# Intel Processors

Figure – Die of a VIA Isaiah Architecture



# Intel Processors

Figure – Die of a VIA Isaiah Architecture


- Branch Prediction Unit (BPU)

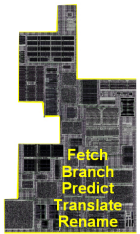
# Intel Processors

Figure – Die of a VIA Isaiah Architecture


- Branch Prediction Unit (BPU)
  - Branch Target Buffer (BTB)

# Intel Processors

Figure – Die of a VIA Isaiah Architecture




- Branch Prediction Unit (BPU)
  - Branch Target Buffer (BTB)
- Lots of attacks (x17 ref)
  - Heavily studied, in depth reverse-engineering

# Intel Processors

Figure – Die of a VIA Isaiah Architecture


- Branch Prediction Unit (BPU)
  - Branch Target Buffer (BTB)
- Lots of attacks (x17 ref)
  - Heavily studied, in depth reverse-engineering
  - 11th-generation (2020) set "hardware isolation" mechanism

# New attack : OCCUPY+PROBE

**1rst limitation passed — Isolation hardware (Intel 11$^{e}$ gen+)**

User programs cannot use BTB kernel entries

**2nd limitation passed - instruction granularity**

Previous attacks (*end 2000 - 2024*) observe the set of entries and cannot distinguish which branch got executed

# 02
# Background

# Branch Target Buffer

## Usage

The Branch Target Buffer (BTB) is a key component of the Branch Prediction Unit (BPU), recording executed branch instructions *to assist* in predicting future control flow.

# BTB structure

- BTB : associative cache set
- Each branch generate a **tag** / **index** / **offset**
- Then, on the **index** we store : **tag**, **offset**, **lower bits of the target adress**

# BTB structure

```
0x403403 e9 99 fe ff ff :    jmp     0x4032a1
```

- BTB : associative cache set
- Each branch generate a **tag** / **index** / **offset**
- Then, on the **index** we store : **tag**, **offset**, **lower bits of the target adress**

# BTB structure

- BTB : associative cache set
- Each branch generate a **tag** / **index** / **offset**
- Then, on the **index** we store : **tag**, **offset**, **lower bits of the target adress**

```
0x403403 e9 99 fe ff ff :    jmp     0x4032a1
```

```
0x403407 -> 00000000010000000011010000000111
  00                [31:30]
    00000001        [29:22] \ ( XOR ) tag 0x01
      00000000      [21:14] /
        110100000   [13: 5] index          0x1A0
          00111     [ 4: 0] offset         0x07
```

# BTB structure

- BTB : associative cache set
- Each branch generate a **tag** / **index** / **offset**
- Then, on the **index** we store : **tag**, **offset**, **lower bits of the target adress**

```
0x403403 e9 99 fe ff ff :    jmp     0x4032a1
```

```
0x403407 -> 00000000010000000011010000000111
  00                [31:30]
    00000001        [29:22] \ ( XOR ) tag 0x01
      00000000      [21:14] /
        110100000   [13: 5] index          0x1A0
          00111     [ 4: 0] offset         0x07
```

```
0x403407 : BTB entry
 ____________________________________________
|  0x1A0   | 0x01  |  0x07  | 0x4032a1 / 0x2a1 |
|  index   |  tag  | offset |      target      |
|__________________________________________|
```

# What was already in place

**Access-based**

Similar to FLUSH+RELOAD

- You create a dummy branch colliding with victim branch
- Read prediction result
- Ex : Branch Shadowing, BunnyHop-Reload, JumpOverASLR

**Eviction-based**

Similar to PRIME+PROBE

- Attacker primes an entire BTB set
- Victim use an entry → you can observe which one get modified
- Ex : Acıiçmez et al. (2006 - 2007), BunnyHop-Probe

**Solved** with hardware isolation ($11^{e}$ gen+)

# Timing measurement

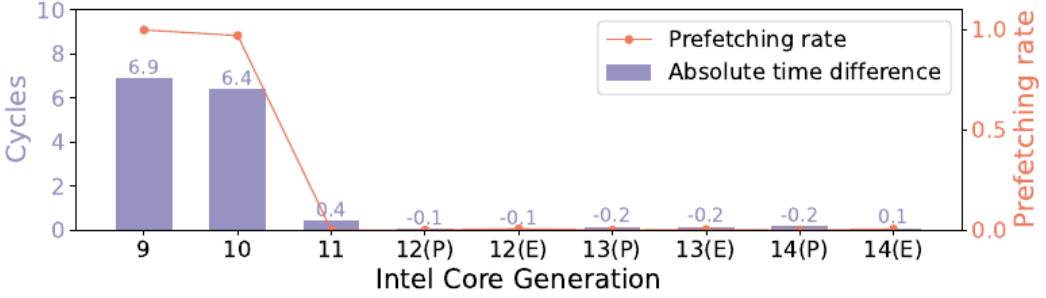



Figure – Cross-privilege BTB sharing across Intel Core generations.

# Read BTB via Instruction Prefetch

- BTB not directly accessible from user space

# Read BTB via Instruction Prefetch

- BTB not directly accessible from user space
- BUT, the predicted target goes into the **Instruction Cache** (i-cache)
- **Bridge BTB** $\rightarrow$ **I-Cache**; when a branch is getting updated

## Protocol to OCCUPY+PROBE

1. Generate a colliding branch to `TARGET`
2. Run aliased `PROBE` branch
3. Measure access time

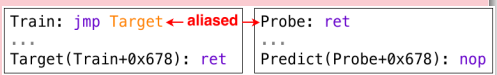
$\sim$60 cycles $\Rightarrow$ **hit** $\sim$290 cycles $\Rightarrow$ **miss** (entry replaced)

# Read BTB via Instruction Prefetch

- BTB not directly accessible from user space
- BUT, the predicted target goes into the **Instruction Cache** (i-cache)
- **Bridge BTB** $\rightarrow$ **I-Cache**; when a branch is getting updated

## Protocol to OCCUPY+PROBE

1. Generate a colliding branch to `TARGET`
2. Run aliased `PROBE` branch
3. Measure access time




$\sim$60 cycles $\Rightarrow$ **hit** $\sim$290 cycles $\Rightarrow$ **miss** (entry replaced)

# 03
# OCCUPY+PROBE

# Workflow of the attack

## ① Phase Occupy — userspace

Build a occupy branch with the lasts 36 bits = target kernel branch
⇒ same index + tag + offset ⇒ BTB entry inserted

## ② Victim execution — kernel

**Branch executed** ⇒ occupy entry **remplaced**
**Branch not executed** ⇒ no update ⇒ occupy entry **persist**

## ③ Phase Probe — userspace

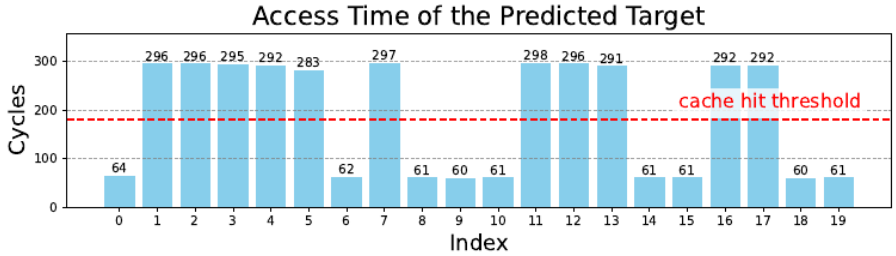
`PROBE` branch aliased executed → read in the I-Cache
**BTB miss** (∼290 cycles) ⇒ branch executed ⇒ secret = 0
**BTB hit** (∼60 cycles) ⇒ branch not executed ⇒ secret = 1

# 04
# Evaluation

# Bypass hardware isolation

- 1000 secret bits randomly generated
- Kernel branch triggered via `ioctl`

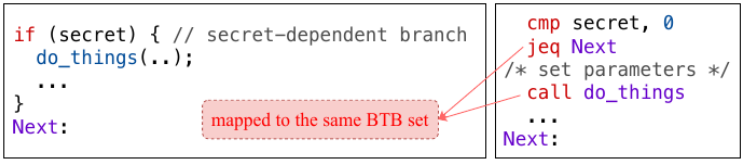
| Processeur | BTB P+P | OCCUPY+PROBE |
|---|---|---|
| i7-9700 | 100% | 100% |
| i7-11700K | 85.5% | 96.4% |
| i7-12700K | 100% | 99.4% |
| i9-14900K | 100% | 100% |



# Instruction granularity vs BTB PRIME+PROBE

## Problem for BTB PRIME+PROBE

CDB = Confusing Dummy Branch — branch modified in the same BTB set




(a) Source Code (b) Assembly

| Processeur | BTB P+P | OCCUPY+PROBE |
|---|---|---|
| i7-9700 | 48.9% | 100% |
| i7-11700K | 49.1% | 91.9% |
| i7-12700K | 48.9% | 100% |
| i9-14900K | 48.9% | 100% |

# Found the KASLR

## KASLR

KASLR aims to mitigate code-reuse attacks and other similar threats, which rely on specific kernel runtime addresses, by randomly generating kernel addresses at boot time. In the Linux kernel implementation, address bits [29 :21] are randomized, with all kernel addresses maintaining the same offset from the base.

**Principle** : Have a correct aliased $\rightarrow$ guess the kernel address

## Resultats (i7-11700K)

- 10 reboots, 100 tentatives par reboot = 1000 essais $\Rightarrow$ success rate : **97.5%**
- On i9-14900K : reduce from $2^9$ to $2^6$ possibilities ($\times 8$)

# 05
# Real case : Linux Kernel Crypto API

# Target : RSA in `mpi_powm`

- Linux Kernel Crypto API : crypto library in the kernel
- RSA with CRT optimization : $m_p = c^{d_p} \bmod p$ and $m_q = c^{d_q} \bmod q$
- Algorithm square-and-multiply $\rightarrow$ **secret branch** :

```
if ((mpi_limb_signed_t) e < 0) { multiply(...) }
```

- Each bit of $d_p$ & $d_q$ infer on the branch direction
- Leakage of the result on each iteration $\Rightarrow$ leak the secret key

# HOWTO leaked secret bit

**Problem**

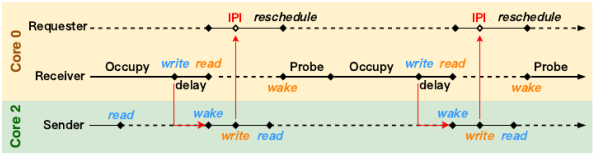
Kernel option `CONFIG_PREEMPT_VOLUNTARY`
⇒ not interruption forced from user space

**Solution : `eventfd` + IPI (Inter-Processor Interrupt)**

Three threads : requester (run RSA), receiver (pirate, same core), sender (core+1)

1. `Receiver` do occupy, write on `synefd`, halt on `intefd`
2. `Requester` do the critic computation (RSA)
3. `Sender` write on `intefd` → IPI → reset `preempt_count` = 0
4. `Requester` atteint `cond_resched` → context switch → `Receiver` have the hand
5. `Receiver` probe the entry → secret bit is leaked

# HOWTO leaked secret bit




**Solution : `eventfd` + IPI (Inter-Processor Interrupt)**

Three threads : requester (run RSA), receiver (pirate, same core), sender (core+1)

1. `Receiver` do occupy, write on `synefd`, halt on `intefd`
2. `Requester` do the critic computation (RSA)
3. `Sender` write on `intefd` → IPI → reset `preempt_count` = 0
4. `Requester` atteint `cond_resched` → context switch → `Receiver` have the hand
5. `Receiver` probe the entry → secret bit is leaked

# Results

- Hard/Soft : Intel i9-14900K, Ubuntu 22.04.5, kernel 6.8.0-51
- $2 \times 10^6$ tries, executed 10 times
- Redundant bits : monitoring the execution

| **Metric** | **Value** |
|---|---|
| Maximal precision (best execution) | $> 99\%$ |
| Average precision (150 executions) | **98.6%** |
| Standard deviation | 4.7% |

# 06
# Conclusion

# Mitigations

## Hardware

- Remove DR cross-privilege mechanism → reenforce BTB isolation
- Store privilege level in the BTB entry
- Flush BTB on switch context → overhead significatif

## Software

- **Data-oblivious programming** : do not use a secret as a branch
- Execute sensibles instructions on **E-cores** (Gracemont) : DR missing
- Delete `cond_resched` from `mpi_powm` (delay tradeoff)

*Intel confirmed : adequate mitigation measures (disclosure 18/04/2025)*

# Conclusion

1. **Reverse engineering** of the BTB update mechanism based on offset
   → 4 mechanisms (DR / IA / I / A), documentation for each case
2. **OCCUPY+PROBE** : first BTB cross-privilege attack
   at instruction granularity on CPU with hardware isolation
   → with all hardware mitigations against Spectre-v2
3. **End-to-end attack** on Linux Kernel Crypto API
   → private RSA key with a precision of **98.6%**

`https://github.com/CPU-Security/Occupy_Probe`