UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

ATACURI SPECULATIVE

Absolvent
Radu Ștefan-Octavian

Coordonator științific
Conf.univ.dr.ing. Paul Irofti

București, iunie 2021

Rezumat

Computerele moderne folosesc tehnici de optimizare precum *execuție out-of-order* și *branch prediction*. *Meltdown* și *Spectre* sunt două atacuri care exploatează efectele secundare apărute la nivel microarhitectural în urma optimizărilor menționate. Prin intermediul acestora un atacator poate citi date private din zone arbitrare din memorie, fară privilegii și fară a exploata niciun bug de natură software. *Intel*, *AMD* și *ARM* au fost forțate în urma divulgării acestor atacuri să își schimbe designul procesoarelor în încercarea de a mitiga vulnerabilitățile la nivel hardware. În ciuda soluțiilor implementate, la jumătatea anului 2022, *Spectre* afectează în continure majoritatea computerelor din lumea întreagă și rămâne un pericol pentru utilizatori și un subiect de mare interes pentru cercetători. În această lucrare vor fi prezentate particularitățile celor două atacuri, și o implementare demonstrativă 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 și *Spectre* fac parte din clasa largă a Atacurilor de tip *Side-Channel*, care exploatează mai degrabă efecte secundare rezultate din implementarea unui sistem, decât defecte în implementarea algoritmilor ce rulează pe acel sistem. De-a lungul timpului au apărut numeroase atacuri de tip *Side-channel*:

- Timing attacks. Aceste atacuri au ca scop compromiterea unui sistem prin intermediul analizei statistice a timpilor de execuție ai unui algoritm, pe diverse seturi de date de intrare (eg. compromiterea unui sistem de criptare RSA la distanță [3])
- Cache attacks. Aceste atacuri se bazează pe abilitatea unui atacator de a monitoriza accesările unor zone de memorie partajate de către o victimă, și deducerea unor concluzii din modul în care aceste accesări influențează cache-ul procesorului. În cazuri extreme s-a demonstrat că se pot divulga chei criptografice secrete prin intermediul acestor tehnici [25].
- Data remanence attacks. Aceste atacuri implică accesul asupra unor date după o presupusă ștergere a acestora în prealabil. Un exemplu clar este atacul de tip *Cold Boot* [15] în care un atacator cu acces fizic la o mașină poate citi întreaga memorie RAM după efectuarea unui resetări a computerului.
- Rowhammer are un loc special în clasa de atacuri de tip *Side-Channel*. Prin accesul repetat al unei zone de memorie s-a observat că încărcătură electrică poate afecta zonele adiacente, provocând scurgeri de informație. Pe baza acestei tehnici s-au putut construi atacuri de tip escalare de privilegii [30].
- Mai există și alte tipuri de atacuri care exploatează consumul energetic, câmpul electromagnetic generat de componentele electronice, sau chiar și sunetul generat de sistem.

Meltodwn și *Spectre* se folosesc de idei asemănătoare cu cele menționate în *Timing Attacks* și în *Cache Attacks* pentru a crea un canal de comunicare ascuns (*covert channel*). Pe acest canal se transmit informații accesate în mod malițitos prin intermediul unor hibe în implementarea la nivel hardware a computerelor moderne. Se va descrie cum exploatarea acestor defecte este posibilă prin intermediul execuției speculative în cadrul procesorului. Execuția speculativă presupune execuția în avans a instructiunuilor pentru a salva timpi morți și a îmbunătăți performanța. Fluxul de execuție poate fi manipulat în așa fel încât speculativ să se execute instrucțiuni care nu s-ar executa vreodată în cadrul fluxului normal de execuție al programului. Pentru menținerea consistentei și corectitudinii rezultatelor obținute în urma execuției algoritmului, rezultatele instrucțiunilor executate speculativ în mod **eronat** sunt omise, iar starea internă este resetată la una anterioară, dar corectă. În momentul resetării, starea cache-ului nu este și ea resetată, iar acest fapt poate fi exploatat pentru obținerea unor informații secrete, care ulterior vor putea fi transmise prin intermediul unui canal ascuns (*covert channel*), menționat anterior.

1.2 Motivația Personală

Din fire, încerc mereu să aprofundez cât mai în amănunt subiectele care mă interesează, pentru a înțelege în profunzime. Natural – consider eu – am ajuns atras de subdomeniul securității care presupune o înțelegere profundă a sistemelor informatice. Dintre atacurile cibernetice, *Spectre* și *Meltdown* mi-au stârnit interesul și curiozitatea pe deoparte prin rezultatele remarcabile obținute și pe de altă parte prin nivelul ridicat de subtilitate al vulnerabilitatilor exploatate.

1.3 Scopul Lucrării

Lucrarea de față este rezultatul studiului personal al materialelor originale care expuneau aceste atacuri. Aceasta are drept scop explicarea clară, dar succintă a *Meltdown* și *Spectre*, într-un mod accesibil cititorilor interesați. Este de asemenea prezentată și implementarea personală, aferentă a unuia dintre atacurile discutate.

1.4 Contribuții personale

În urma studiului acestor tipuri de atacuri am reușit să realizez o implementare cu scop demonstrativ a *Spectre-v1* într-un scenariu mult mai realist față de varianta prezentată în lucrarea de cercetare [22]. Implementarea mea propune un scenariu în care un proces neprivilegiat citește în mod neautorizat zone de memorie dintr-un proces victimă, fără

acces la spațiul acestuia de memorie, ci doar la o zona limitată de memorie partajată. În scopul realizării acestui exemplu au fost utilizate tehnicile și noțiunile ce vor fi descrise pe parcursul acestei lucrări. Sursele sunt publice și pot fi vizualizate la următoarele adrese: victima, atacator.

1.5 Structura Lucrării

Lucrarea este împărțită în următoarele capitole:

1. Introducere - se prezintă o viziune de ansamblu asupra atacurilor ce urmează să fie prezentate, un mic istoric al tehiniclor, și motivația personală pentru realizarea acestei lucrări.

2. Preliminarii - se prezintă noțiuni de *Sisteme de Operare*, *Arhitectura Sistemelor de Calcul* și *Securitate* relevante înțelegerii atacurilor discutate.

3. Atacul Meltdown - se discută detalii de funcționalitate, metode de mitigare și detalii de reproducere a atacului meltdown.

4. Atacuri Spectre - se ilustrează deosebirile față de Meltdown, precum și detalii specifice de funcționalitate pentru cele două variante principale ale Spectre. Sunt prezentate metode de mitigare și starea actuală a vulnerabilităților.

5. POC - Spectre-v1 - se prezintă o implementare demonstrativă a *Spectre v1*

6. Concluzii - se sumarizează cele discutate pe parcurs și se menționează direcțiile viitoare

Capitolul 2

Preliminarii

2.1 Noțiuni de Sisteme de Operare

2.1.1 Kernel

După cum sugerează și numele, kernel-ul este componenta principală a unui sistem de operare, care servește drept interfață între hardware și software. Kernel-ul îndeplinește patru roluri principale în cadrul sistemului de operare:

- gestionarea eficientă a memoriei de pe sistem
- programarea proceselor pe *CPU*
- gestionează driverele de hardware, astfel acționând ca un mediator între hardware și restul proceselor
- comunică cu restul proceselor prin intermediul unui interfețe a apelurilor de sistem (SCI)

Codul rulat în Kernel este izolat de restul codului de pe sistem. Acesta întreprinde acțiuni în mod privilegiat cu drepturi depline de acces asupra hardware-ului. Codul obișnuit de pe sistem funcționează în userland, și rulează cu acces restricționat asupra resurselor, având acces la acestea doar prin interfață sigură de comunicare cu Kernel menționată mai sus (SCI) [18].

2.1.2 Race Condition / Întrecere la rulare

Un *race-condition* apare la nivel de cod în momentul în care funcționarea corectă a unui program depinde de ordinea de execuție sau de sincronizarea temporală a mai multor fire de execuție paralele (*thread-uri*, sau procese). În general aceste situații apar în cazul în care firele de execuție vizează simultan o resursa comună. Pentru evitarea bug-urilor

în aceste situații, secvențele de operații executate asupra resursei comune, numite zone critice, trebuie executate într-un mod **reciproc exclusiv**.

La nivel microarhitectural pot apărea *race-condition-uri* în timpul execuției speculative 2.2.3 în urma cărora pot apărea efecte secundare exploatabile.

2.1.3 IPC

Interprocess comunication, sau *IPC* face referire la mecanismele puse la dispoziție de sistemul de operare pentru comunicarea între procese și gestionarea datelor comune. Principalele metode folosite în practică și amintite în această lucrare sunt:

- Fișiere. Comunicarea prin intermediul unor fișiere accesibile tuturor proceselor implicate.
- Semnale. Mesaje transmise de la un proces la altul, în general sub formă de instrucțiuni, corespunzând unui protocol stabilit anterior.
- Memorie partajată. Bloc de memorie la care au acces mai multe procese și prin intermediul căruia pot comunica.

2.1.4 Excepții de sistem

O excepție reprezintă o schimbare bruscă în rularea programului ca răspuns la o schimbare bruscă în starea procesorului. Exemple de excepții la nivel de aplicație ar fi: cereri de alocare a memoriei pe heap, cereri de input/output, încercări de împărțire la 0, încercări de accesare a memoriei în afara limitelor impuse de memoria virtuală dedicată procesului, etc. Excepțiile se împart în mai multe categorii: *interrupts*, *traps*, emphfaults, *aborts*. Categoria care va fi adusă în discuție în prezenta lucrare este cea de-a treia, *faults*. Defectele (*faults*), sunt erori (posibil recuperabile de către sistemul de operare) cauzate de o aplicație (eg. accesarea unei zone de memorie asupra căreia programul nu are drepturile necesare — *segmentaion fault* —, accesarea unei unor date care nu sunt încărcate în memorie — *page fault*) [29].

2.1.5 Procese

Procesul reprezintă cea mai primitivă unitate de alocare a resurselor de sistem și este o instanță activă a unui program [26]. Programul în acest caz nu este nimic mai mult decât un fișier executabil stocat pe mașină. Un program nu poate rula decât în contextul unui proces, care constă în id-ul procesului (*PID*), spațiul de adrese (*TEXT, DATA, STACK, HEAP, BSS, etc*), starea procesului (starea registrilor), etc. [29]. Procesele sunt izolate între ele, fiecare având dedicat spațiul său propriu de adrese virtuale de memorie. Implicit

procesele nu împart resurse între ele, dar pot comunica între ele partajând resurse în mod intenționat când acest obiectiv este dorit.

2.1.6 Memorie Virtuală

Procesoarele folosesc adrese virtuale de memorie și un mecanism de traducere a acestora în adrese fizice pentru a asigura izolarea și separarea proceselor între ele. Fiecare zonă de memorie virtuală este împărțită în multiple pagini (cea mai comună dimensiune este de 4096 de bytes). Fiecare pagină virtuală este mapată prin intermediul tabelelor de traducere a paginilor (*translation table*), către corespondentul fizic. În procesor există un registru dedicat pentru reținerea tabelului de traducere utilizat la un moment dat și care se schimbă la fiecare schimbare de context. În consecință, ficare proces își poate accesa doar zona să virtuala de memorie.

Tabelele de traducere mai au și rolul de a asigura separarea între zona de memorie dedicată utilizatorului și zona de memorie dedicată kernel-ului în cadrul fiecărui proces. În timp ce zona de memorie dedicată utilizatorului poate fi accesată de aplicația care rulează în procesul curet, zona de kernel poate fi accesată doar dintr-un mod de rulare privlegiat. Restricțiile acestea sunt precizate în tablele de traducere prin intermediul unui bit (*user/supervisor bit*), iar respectarea acestora este asigurată de sistemul de operare. Mai este important de notat faptul că zona pentru kernel în general mapează întreaga memorie fizică din cauza necesității de execuție a diverselor operații asupra acesteia (eg. scriere, sau citire de date).

Pe parcursul acestei lucrări vor fi expuse atacuri prin care limitele impuse de tablele de traducere au fost ocolite, putându-se accesa memoria kernel, și implicit toată memoria fizică prin intermediul unui proces neprivilegiat, cât și zone de memorie ale altor procese, prin intermediul unor pagini partajate.

2.2 Noțiuni de Arhitectura Sistemelor de Calcul

2.2.1 SMAP și SMEP

SMAP și SMEP sunt două caracteristici cu roluri în securizarea sistemelor prin izolarea mai bună a Kernel-ului de spațiul utilizatorului (*userland*). Acestea sunt implementate în cadrul memoriei virtuale și activate prin setarea biților corespunzători (20 și 21) din registrul `CR4` pe arhitectura *Intel*).

SMAP este o caracteristică care presupune restricția accesului asupra anumitor zone de memorie din *userland* în modul de execuție Kernel. În timp ce mecanismul de protecție

este activat, orice tentativă de acces a zonelor protejate va duce la declanșarea unei excepții. Rolul SMAP este de a împiedica programele malițioase din a manipula Kernelul să acceseze instrucțiuni, sau date nesigure din spațiul utilizatorului [7].

SMEP este o caracteristică implementată cu scopul de a complementa SMAP. Are rolul de a preveni execuția neintenționată a unor fragmente de cod în spațiul user-ului, prin restricționarea dreptului de execuție asupra acestora. Diverse atacuri precum cele de tipul *Privilege-Escalation* pot fi prevenite datorită acestor caracteristici.

2.2.2 Out-of-order Execution

În trecut procesoarele executau instrucțiunile în ordinea în care acestea erau preluate de la compilator, câte una pe rând. În multe situații instrucțiuni mai costisitoare blocau fluxul de execuție, iar procesorul devenea parțial inactiv. Procesoarele moderne se folosesc de o serie de tehnici grupate sub umbrela *Out-of-order Execution*, introduse pentru prima dată la mijlocul anilor 1990 [17], în urma unui algoritm dezvoltat de Tomasulo în 1967 [33] care permitea programarea dinamică a ordinii instrucțiunilor și alocarea acestora pe mai multe unități de execuție care rulează în paralel. Scopul acestei tehnici este utilizarea exhaustivă a resurselor disponibile pe procesor, pentru creșterea performanței. Datorită beneficiilor aduse, *Out-of-order Execution* a devenit o caracteristică indispensabilă a sistemelor moderne de procesare.

Această optimizare poate duce la situații în care unele instrucțiuni executate în avans trebuie respinse, iar starea programului resetată la una anterioară (din cauza declanșării unei excepții în urma accesării unei zone de memorie interzisă de exemplu). Aceste tipuri de instrucțiuni numite *Instrucțiuni Tranzitorii* stau la baza atacului *Meltdown* [23].

2.2.3 Execuție Speculativă & Prezicerea ramurilor de execuție

Execuția Speculativă presupune executarea codului în avans *out-of-order* în situații nesigure (spre exemplu în așteptarea determinării ramurii pe care va continua fluxul de execuție în cazul unei bifurcări). În cadrul acestei lucrări execuția speculativă vă fi exploatată primordial prin intermediul *Branch-Predictor-ului*.

Branch Processing Unit (*BPU*) din interiorul procesoarelor moderne încearcă să prezică, în cazul unei ramificări a fluxului de execuție (de exemplu o structură decizională – *if*), sau final de iterație (*for, while*), ramura corectă care vă fi urmată. În cazul în care fluxul de execuție stagnează la un astfel punct de bifurcare (de exemplu, în așteptarea încărcării din memorie a valorii unei variabile), instrucțiunile următoare se vor executa speculativ, urmând ramura prezisă de *BPU*. După ce execuția instrucțiunii care decide ramura corectă a execuției, rezultatele obținute speculativ sunt fie păstrate (caz în care se câștiga timp de rulare prețios) fie respinse, caz în care se revine la o stare anterioară. [22].

Branch predictor-ul are în general o acuratețe foarte ridicata, chiar de peste 95% [17], așadar executând speculativ s-au obținut îmbunătățiri considerabile de performanță. Cu toate acestea, în cazurile în care ramura de execuție nu este prezisă corect, se vor executa instrucțiuni care nu ar fi avut loc în cadrul execuției secvențiale, *în-order execution*. Bineînțeles, aceste instrucțiuni vor fi *rolled-back*, iar rezultatul final va fi cel așteptat, dar la nivel micro-arhitectural se pot observa și măsură niște efecte secundare neprevăzute ale acestor instrucțiuni executate *out-of-order*. Analizarea cu grijă a acestor efecte secundare stă la baza atacurilor de tip *Spectre* [22].

2.2.4 CPU Cache

Deoarece încărcarea valorilor din memoria RAM în CPU este foarte costisitoare, în cadrul procesoarelor există o ierarhie de zone de memorie foarte rapide, separate în linii de dimensiuni mici (de obicei între 16 și 128 de bytes), ce poartă denumirea de emphcache-uri [4]. După prima accesare a unei adrese din memorie, valoarea obținută este reținută în cache. Astfel, la accesări ulterioare ale aceleiași zone de memorie, timpul în care valoarea este încărcată este redus semnificativ. În final, prin citiri repetate ale valorilor din cache, se maschează încărcarea inițială din memorie, semnificativ mai lentă, și se câștiga timp prețios de execuție. Cache-urile sunt de obicei partajate între nucleele unui procesor, optimizandu-se astfel și performanța multi-core.

2.3 Noțiuni de Securitate

2.3.1 Atacuri asupra memoriei cache

Deoarece memoria cache este mult mai rapidă, prin intermediul unui ceas de mare precizie putem distinge între accesare din memorie și accesarea din *cache* a unei variabile, conform figurii 2.1.

Timpul de accesare al valorii corespunzătoare variabilei `value` poate fi calculat utilizând instrucțiunea `__rdtscp` specifică procesoarelor Intel. Aceasta permite citirea *time-stamp counter-ului* din procesor [27]. Prin doua măsurători ce încadrează dereferentierea pointer-ului către `value`, putem măsură numărul de ciclii de procesor necesari operației. Repetând experimentul de 10000 de ori și calculând media timpului de acces pentru fiecare caz, se obțin următoarele rezultate:

- încărcarea din memorie durează aproximativ 250 de ciclii și poartă numele de *cache miss*
- încărcarea din cache durează aproximativ 23 de ciclii și poartă numele de *cache hit*

Aceste diferență măsurabile sunt exploatate în cadrul diferitelor tehnici de atac asupra memoriei cache, printre care și *FLUSH and RELOAD* care vă fi discutat în continuare.

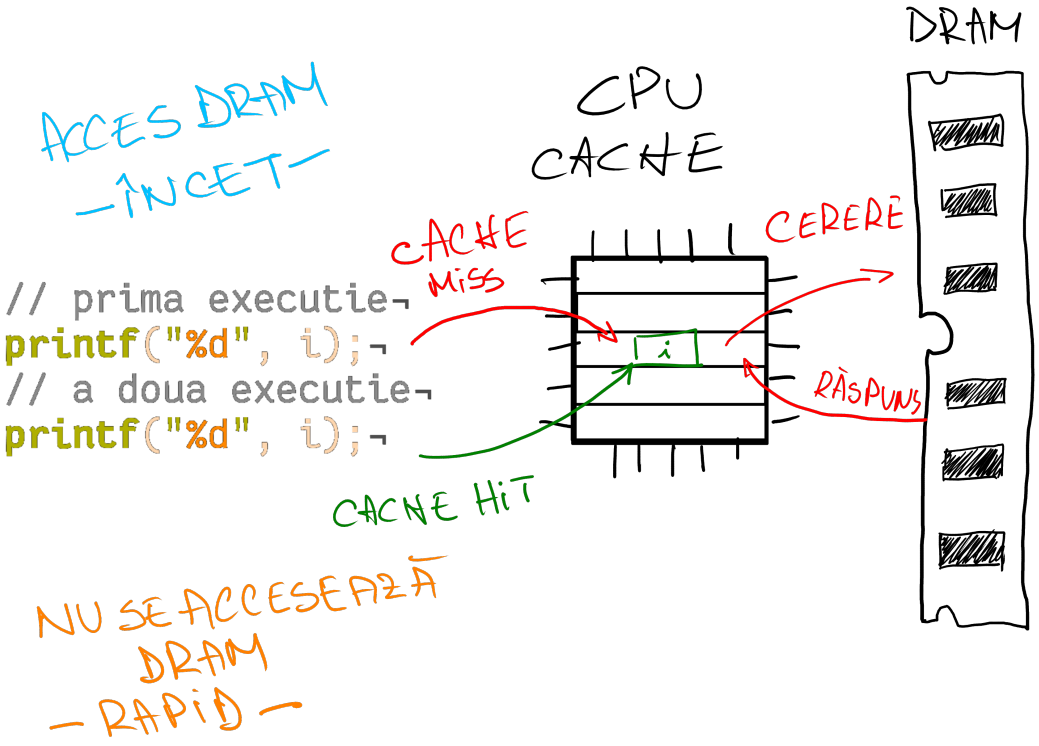

Figura 2.1: Cache hit vs Cache miss

Observație

Numărul de ciclii de procesor necesari execuției unui set de instrucțiuni diferă în funcție de sistem. Rezultatele ilustrate anterior sunt specifice unui sistem ce rulează o versiune la zi a Kernelului Linux în data de 22.05.2022 (`Linux 5.17.9-arch1-1 x86_64`), cu un procesor al producătorului *Intel*, modelul `i5-8250U` (mai multe specificații pe site-ul producătorului [20]), cu 8GB de memorie RAM tip DDR3.

FLUSH and RELOAD

O practică comună de reducere a memoriei utilizate este partajarea între 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 în lucrarea de cercetare în care a fost introdus atacul este acela al unei victime și al unui spion care împart o zonă partajata de memorie. Spionul se folosește de instrucțiunea `clflush` care invalidează liniile aferente unei zone de memorie din toată ierarhia cache-ului din procesor [6], iar apoi așteaptă o perioada scurta de timp. În final verifică daca la accesarea zonei respective obține un *cache hit* sau un *cache miss*, astfel aflând daca victima a accesat sau nu în fereastra respectivă de timp, zona de memorie urmărită. Repetând experimentul, s-au putut extrage informații suficiente pentru realizarea unor atacuri de succes asupra implementării de la vremea respectiva a unor algoritmi

criptografici (*OpenSSL*, *AES*), monitorizarea activității unui utilizator, etc. [34].

Alte tipuri de atacuri asupra memoriei cache

FLUSH and FLUSH se aseamănă cu *FLUSH and RELOAD*, diferența constând în faptul că spionul în loc de a măsură timpul de acces a zonei tintă, vă apela iarăși `clflush`. Execuția mai rapida vă corespunde unui *cache miss*, iar cea mai rapida unui *cache hit* [24].

EVICT and RELOAD folosește un *eviction set* pentru a elimina din cache zona de memorie tintă. Apoi, pentru măsurarea timpului se procedează identic ca la *FLUSH and RELOAD*. Tehnica se aseamănă în eficienta cu cele menționate anterior [24].

PRIME + PROBE constă într-o abordare diferita. Atacatorul umple toată zona partajata din cache (*PRIME*). Victima vă elimina valori încărcate de atacator în cache în timp ce rulează (*evict*). Atacatorul va măsura apoi timpul de acces pentru toată zona de cache (*probe*). În cazul în care observă un *cache hit*, constată că victima a accesat zona respectivă [24].

2.3.2 Covert-channel

Un *cover channel* reprezintă un mod, sau protocol de comunicare ascuns. Prin ascuns înțelegem că este foarte greu, sau chiar imposibil de detectat de către un administrator, sau alți utilizatori de pe sistem, întrucât nu presupune transmiterea de date într-un mod uzual. Utilizarea tehnicilor descrise anterior pentru extragerea diverselor informații, fac ca memoria cache să devină un astfel de canal de comunicare ascuns. Un astfel de mod de comunicare poate fi utlizat pentru transmiterea de informații între procese care în mod normal nu ar fi avut dreptul să comunice, conform politicilor de securitate prezente pe sistem [8].

2.3.3 Atacuri Speculative

Atacurile Speculative se bazează pe exploatarea tehnicii de optimizare numită *Execuție Speculativă* care, datorită avantajelor aduse în performata, este utilizată în prezent de majoritatea procesoarelor folosite în prezent. Atacurite se folosesc de această optimizare pentru a produce intenționat efecte secundare măsurabile, cu scopul de a accesa date în mod neautoriazat, prin intermediul unor canale secundare (*side channeles*) sau ascunse (*covert channels*). În continuarea acestei lucrări voi discuta particularitățile și implicațiile câtorva atacuri din această clasă care au avut un impact semnificativ asupra industriei în ultimii ani.

2.3.4 ROP

ROP [31] 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 [16]. 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.

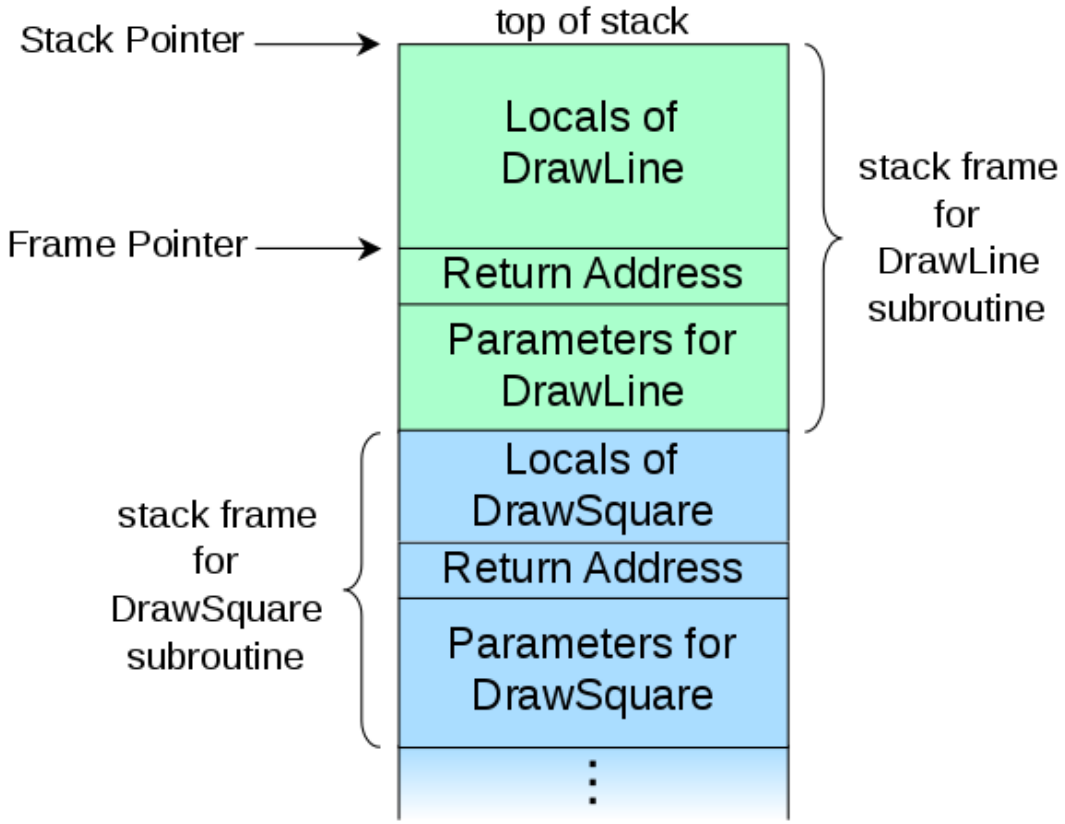

Figura 2.2: Stack Layout - ROP attack [32]

Capitolul 3

Atacul Meltdown

Meltdown, descoperit în 2017, este un atac care permite citirea întregii memorii de sistem (inclusiv a datelor personale și a parolelor), în ciuda mecanismele de protecție care asigură izolarea memoriei kernel de cea a unui utilizator neprivilegiat. Atacul crează un canal de comunicare ascuns pe baza efectelor secundare ale *Out-of-Order Execution*. Astfel, vulnerabilitatea depinde doar de tipul procesorului pe care rulează sistemul și este independent de software, sau tipul sistemului de operare. La momentul apariției Meltdown afecta orice utilizator al unui procesor modern de tip *Intel* produse începând cu 2011, și posibil alte mărci. Atacul a fost mitigat prin intermediul unor patch-uri software pe toate sistemele mari de operare (Windows, Linux, Android, IOS, etc.) [23].

În acest capitol vor fi prezentate particularitățile atacului, modul în care a fost mititgat și modul în care poate fi reprodus.

3.1 Descrierea Atacului

Meltdown se bazează pe executarea intenționată a unor instrucțiuni care produc excepții 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* și a instrucțiunilor tranzitorii, datele respective sunt folosite pentru a produce o schimbare masurabilă la nivel arhitectural și anume, în cache-ul procesorului. Prin tratarea excepțiile ridicate de sistemul de operare în urma instrucțiunilor ilegale și folosind unul dinte atacurile documentate asupra memoriei cache, precum *FLUSH and RELOAD*, creăm un canal secundar de comunicare prin care atacatorul capătă acces la date în mod neprivilegiat.

3.1.1 Structura Memoriei Virtuale în Cadrul unui Proces

Pentru înțelegerea atacului este important de înțeles cum arată spațiul memoriei virtuale la nivelul fiecărui proces. Kernelul are nevoie de acces la toată memoria. Astfel

toată memoria fizică este mapata în kernel la o anumită adresă. De asemenea, spațiul de memorie alocat unui proces este împărțit între zona utilizatorului și zona de kernel. Kernelul administrează tabelele de traducere a adreselor de memorie și veghează ca accesul utilizatorului să fie restricționat doar la spațiile de memorie în care se află datale folosite de programul curent. Astfel, în ciuda faptului că în cadrul procesului există alocate adrese virtuale de memorie care corespund adresei fizice a kernelului, acestea nu pot fi accesate fără privilegii speciale (determinate de *user/supervisor bit* vezi 2.1.6 și figura 3.1) în cadrul rulării unui program în parametrii normali [21].

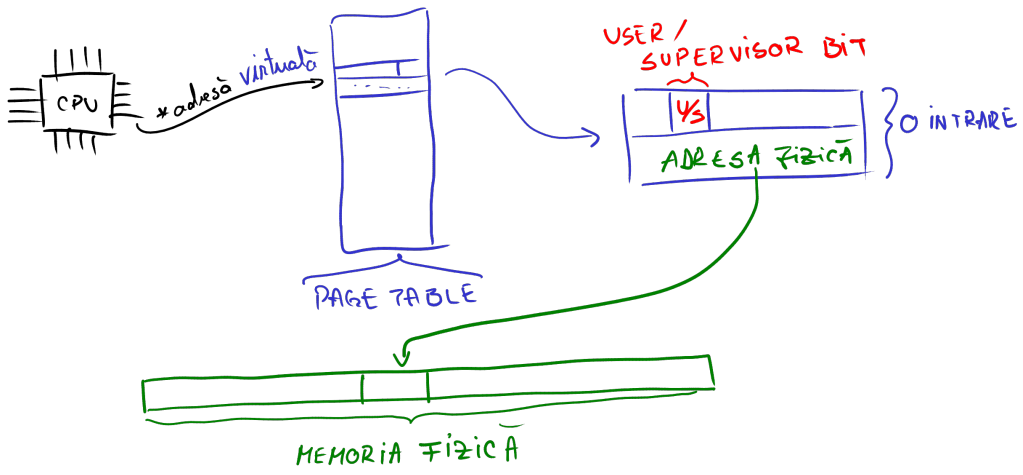

Figura 3.1: Traducerea adresei virtuale în adresa fizică

La momentul descoperirii atacului, întreaga memorie fizică putea fi accesată prin intermediul kernel-ului, fiind mapata direct în cadrul acestuia. Meltdown reușește să ignore regulile stricte de separare dintre user și kernel și prin intermediul acestei legături directe între kernel și restul memoriei poate citi date fară drepturi asupra acestora. Mitigarea vulnerabilității a constat în consolidarea mecanismelor de separarea dintre utilizatorul neprivilegiat și kernel.

3.1.2 Exectuarea Instrucțiunilor Tranzitorii

Scopul atacului este de a accesa și citi date asupra cărora utilizatorul nu are drepturi, așadar în cadrul atacului vom ținti acest tip de zone de memorie. După cum am menționat în secțiunea 2.2.2, procesoarele moderne execută adesea instrucțiunile într-o ordine diferită de cea în care apar acestea în program. Există posibilitatea ca astfel de instrucțiuni să ruleze înainte ca verificările asupra drepturilor de executare asupra acelei instrucțiuni să fie finalizate. Să considerăm următoarea secvență de cod:

```
1   // declararea în prealabil a variabilelor
2   int x;
3   char kernel_data;
```

```
4
5    // accesare kernel produce exceptie
6    kernel_data = *kernel_data_addr;
7    // instructiunile care urmeaza nu sunt niciodata executate la nivel
     macro
8
9    // accesarea elementului din tabloul  duce la
10   // încarcarea în 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];
```

Listing 3.1: Executarea instrucțiunilor tranzitorii

Se presupune că avem la cunoștință adresa unui secret stocat în zona de kernel stocată în variabila `kernel_data_addr`. Execuția instrucțiunilor din linia 6 va duce la dereferentia adresei respective, ceea ce va produce un *Segmentation Fault*. Astfel, la nivel macro, prin intervenția sistemului de operare, datele de la adresa respectivă nu vor fi accesate, conform restricțiilor impuse de kernel asupra utilizatorului neprivilegiat. Comportamentul astfel rezultat al programului este cel așteptat.

La nivel microarhitectural fluxul de execuție diferă. Verificarea drepturilor de acces asupra unei zone de memorie au loc după accesarea acesteia. Din cauza *Out-of-Order Execution* și rulării relativ lentă a verificării drepturilor de accesare, instrucțiunile care succed vor fi executate speculativ. La nivel microarhitectural datele de la adresa `kernel_data_addr` vor fi preluate. Și încărcate în variabila `kernel_data`. Ulterior, tabeloul `probe` vă fi accesat la index-ul corespunzător `kernel_data`, iar pagina corespunzătoare acelei valori vă fi încărcată în cache (linia 12). Rezultatele obținute vor fi evident omise când dreptului de acces asupra adresei respective este invalidat, iar starea registrilor vă trebui resetata la cea anterioara liniei 6.

Din cauza unui bug în arhitectura majorității procesoarelor, datele încărcate în cache-ul procesorului în cadrul executării speculative, rămân în cache chiar și după resetarea stării microarhitecturale. Aceste *Instrucțiuni Tranzitorii* prin faptul că nu resetează starea cache-ului permit folosirea tehnicilor specifice atacurilor cache pentru recuperarea secretului [23].

Observație - Principiul Localității

Trebuie făcută o observație importantă pentru linia 12. În cadrul instrucțiunii tranzitorii pe care vrem să o executăm am accesat tabloul `probe` la index-ul *kernal_data* × 4096, în loc de *kernel_data*. Conform *principiului localității* [9], când are loc un cache miss procesorul nu încarcă doar 1 byte din memorie, ci multiplii bytes în funcție de dimensiunea liniilor din cache (de obicei 64 de bytes). Rezultatul este că în momentul în care un element `probe[k]` ar fi accesat, și el și elementele adiacente acestuia ar fi

încărcate în cache. Pentru a putea distinge ușor între elemente adiacente vom folosi un multiplicator (în acest caz 4096) mai mare decât dimensiunea tipică a unei linii de cache pentru ca accesarea a două elemente adiacente să nu determine încărcarea în cache a unor blocuri cu zone comune.

3.1.3 Gestionarea Excepțiilor

Pentru a nu întrerupe fluxul de execuție și a putea interpreta datele rezultate în urma *Instrucțiunilor Tranzitorii* trebuie tratate execeptiile apărute în mod natural. Conform lucrării de cercetare [23] avem două variante: Suprimarea Excepțiilor și Tratarea explicită a Excepțiilor.

Duplicarea Procesului

O metodă trivială presupune duplicarea (*forking*) procesului chiar înainte de declanșarea excepției. Linia 6 a codului ar fi astfel executata în procesul copil, care va fi terminat de către sistemul de operarare. Efectele secundare apărute în urma execuției speculative pot fi apoi măsurate din procesul părinte prin intermediul unui canal secundar (*side-channel*), eg. memoria cache.

Signal Handler

O meteoda alternativă este instalarea unui *signal handler* care se declanșează la apariția excepției corespunzătoare (eg. *Segmentation Fault*) Astfel evităm terminarea programului de către sistemul de operare și reducând timpul de execuție necesar creării unui nou proces.

Suprimarea Excepțiilor

Putem de asemenea preveni declanșarea excepțiilor în totalitate. În cadrul execuției speculative se pot executa instrucțiuni care nu s-ar executa în mod normal din cauza unei predicții greșite a căii urmate de fluxul de instrucțiuni în urma unei bifurcări. Astfel, deoarece la nivel macro instrucțiunile ilegale nu vor fi executate, excepția nu va fi declanșată de sistemul de operare, chiar dacă la nivel microarhitectural, liniile de cod au fost executate speculativ, iar efectul secundar a avut loc. Această abordare implică antrenarea oracolului de prezicere a bifurcărilor (*branch predictor*) și va fi descris în cadrul discuției despre *Atacuri Spectre* [22].

3.1.4 Trecerea din planul micro în planul macro

Ca urmare a gestionării excepțiilor, execuția programului continua neîntreruptă. În urma executării instrucțiunilor tranzitorii, la nivel microarhitectural starea sistemului

s-a schimbat, prin încărcarea în cache a adresei accesate speculativ. Prin intermediul tehnicilor specifice atacurilor asupra memoriei cache (descrise în secțiunea 2.3.1) putem crea un canal secret de comunicare (covert channel). Va fi prezentată inițial metoda de transmitere a unui bit pe canalul secret, iar ulterior această metodă va fi extinsă la transmiterea unui byte.

Pentru a transmite un bit se procedează în felul următor. Pentru transmiterea de informație se executa o serie de instrucțiuni tranzitorii în urma cărora zona de memorie dorita este încărcată în cache. Pentru citirea informației de pe canalul ascuns este folosită tehnica *FLUSH and RELOAD* (2.3.1). Inițial se folosește instrucțiunea `clflush` pentru eliberarea din cache (eviction) a adresei tintă, iar apoi se așteaptă transmiterea unui bit de informație. După trecerea perioadei de timp se măsoară timpul necesare accesării valorii de la adresa tintă. Dacă timpul de accesare este mic (sub un prag stabilit anterior) îl clasificam ca *cache hit*. În acest caz, constatăm că victima a accesat în fereastra de timp adresa de memorie prin intermediul unei instrucțiuni tranzitorii, și a transmis atfel un bit cu valoarea emph1. Dacă timpul de accesare este mare (peste pragul stabilit anterior), clasificăm accesarea că un *cache miss*. În acest caz, victima nu a executat instrucțiunile tranzitorii corespunzătoare încărcării în cache a valorii țintă, deci a transmis implicit un bit cu valoare *0*.

Pentru transmiterea a 1 byte de date (dimensiunea corespunzătoare unui caracter în format ASCII) se va proceda în felul următor. Conform secțiunii de cod din 3.1.2, pentru fiecare dintre cele 256 de valori posibilie pe care le poate avea un byte de informație, se accesează o linie separată în cache în mod speculativ prin intermediul unui set de instrucțiuni tranzitorii. Pentru reconstruirea informației, atacatorul vă efectua atacul *FLUSH and RELOAD* pentru fiecare dintre cele 256 de zone posibile din cache. Inițial, fiecare dintre cele *256* de adrese trebuie eliminate din cache (se folosește instrucțiunea `clflush`). În final, index-ul pentru care s-a obținut un *cache hit* va corespunde valorii byte-ului de informație transmis prin canalulu secret (vezi figura 3.2). În particular, pentru array-ul `probe` se vă rula *FLUSH and RELOAD* pentru fiecare dintre `probe[i * 4096]` (cu $0 \leq i \leq 255$). Dacă se obține un *cache hit* pentru `probe[k * 4096]` (cu $0 \leq i \leq 255$), atunci pe canal s-a transmis valoarea k [23].

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

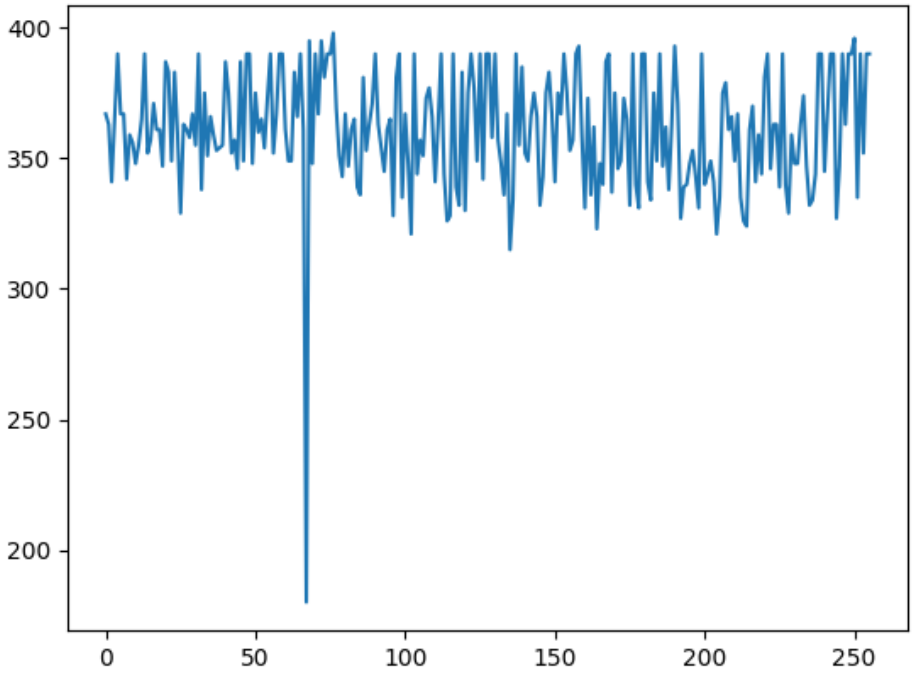

Figura 3.2: Recuperarea unui byte prin Flush & Reload

3.1.5 Scenariul și realizarea atacului Meltdown

La momentul apariției, atacul Meltdown avea ca tintă orice fel de computer personal, ori mașină virtuală în cloud. Se presupune că atacatorul nu dispune de acces fizic asupra mașinilor atacate, dar poate executa orice fel de cod în mod neprivilegiat, cu aceleași drepturi ca un utilzator obișnuit (fără drepturi de root, ori administrator). Sistemul tintă este protejat de mecanisme considerate *state-of-the-art* la vremea respectivă (i.e. nu luăm în considerare mecanismele de protecție apărute ulterior care au ca rezultat mitigarea atacului, acestea fiind discutate în secțiunea 3.4), precum *ASLR* și *KASLR*. Sistemul dispune de asemenea de un procesor care suportă *Out-of-Order Execution*. Atacul nu se bazează pe niciun fel de vulnerabilitate de tip software, exploatând doar hibe la nivel hardware. Astfel, se presupune rularea unui sistem de operare fără probleme cunoscute care pot fi abuzate pentru elevarea nivelului de privilegii. Ținta atacatorului va fi orice tip de informație de valoare precum chei de acces, parole, hash-uri, date personale, etc.

Atacul presupune obținerea adresei din kernel a secretului prin diferite mecanisme (care nu reprezintă scopul acestei lucrări), ori ghicirea acesteia. Ulterior se execută pașii descriși mai sus. În primă fază va trebui să se execute un set de instrucțiuni alese special, care folosesc adresa tintă și se execută în mod speculativ, devenind ulterior tranzitorii în urma declanșării unei excepții. Între decodarea adresei virtuale în adresă fizică, accesarea valorii corespunzătoarea și încărcarea în cache a liniei aferente *(1)*, și

verificarea drepturilor de acces asupra zonei de memorie conform drepturilor de utilizator și restricțiilor din tabela de traducere *(2)*, se produce un *race condition*. În cazul în care *(1)* se execută mai rapid decât *(2)* starea microarhitecturala va fi modificată în modul dorit, iar abia apoi sistemul de operare va constata accesul nepermis și va declanșa o excepție. Ajunși în acest punct, rolul emițătorului este finalizat. Pentru receptarea mesajului se gestionează excepția de tip *Segmentation Fault* declanșată de sistemul de operare (printr-una din metodele descrise anterior), iar apoi se folosește *FLUSH and RELOAD* pentru a recupera secretul (3.1.4).

Acești trei pași determină obținerea unui byte din secret. Pentru obținerea întregului secret se iterează prin toate adresele de memorie ale căror valoari sunt de interes. În mod similar se poate descărca întreaga memorie fizică a sistemului tintă.

3.2 Sisteme evaluate

Conform particularităților atacului, orice sistem al cărui hardware este vulnerabil va fi vulnerabil indiferent de software-ul care rulează (considerând că patch-ul software nu este aplicat). Într-adevăr s-a constatat că atacul poate dezvălui informații secrete ale unui utlizator neprivilegiat pe multiple sisteme utilizate la scară largă.

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 mapând adresele kernelului în spațiul virtual al oricărui proces neprivilegiat, bineînțeles cu restricțiile de acces aferente, acestea fiind implementate corect în tabelele de traducere a adreselor [23].

3.2.2 Microsoft Windows

Meltdown a putut fi rulat pe un sistem cu Windows 10, cu actualizările la zi, chiar înainte de aplicarea patch-urilor. În ciuda modului diferit în care este mapată memoria fizică și management-ului diferit al acesteia, majoritatea memoriei fizice tot se poate citi prin intermediul atacului. Cercetătorii au putut citit întregul binar al kernelul-ului de Windows [23].

3.2.3 Android

Deoarece Android-ul are la bază to Kernel-ul de Linux, succesul atacului pe această platforma depinde de procesorul instalat pe dispozitiv. Astfel în urma testelor asupra unui Samsung Galaxy S7 rulând un Linux Kernel cu versiunea 3.18.14, atacul nu a avut

succes pe asupra procesorului ARM Cortex A53, dar a funcționat asupra celor marca Exynos dezvoltate de Samsung.

3.2.4 Containere

Tehnologiile de containerizare care împart între ele același kernel sunt și ele vulnerabile. Meltdown poate fi rulat cu succes în containere Docker, LXC, sau OpenVZ și s-a demonstrat că prin intermediul lui se pot extrage informații nu numai din kernel, dar și din celelalte containere care rulează pe aceeași mașină fizică. Rezultatul vine din cauză că fiecare container împarte kernel-ul cu toate celelalte, așadar orice proces va avea încărcat în spațiul său virtual de memorie întreagă memorie fizică a sistemului, prin intermediul kernel-ului împărțit cu celelalte containere și procese care rulează pe același sistem.

3.2.5 ARM și AMD

În ciuda succesului pe arhitectura *Intel* și pe câteva modele *Exynos*, atacul nu a putut fi executat cu succes pe procesoare *AMD*. *AMD* susțin că acest rezultat se datorează unor diferențe arhitecurale în comparație cu *Intel* [5]. În cazul *ARM*, signrul procesor afectat ar fi modelul *Cortex-A75*, conform [12]. Deoarece detaliile de implementare ale microarhitecturilor nu sunt în general dezvăluite publicului larg nu se poate determină cu exactitate ce a determinat aceste diferențe.

3.3 Performanța

Performanța atacului depinde de modul și eficiența cu care se câștigă *race condition*-ul. Indiferent de locația din memorie în care se află datele dorite, s-a demonstrat că *race condition*-ul poate fi câștigat, dar diferențele de performanță sunt notabile.

3.3.1 Secretul în cache

Când datele sunt în imediata apropiere a procesorului (în cache-ul *L1*), *race condition*-ul poate fi castigat cu ușurință. Astfel s-au obținut performanțe ridicate de până la 582 KB/s cu rata de eroare de până la 0.003% (Intel Core i7-8700K). O variantă mai lentă a atacului care reduce eroarea la 0 ajunge la viteze de 137 KB/s [23].

În cazul în care secretul se află în cache-ul *L3*, dar nu în cache-ul *L1*, *race condition*-ul încă se poate căștiga des, dar viteza vă fi mult mai mică, de până la 12.4 KB/s, cu erori de până la 0.02% [23].

3.3.2 Secretul în afara cache-ului

În acest caz câștigarea *race condition*-ului este mai dificilă. S-au observat rate de transimitere a datelor de sub 10 B/s pe majoritatea sistemelor. Aceste rezultate au putut fi îmbunătățite cu ajutorul a două optimizări. Acestea presupun preîncărcarea zonelor de memorie din cache prin intermediul unor thread-uri care rulează în paralel (conform [14]) și abuzând de implemntarea cache-ului conform principiului localitaii (vezi 3.1.2) prin accesarea speculativă a zonelor adiacente țintei, astfel încărcând în cache și adresa tintă. Optimizările cresc viteza de citire până la 3.2 KB/s [23].

3.4 Metode de mitigare

3.4.1 Hardware

Meltdown nu exploatează niciun defect de natură software, ci ocolește restricțiile de acces prezente la nivel hardware prin intermediul execuției instrucțiunilor *out-of-order*.

Considerând mecanismele exploatate în cadrul acestui atac, două contramăsuri posibile ar fi eliminarea completă a *out-of-order execution*, sau serializarea instrucțiunilor care verifică permisiunile de acces și accesarea propriu-zisă a zonei de memorie, pentru a evita executarea acestora în paralel. Aceaste soluții nu ar fi fezabile întrucât impactul pe care l-ar aduce asupra performanței este mult prea mare.

O soluție mai realistă ar fi separarea la nivel hardware a zonei utilizatorului și a zonei Kernel printr-un *bit* suplimentar care marchează activarea acestui sistem. În cazul în care ar fi activat se impune ca adresele kernel-ului să se regăsească în jumătatea superioară a memoriei, iar adresele utilizatorului în jumătatea inferioară. Acest mecanism ar avea un impact neglijabil asupra performanței, deoarece nivelul de permisiuni poate fi dedus direct în adresa virtuală, astfel dreptul de acces putând fi confirmat sau infirmat direct, fară accesări suplimentare ale tabelei de traducere.

3.4.2 Software – KAISER

Deoarece rezolvarea problemelor prezente în hardware-ul aflat în prezent în folosință este imposibil de implementat în practică, a trebuit găsită o soluție de natură software. Meltdown poate avea loc deoarece adresele kernel sunt mapate în spațiul procesului chiar dacă acestea rulează neprivilegiat. Adresele virtuale mapate sunt legitime și corespund unor adrese fizice. Singurul mecanism care limitează accesul unui utilizator oarecare în spațiul kernel-ului constă în restricțiile prezente în tabelele de traducere a adreselor fiecărui proces. Acest mecanism funcționează la nivel arhitectural, rezultatul final al execuției fiind cel așteptat. La nivel microarhitectural, s-a demonstrat că prin intermediul execuției instrucțiunilor într-o altă ordine, în mod speculativ, se pot accesa din spațiul user-ului

zone de memorie din kernel înainte ca dreptul de acces să fie confirmat. Soluția naturală propusă a fost utilizarea a două spații de memorie separate pentru fiecare proces: unul dedicat utilizatorului în care kernel-ul nu este mapat, și unul dedicat kernel-ului în care spațiul utilizatorului nu este mapat. Din diverse motive de performanță și practicalitate, printre care faptul că nemaparea spațiului utilizatorului în Kernel ar presupune rescrierea unor porțiuni considerabile din Kernel, s-a ajuns la implementarea pe majoritatea sistemelor a unei soluții bazate pe KAISER ([13]) (în Kernel-ul de Linux aceasta poartă numele de KPTI sau PTI – *Kernel Page-Table Isolation*).

KAISER a fost propus ca soluție pentru atacurile asupra *Kernel Address Space Layout Randomization* (KASLR) și presupune o metodă de separare a spațiului kernel de spațiul utilizatorului. Acest mecanism de protecție are la bază conceptul de spațiu de adrese fantomă (*shadow address space*). Astfel, în mod neprivilegiat adresele kernel nu sunt mapate, iar în mod privilegiat adresele utilizatorului sunt mapate și vegheate de mecanismele de securitate *SMEP* și *SMAP* care previn execuția codului unui utilizator în spațiul kernel, sau posibile corupții ale memoriei. Schimbarea între cele două contexte se face foarte ușor, prin aplicarea unei măști pe biți asupra registrului `CR3`, în care se reține adresa tabelei de traducere corespunzătoare contextului curent. Implementarea acestei soluții previne atacul Meltdown, întrucât în modul de rulare neprivilegiat nu există adrese virtuale care să corespunda unor zone sensibile din kernel. S-a demonstrat că aceasta soluție are costuri de performanță neglijabile, în medie de 0.28% [13].

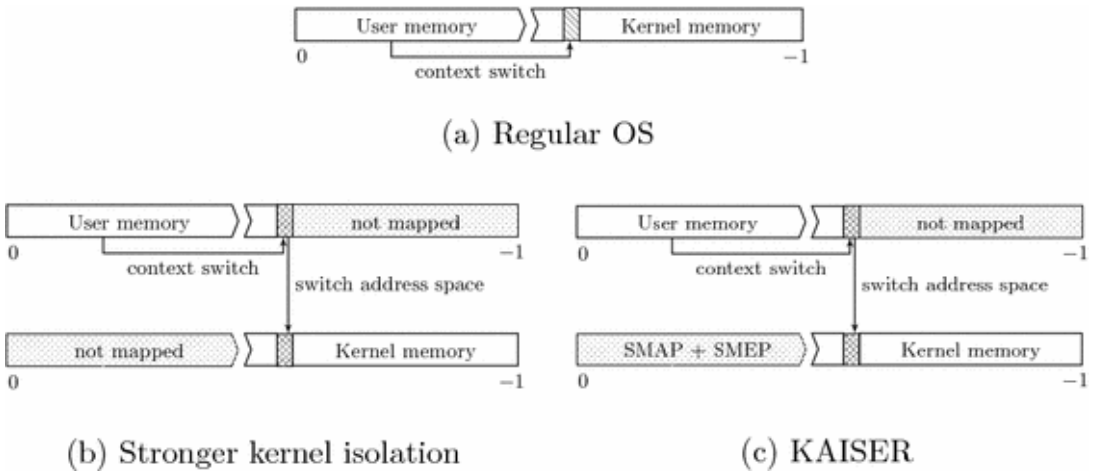

Figura 3.3: Cum se comportă memoria virtuală la schimbarea de context înainte și după implementarea KAISER [13]

3.5 Reproducerea Atacului

3.5.1 Platforma

Atacul se poate reproduce cu succes folosind mașina virtuală și ghidul disponibile pe platforma SEEDLabs [10]. Mediul virtualizat este important întrucât oferă flexibilitatea rulării sistemului cu un Kernel învechit care este încă vulnerabil la atacul *Meltdown*. În acest sens se folosește o mașină virtuală care rulează *Ubuntu 16.04* cu versiunea 4.8.0 cu KPTI neimplementat a Kernel-ului Linux. De asemenea pentru succesul experimentului este necesară rularea pe o mașină ce are instalat un procesor marca *Intel* mai vechi de generația a 9-a, deoarece începând cu arhitectura *Ice Lake* Intel a iceput să introducă patch-uri la nivel de hardware (*in-silicon*) pentru Meltdown și Spectre.

3.5.2 Aspecte importante

În scopul reproducerii atacului se va crea un scenariu fictiv ce îndeplinește multiple condiții favorabile montării Meltdown, în scopul evidențierii efectelor pe care le poate avea acesta asupra unui sistem. Cunoștințele ilustrate astfel nu vor servi decât ca o bază a înțelegerii, și nu ca o unealtă împotriva unui sistem real.

Secretul

Se creează un modul de kernel în interiorul căruia se reține un mesaj secret cu textul *"Cheia secretă din spațiul kernel"* și adresa acestuia se afișează în buffer-ul de mesaje dedicat Kernel-ului. Modulul de kernel se compilează și ulterior se instalează pe sistem, după care se reține adresa secretului pentru a direcționă atacul exact asupra țintei. În secvența 3.2 se observă declararea și afișarea în buffer-ul de kernel a adresei mesajului secret. În secvența 3.3, se observă instalarea și recuperarea adresei secretului dorit.

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

Listing 3.2: Declararea secretului și afișarea 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"

Optimizări

Pentru a creste șansele câștigării *race-condition-ului* se iau în calcul toate cele ce urmează:

1. la fiecare inițiere a atacului se încarcă secretul în cache

2. succesul atacului depinde de o sincronizare fină la nivel microarhitectural, așadar succesul în urma unei singure rulări nu este garantat. În practică, pentru aflarea unui byte, atacul asupra aceleiași zone de memorie se repetă de multiple ori, iar la urmă se folosesc niște tehnici statistice pentru identificarea valorii celei mai probabile pentru adresa țintă. Am obținut rezultate bune cu repetări între 50 și 1000. Un număr mai mic de repetări rezultă în viteze mai mari de citire, dar și o rată a erorii mai mare. Un număr mai mare de repetări rezultă în viteze mai mici de citire, dar și precizie mai mare.

3. pragul în funcție de care diferențiem *cache-hituri* de *cahe-miss*-uri trebuie calibrat. Mai multe detalii în secțiunea 5.3.1.

4. chiar înainte de execuția instrucțiunilor tranzitorii poate fi benefic să *"ținem unitățile de calcul ocupate"* prin executarea unor instrucțiuni goale. Am evidențiat aceasta idee și în implementarea atacului *Spectre-v1* 3.

5. declararea statică a unor variabile (keyword `static`), sau accesarea lor în mod repetat poate preveni compilatorul din a optimiza fragmentul de cod din care fac parte, ceea ce ar putea determina eșecul experimentului. Experimente personale au arătat că acest fapt este foarte important în cazul vairabilei `junk` utilizata la măsurarea timpilor de acces în cadrul *FLUSH and RELOAD* 3.

3.5.3 Starea actuală - Testarea efectului KPTI

Sistemul meu principal rulează ultima versiune disponibilă a kernel-ului de linux în data de 25.05.2022, `5.17.9-arch1-1`. Am verificat prezența *KPTI* pe acest sistem cu ajutorul următoarei 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 și Verifcare prezență KPTI

Într-adevăr, încercarea de a rula aceeași imlementare a atacului Meltdown pe acest sistem protejat nu duce la niciun rezultat.

Capitolul 4

Atacuri Spectre

Spectre [22] este o clasă de atacuri asemănătoare cu *Meltdown*, care exploatează efectele secundare ale execuției speculative pentru a extrage informații în mod malițios din spațiul de memorie al unei victime. La nivel înalt, se bazează pe găsirea sau introducerea unei secvențe de instrucțiuni în spațiul de adrese al procesului victimă. Mai apoi, execuția acestei secvențe va crea un canal de comunicare ascuns prin care sunt transmise date din spațiul de memorie al victimei către atacator. Similar cu *Meltdown* [23], atacul nu se bazează pe niciun fel de vulnerabilitate software, ci abuzează vulnerabilități microarhitecturale la nivel hardware. Chiar dacă în urma executării speculative a unei secvențe de instrucțiuni, se revine la o stare anterioară, schimbările apărute pe parcus la nivel de cache pot persista, aceasta fiind vulnerabilitatea principală.

În acest capitol se vor prezenta cele două variante principale din clasa de atacuri Spectre. În ultimul capitol va fi descrisă o implementare demonstrativă a variantei 1, care exploatează o zona de memorie partajată cu procesul victimă.

4.1 Diferențe față de Meltdown

Prima diferență față de *Meltdown* este că variantele *Spectre* evită provocarea unei excepții prin accesul ilegal al unei zone de memorie. În schimb, se bazează ori pe antrenarea *branch-predictor-ului* ori pe injectarea unor adrese alese special în *branch-target-buffer* cu scopul de a accesa zona de memorie de interes, doar în mod speculativ. Astfel, nu este necesară gestinarea exeptiilor, atacul interferând minimal, chiar insesizabil, cu executarea normală a programului.

A doua diferență constă în faptul că *Meltdown* exploatează o vulnerabilitate specifică procesoarelor *Intel* și câtorva procesoare *ARM* prin intermediul cărora instrucțiuni executate speculativ pot ignora restricțiile impuse de bitul *user/supervisor* prezent în intrările din tabelele de traducere ale adreselor virtuale. Astfel, *Meltdown* poate accesa memeora Kernel și implicit poate citi toată memoria fizică a sistemului prin intermediul unui canal

ascuns. *Spectre*, în schimb extrage prin intermediul unei zone de memorie partajată și a unui canal ascuns implementat în zona respectivă de memorie, doar informații la care victima țintă are acces. Se violează astfel izolarea inter-proces, dar nu se poate accesa direct zona de Kernel.

Spre deosebire de *Meltdown*, atacurile *Spectre* afectează o plajă mult mai largă de arhitecturi și s-au dovedit a fi mai dificile de mitigat. Mai mult, mecanismul care stă la baza mitigarii *Meltdown* (KAISER [13]) nu protejează în niciun fel împotrivă variantelor *Spectre*.

4.2 Spectre V1

Varianta întâi de *Spectre* presupune manipularea *branch-predictor-ului* 2.2.3 în prezicerea eronată a ramurii de execuție în cadrul unei structuri decizionale. Astfel, prin intermediul execuției speculative un atacator poate citi zone arbitrare de memorie din afara contextului său de execuție, ceea ce violează principiile de izolare impuse de memoria virtuală și sistemul de operare.

4.2.1 Descrierea Atacului

O secvență de cod precum 4.1 se poate regăsi în cadrul unui apel de funcție (e.g. funcție de sistem sau parte dintr-o librărie), care primește ca argument o variabilă dintr-o sursă oarecare, neverificată (e.g. în urma unui apel către un API). Să considerăm situația în care procesul care rulează codul respectiv (i.e. victima) are acces la un tabloul cu elemente de tip byte `array` de dimensiune `array_size` și la tabloul cu același tip de elemente, `probe` de dimensiune $256 * 4096 = 1MB$. Verificarea de la începutul secvenței de cod are rol în securizarea programului. În eventualitatea rulării secvenței cu o valoare în `x` mai mare sau egală cu `array_size` să se evite declanșarea unei excepții (e.g. *Segmentation Fault*), sau a accesării unei alte zone de memorie din cadrul de execuție a procesului victimă (e.g. dacă valoarea din x reprezintă diferența dintre adresa de început a tabloului `array` și adresa de început 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 speculativă în structura decizională

Se consideră cazul în care `array_size` nu este încărcat în cache. Determinarea ramurii care va fi urmată de fluxul de execuție va fi semnificativ întârziată din cauza cererii din *DRAM* a valorii variabilei respective, care precede evaluarea condiției logice. Pentru

a nu stagna execuția și a ține unități de lucru din *CPU* inactive se realizează o presupunere educată ce vizează ramura pe care urmează să continue execuția, prin intermediul *BPU* (vezi secțiunea 2.2.3 și figura 4.1). Urmând prezicerea se pot executa speculativ instrucțiunile ce urmează.

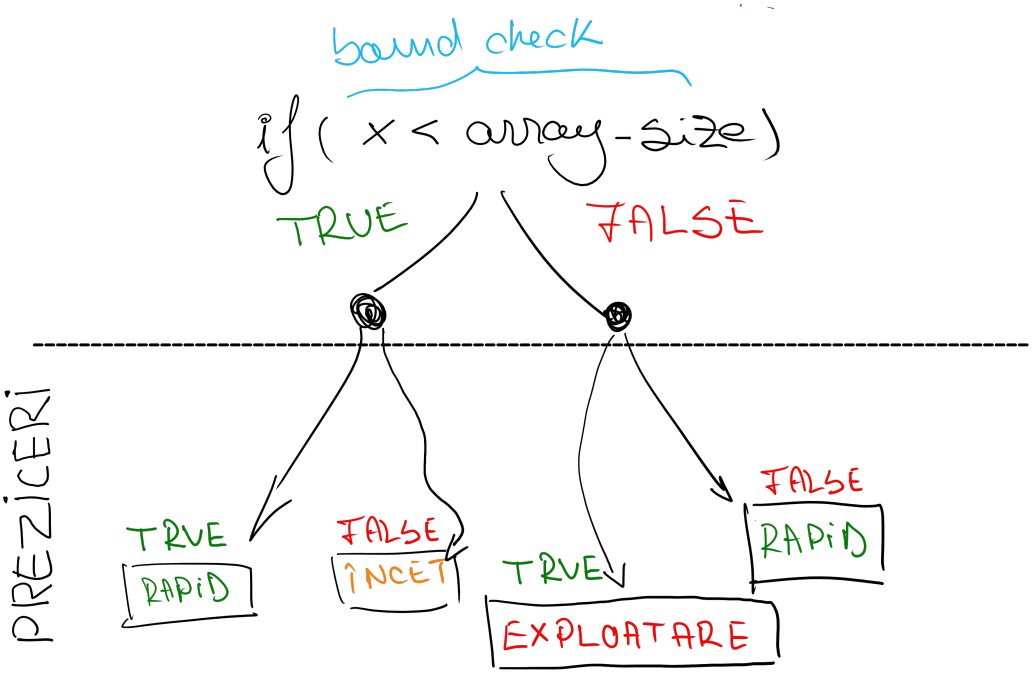

Figura 4.1: Cele patru cazuri posibile în cadrul prezicerii unei ramuri.

Acest tipar poate fi exploatat după cum urmează. Un atacator poate executa intenționat în mod repetat zona respectivă de cod cu valori mai mici decât `array_size`. Astfel, la următoarea execuție a zonei respective, *BPU* va prezice că fluxul de execuție va urma prima ramură a structurii decizionale (corespunzătoare evaluării condiției logice la *True*). La următoarea executarea a zonei respective atacatorul va transmite un `x` cu o valoare malițioasă aleasă special în afara limitelor impuse de `array_size` ca în urma evaluării `array[x]` să întoarcă o valoare `k` dintr-o alta zonă din spațiul de memorie al victimei. Fereastra de timp poate fi suficient de mare ca *linia 4* din 4.1 să fie executată speculativ. Astfel, va ajunge în cache o linie din memorie corelată cu byte-ul `k` ce face parte din secretul accesat speculativ. După ce valoarea din `array_size` este primită din *DRAM*, iar condiția logică este evaluată la *False*, starea CPU-ului se întoarce la cea dinaintea executării speculative. *Spectre*, exploatează faptul că în urma executării speculative eronate, la nivel microarhitectural, starea cache-ului din *CPU* nu este resetată.

Pentru finalizarea atacului și recuperarea datelor transmise pe canalul ascuns se procedează ca în atacul *Meltdown* (vezi 3.1.4). Se utlizeaza tehnici precum *FLUSH and RELOAD*, sau *EVICT + TIME* pentru a măsura timpul de acces în fiecare frame de

$4KB$ din tabloul `probe`. În cazul în care timpul de acces pentru variabila `k` este semnificativ mai scăzut comparativ cu al celorlalte adrese, concluzionăm că byte-ul de date tansmis corespunde cu valoarea `k` [22].

4.2.2 Reproducerea atacului

În practică, urmând strict detaliile descrise anterior, vom obține un nivel scăzut de acuratețe. Deoarece comportamentul canalului de comunicare ascuns nu este constant, ci fluctuează și executarea cu succes în mod speculativ a instrucțiunilor dorite nu este garantată, este necesară repetarea pașilor de multiple ori pentru fiecare byte, iar apoi determinarea statistică a valorii celei mai probabile.

S-a confirmat că această variantă a *Spectre* afectează arhitecturile *Intel*, *AMD Ryzen* cât și implementări *ARM* care suportă execuția speculativă. O variantă neoptimizata a atacului implemenatata în limbajul `C` care este prezentată în lucrarea de cercetare [22] atinge viteze de 10 KB/s pe un sistem cu un procesor *Intel i7-4650U*.

Javascript

Atacul s-a dovedit practic și în contextul browserelor web. S-a reușit implementarea acestei variante a *Spectre* în JavaScript și citirea unor informații private în mod neprivilegiat din spațiul de memorie al procesului în cadrul căruia rulează codul. Deoarece instrucțiunea `clflush` nu este accesibilă prin intermediul limbajului acesta, se folosește o tehnică alternativă precum *EVICT and RELAOD*. De asemenea instrucțiunea `rdtscp` nu este disponibilă, iar motorul din Chrome nu oferă un ceas de mare precizie pentru a preveni *timing attacks*. Pentru a depăși aceste obstacole se poate construi un ceas cu precizie suficient de mare prin incrementarea într-un thread separat a unei zone de memorie controlată de atacator [22].

Experimente personale

Dezvoltând ideile anterioare am realizat o implementare personală care ilustrează un scenariu mai realist în care atacatorul rulează un proces separat de victimă și împarte cu aceasta doar o zonă de 1MB de memorie partajată. Implementarea cât și detaliile se găsesc în capitolul 5.

4.3 Spectre V2

Varianta a doua a clasei de atacuri *Spectre* presupune otrăvirea (*poisoning*) salturilor indirecte (*indirect branches*) (eg. o instrucțiune de tip `jump` la o adresă reținută într-un

registru). Un atacator poate astfel determina execuția speculativă a unor instrucțiuni alese special, prin tehnici care amintesc de ROP (2.3.4), pentru accesarea și apoi extragerea prin intermediul unui canal secret ale unor date accesibile victimei.

4.3.1 Descrierea atacului

În momentul unui salt indirect, *branch-predictor-ul* va încerca să prezică adresa la care urmează să se continue execuția pentru a executa speculativ în avans instrucțiunile ce urmează. În vasta majoritate a cazurilor se obține o îmbunatățire a timpului de execuție a programului ca urmare a acestei tehinici, deoarece timpul de recuperare a adresei de destinație poate fi considerabil în cazul în care adresa de memorie nu este în cache-ul din *CPU*. Prezicerea se bazează pe istoricul adreselor accesate în trecut în urma aceluiași salt indirect. Aceste date sunt reținute într-o structură numita *branch target buffet* și sunt codificate, în funcție de arhitectură, în funcție de ultimii `k` biți ai adresei corespunzătoare.

Antrenarea

Atacul începe din contextul atacatorului. Acesta mimează salturile indirecte executate în contextul victimei. Astfel se poate plasa un salt indirect în contextul atacatorului la aceeași adresă virtuala la care se regăsește saltul în contextul victimei, sau la o adresă care are în comun ultimii `k` biți cu adresa corespunzătoare a victimei. Prin repetarea saltului la o adresă aleasă de atacator aceasta va ajunge în *branch targe buffer*, iar *branch-predictor-ul* va fi antrenat să prezică că fluxul de execuție va continua la adresa respectivă. Acest comportament depinde doar de adresa virtuală și nu ia în considerarea *PID-ul*, adresa fizică, etc. Astfel, după antrenare, dacă în contextul victimei se execută saltul aferent, se vor executa speculativ instrucțiuni începând cu adresa injectată de atacator.

S-a observat că atacatorul poate injecta chiar și adrese de memorie la care nu are acces. Astfel, în contextul său pentru antrenare poate accesa direct adrese ilegale, iar apoi să gestioneze eventualele excepții declanșate. În acest fel, atacatorul poate alege orice adresă validă în contextul victimei, chiar dacă nu are un corespondent valid în contextul său de execuție.

Alegerea adresei de destinație

În urma saltului indirect victima va continua fluxul de execuție la adresa injectată de atacator. Luând inspirație din atacurile de tip *ROP* (2.3.4) putem alege o zonă de memorie în care începe un așa numit *Gadget Spectre*. Un *Gadget Spectre* este o secvență de instrucțiuni a cărei execuție va transmite informații private ale victimei, dorite de atacator, printr-un canal secret.

Presupunând că victima are acces la doi registri `R1` și `R2`, un gadget suficient ar trebuie să conțină două instrucțiuni după cum urmează. Prima instrucțiune adună (scade, XOR-

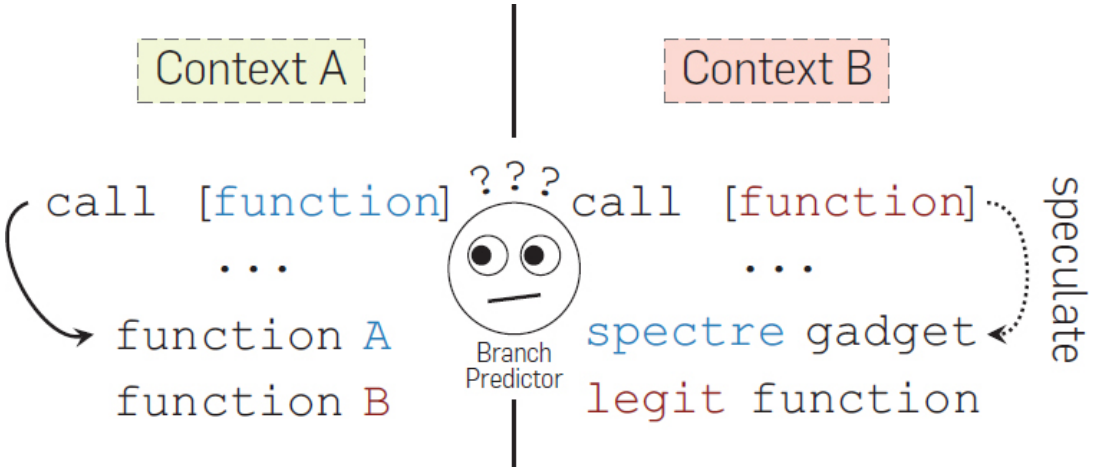

Figura 4.2: Antrenarea branch-predictorului pentru a executa cod speculativ la adrese alese special de către atacator [22].

easă, înmulțește, etc.) adresa reținută în registrul `R1` la registrul `R2`. A doua instrucțiune accesează valoarea de la adresa registrului `R2`. Un atcator va deține în acest scenariu controlul asupra adresei de memorie accesată (prin intermediul `R1`) și asupra modului în care zona de memorie dorită este asociată la o altă adresă citită prin intermediul `R2`.

Gadget-urile folosite trebuie să facă parte din zona executabilă din contextul victimei pentru a garanta execuția speculativă a acestora în *CPU*. Gadget-urile pot fi astfel alese din colecția largă de biblioteci partajate la care victima are acces, fără a fi nevoie să ne folosim de codul victimei.

4.3.2 Rezultate

În final se obține un atac asemănător conceptual cu *ROP*, dar care exploatează chiar și cod lipsit de buguri. Gadget-urile rulează doar într-o fereastră scurtă de timp și trebuie să transmită informația printr-un canal ascuns pentru ca aceasta să poată fi recuperată în urma execuției speculative. Multiple teste au demonstrat că tehnica este eficientă, reușind să injecteze adresa malițioasă în peste 98% din cazuri în cadrul a milioane de iterații [22].

4.4 Metode de Mitigare

De la momentul divulgării acestei clase de atacuri, companiile și cercetătorii au investit resurse considerabile pentru devoltarea unor metode de mitigare a vulnerabilităților prezente în sistemele afectate. În această secțiune vor fi prezentate câteva 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 [28].

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

Reprezintă o barieră care odată introdusă în cod împiedică execuția unui salt indirect din a influența salturi indirecte ce iau loc după barieră [19] [1].

Intel/AMD STIBP

Această tehnică împiedică partajarea stării din *Branch Predictor* între hiper-thread-uri ale aceluiași nucleu [19] [1].

Intel/AMD (e)IBRS

Indirect Branch Restricted Speculation are ca țintă atacurile de tip *Spectre V2*. Considerând că într-un sistem există 4 nivele de privilegii în baza cărora funcționează unitățile de prezicere, această soluție încearcă să prevină straturi care rulează cu un nivel scăzut de privilegii să influențeze straturi care funcționează la un nivel înalt de privilegii. Pe noile arhitecturi *ARM* s-au introdus soluții asemănătoare cu numele *FEAT_CSV2* [19] [1].

4.5 Starea Actuală

Principalele tehinici de mitigare folosite în practică la care recurge sistemul de operare sunt *Retpoline* și *(e)IBRS*. *Retpoline* este folosit în cazul în care mitigarile la nivel de hardware nu sunt disponibile, sau în cazul în care impactul asupra performanței este mai ridicat decât în cazul soluției software. În restul cazurilor sistemul de operare va recurge cel mai probabil la utilizarea *IBRS*, sau a soluțiilor similare pe alte arhitecturi.

În ciuda eforturilor de mitigare, un nou studiu a fost publicat [2] care demonstrează cum acestea pot fi ocolite, implementând o varianta puternică a *Spectre V2* numită *Branch history injection*. Prin intermediul acesteia se pot extrage informații în mod neprivilegiat din memoria kernel prin intermediul unui program executat în userland.

Capitolul 5

POC – Spectre-V1

În lucrarea de cercetare în care este prezentată clasa de atacuri spectre și variații ale acestora [22], este pusă la dispoziție și o implementare cu scop demonstrativ în care tehnicile descrise sunt utlizate pentru a citi conținutul unui buffer din spațiul de memorie al procesului în rulare. Victima și atacatorul sunt combinate astfel într-o singură entitate, ceea ce rezultă într-un scenariu complet nerealist. În acest capitol voi descrie o implementare personală și diferită în abordare a atacului, cu scopul de a ilustra un scenariu mai realist în care această vulnerabilitate poate fi exploatată.

5.1 Detalii de implementare

5.1.1 Scenariul

Considerăm scenariul în care pe un sistem cu actualizările la zi, cu hardware susceptibil variantei 1 a Spectre (am rulat testele pe computer-ul personal ce rulează cu un procesor marca *Intel* - generația a 8-a), rulează în mod privilegiat un proces vulnerabil (*victima*). Victima poate primi cereri de la alte procese (*clienți*), în urma cărora furnizează un rezultat prin intermediul unei zone partajate de memorie, separată complet de orice zonă privată din cadrul spațiului său de memorie. Un client comunică și se sincronizează cu victima (pentru a evita *race-condition-uri*) prin intermediul unor fișiere uzuale și a unor fișiere de blocare (*lock files*).

Voi demonstra cum, prin intermediul zonei de memorie partajată, un atacator poate obține date private din sptiul de memorie al victimei, folosind tehnici specifice variantei 1 a clasei de atacuri Spectre 4.2.

5.1.2 Vulnerabilitatea

În cadrul cererilor, atacatorul trimite victimei o mulțime de indici: $\{i_0, i_1, ..., i_n\}$. Pentru fiecare indice i_k, victima va prelua o valoare dintr-un tablou static intern, `array1[i_k]`.

Va realiza apoi o serie de clacule oarecare, pe baza unei valori din zona partajată cu clienții (`array2`), aflată la o poziție proportinala cu `array1[i_k]`.

Să considerăm secțiunea următoare de cod:

```
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 };

void victim_function(size_t x) {
  static uint8_t temp = 0;  /* declararea statica previne optimizari
   neprevazute ale compilatorului */
  if (x < array1_size) {
    // calcule cu o valoare din array2 la indice proportional cu array1[x]
    temp &= array2[array1[x] * 512];
  }
}
```

Listing 5.1: Secțiune vulnerabila din codul victimei

Funcția `victim_function` este apelată cu fiecare indice i_k primit de la atacator, transmis prin intemediul parametrului `x`. Se poate observa verificarea `x < array1_size` ce previne accesarea unor poziții în afara limitelor tabloului. Astfel, codul poate fi considerat sigur din punct de vedere software. În schimb, un atacator poate alege valorile într-un mod specific ce va determina schimbări la nivelul cache-ului, măsurabile prin intermediul zonei partajate de memorie, care pot dezvălui informații secrete din spațiul victimei.

Zona partajată

Zona partajată de memorie este creată de către victimă și încărcată de către atacator **doar cu drepturi de citire**, după cum urmează:

```
// mapare a zonei partajate în array2 (cache side channel)
int fd = open(shared_memory_name, O_RDONLY);
array2 = (uint8_t*)mmap(NULL, 256 * 512, PROT_READ, MAP_SHARED, fd, 0);
```

Listing 5.2: Maparea zonei partajate în atacator

5.1.3 Citirea unui byte

În cadrul dezvoltării de aplicații, dezvoltatorii folosesc mecanisme de sicronizare și tehnici specifice *IPC* pentru transmiterea în siguranță și fară pierderi a datelor de la client la server și invers. În cadrul acestui atac, atacatorul, reușește să citească date de la victimă cu mare precizie datorită utilizării acestor tehinici.

Pasul 1 - Preluarea lacătului

Transmiterea datelor și procesarea acestora de către victimă au loc separat. Acest fapt este garantat prin utilizarea unor emphlock files [11], care permit executarea unei secvențe de instrucțiuni numai când procesul în cauză deține lacătul.

La pasul 1, atacatorul preia lacătul în felul următor:

```
locked = -1;
while (locked != 0) {
  fd_lock = open(lock_file_name, O_CREAT);
  locked = flock(fd_lock, LOCK_EX);
}
```

Listing 5.3: Preluarea lacătului

Pasul 2 - Transmiterea informației

Atacatorul transmite un set de indici pe care procesul victimă îi va folosi pentru accesul datelor. Setul este împărțit într-un număr de 4 runde. Fiecare antrenare conține 7 indici ce determină accesări în limitele tabloului `array1` ce au ca scop antrenarea *branch-predictor-ului* în prezicerea primei ramuri și un indice ce produce o accesare **speculativă** în afara `array1` a unei zone dorite de atacator.

```
f = fopen(index_file_name, "w");
training_x = tries % array1_size;

fprintf(f, "%d ", train_rounds * round_length);
for (i = 0; i < train_rounds; ++i) {
  for (j = 0; j < round_length - 1; ++j) {
    fprintf(f, "%zu ", training_x); // indice de antrenament
  }
  fprintf(f, "%zu ", index); // indice de atac
}
fclose(f);
```

Listing 5.4: Transmiterea datelor

Alegerea indicelui de antrenament este importantă. Antrenarea în sine vă rezulta într-un *cache-hit* care corespunde valorii aferente indicelui de antrenare. Această valoare poate fi determinată în prealabil și apoi ignorată în cadrul etapei *FLUSH & RELOAD*.

Indicii sunt scriși într-un fișier ”index.txt” la care victima are acces. Absența ulterioară a acestuia va semnifica faptul că victima a terminat procesarea datelor, fiind folosit de asemenea a un strat suplimentar de sincronizare.

Pasul 3 - Flush și eliberarea lacătului

Atacatorul eliberează zona partajată de memorie din cache pentru a putea urmări schimbările determinate de acțiunile victimei. În final se eliberează lacătul pentru a permite victimei să proceseze datele transmise.

```
1  for (i = 0; i < 256; i++)
2    _mm_clflush(&array2[i * 512]);  /* instructiunea clflush */
3
4  // eliberarea lacatului
5  unlink(lock_file_name);
6  flock(fd_lock, LOCK_UN);
7  close(fd_lock);
```

Listing 5.5: Flush și eliberarea datelor

Pasul 4 - Victima preia lacătul

Asemănător cu atacatorul, pentru a începe executarea instrucțiunilor victima trebuie să preia accesul asupra lacătului partajat.

Pasul 5 - Victima procesează indicii primiți

Victima preia datele transmise de atacator și, pe rând, execută codul vulnerabil cu fiecare dintre indici. Această serie de instrucțiuni va determina introducerea în cache la un indice corespunzător cu valoarea secretă a unei valori din zona de memorie partajată `array2`.

```
1  for (int i = 0; i < no_items; ++i) {
2    _mm_clflush(&array1_size);
3    for (volatile int z = 0; z < 100; z++)
4    victim_function(buffer[i]);
5  }
6
7  // stergerea fisierului "index.txt"
8  fclose(f);
9  remove(index_file_name);
```

Listing 5.6: Executarea codului vulnerabil pentru indicii primiți

Este important de menționat că succesul atacului depinde de câțiva factori ilustrați în secvența de cod de mai sus. În primul rând, absența din cache a variabilei `array1_size` (i.e. linia 2) este importantă pentru a facilita câștigarea *race-condition-ului* la nivel microarhitectural. În al doilea rând, o încărcare ridicată a sistemului rezultă în acuratețe mai ridicată. Putem simula aceasta încărcare prin executarea unor instrucțiuni în prealabil (i.e linia 3).

În final, se șterge fișierul cu date ”index.txt” și se eliberează lacătul, pentru a semnala finalul procesării cererii.

Pasul 6 - Atacatorul finalizează FLUSH & RELOAD

Odată cu ștergerea fișierului ”index.txt” de către victimă, atacatorul poate finaliza atacul cu o tehnică specifică atacurilor asupra memoriei cache. În implementarea de față am folosit tehnica *FLUSH & RELOAD*.

```
addr = &array2[index * 512];
time1 = __rdtscp(&junk);            /* citeste valoare din cronometru */
junk = *addr;                       /* acceseaza memoria */
time2 = __rdtscp(&junk) - time1;    /* calculeaza timpul scurs */
if (time <= CACHE_HIT_THRESHOLD && index != array1[training_x])
  results[index]++;  /* cache hit - se adauga 1 la scor. */
  /* dupa mai multe repetari indicele cu scor
     maxim este considerat rezultatul dorit */
```

Se măsoară timpul de acces pentru fiecare adresă corespunzătoare fiecăruia dintre cei 256 de bytes, iar valorile diferite de cea de antrenament și ai carori timpi de acces sunt sub un prag stabilit anterior `CACHE_HIT_THRESHOLD` sunt marcați într-un tablou de rezultate.

Citirea datelor secrete

Repetarea pașilor 1-6 de multiple ori pentru aceeași zonă de memorie este necesară pentru a garanta acuratețea, iar apoi repetarea și pentru alte zone de memorie rezultă în posibilitatea citirii întregului spațiu de memorie al victimei.

5.2 Rezultate

Prin rularea procesului victimă în mod privilegiat, se pot citi și reține în spațiul acestuia date sensibile precum hash-urile parolelor de sistem prezente în fișierul `/etc/shadow`. În mod normal aceste date, chiar și încărcate sunt în siguranță datorită mecanismelor de separare a contextelor de execuție. În schimb, în acest scenariu, prin intermediul tehnicilor ilustrate, atacatorul **neprivilegiat** poate citi întreg spațiul de memorie al victimei, inclusiv conținutul fișierului `/etc/shadow`.

Pentru protecția datelor personale am ales să încarc în victimă fișierul de configurare al bootloader-ului. Rezultatele se pot observa în captura de ecran din figura 5.2.

Am reușit să obțin viteze de citire de până la 2.2 kB / secundă. Codul complet poate fi accesat la următoarele adrese: victima, atacator.

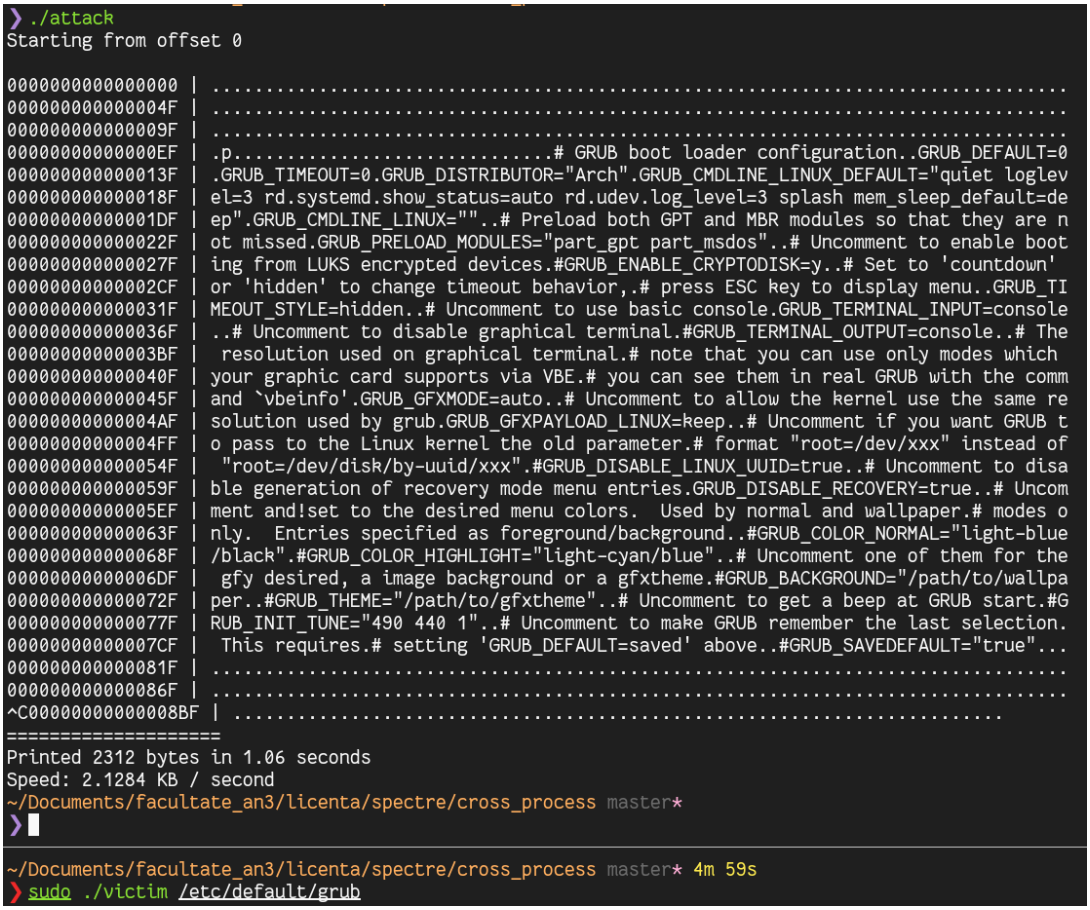

Figura 5.1: Configurația bootloader-ului extrasă cu spectre-v1

5.3 Observații interesante

Pe parcursul experimentelor am putut face multiple observații și am identificat diverse elemente care au un rol important în succesul atacului.

5.3.1 Pragul pentru determinarea cache-hit-rilor

Determinarea cache-hit-urilor în toate atacurile și experimentele descrise se face pe baza unui prag stabilit anterior. Acesta depinde de mulți factori cum ar fi hardware-ul pe care se rulează codul, și nivelul de încărcare al sistemului.

Pentru utlizarea unui prag cât mai potrivit în cadrul experimentelor am realizat o unealtă care poate determina cu precizie mare un o valoare potrivită. Ideea din spatele uneltei este rularea unuia dintre atacurile prezentate pe parcursul lucrării cu mai multe valori ale pragului și contorizarea performantelor obținute. Ulterior, se realizează o medie ponderată a valorilor obținute în funcție de performanța acestora.

Rulând această unealtă în diverse contexte am observat următorul rezultat neașteptat. Apar diferențe de până la 20% în cazul în care laptop-ul este conectat la o sursă de alimentare în comparație cu cazul în care rulează doar pe baterie. Explicația ar fi aceea că

pentru conservarea energiei când nu este alimentat extern, se reduce viteza procesorului.

5.3.2 Flag-uri de compilare și tipul variabilelor

În funcție de tipul variabilelor declarate, sau flag-urile de optimizare utilizate atacul pote rula cu succes, sau eșua total. Spre exemplu, utilizarea unui tip de date non-static pentru variabila `junk` în cadrul *FLUSH & RELOAD* (secțiunea 5.1.3 linia 3) poate rezulta în eșecul atacului. Acest eșec poate, în schimb, fi prevenit prin compilarea fie cu o versiune mai veche a compilatorului (am testat cu gcc 5.3 care probabil aborda optimizările diferit), sau cu un flag de compilare explicit (am avut succes cu `-O`, sau `-O1`).

Capitolul 6

Concluzii și Direcții Viitoare

În această lucrare am abordat subiectul *Atacurilor Speculative*. Am prezentat detalile atacului *Meltdown* care permitea unui utilizator neprivilegiat citirea oricărei zone din memoria fizică prin intermediul Kernelului, soluția care a dus la mitigarea acestuia (*KAISER*), cât și observații personale legate de reproducere. Ulterior am prezentat clasa de atacuri *Spectre* care permite citirea zonelor de memorie accesibile unui proces victimă cu care un proces atacator are în comun o zonă limitată de memorie. Prezint de asemenea și soluții care au încercat să mitigheze cu grade variate de succes aceste vulnerabilități. În final, prezint o implementare inter-proces, cu scop didactic a *Spectre v1*.

Noua variantă a *Spectre* (*BHI*) publicată la începutul anului 2022 demonstrează ineficiența solutilor implementate până în prezent importriva *Spectre V2*. Astfel, *Spectre* rămâne un subiect deschis de cercetare. Se pot dezvolta în continuare noi variante ale atacului care pot fi eficiente chiar și împotriva mitigarilor ce vor apărea în viitorul apropiat. Se pot dezvolta noi mitigari care să afecteze performanța programelor într-un mod minimal, mai puțin decât cele deja existente care reduc performanța considerabil. Se pot evalua implicările atacului pe platforme mai puțin documentate.

În viitor, îmi doresc să studiez mai în detaliu *Spectre V2* și ulterior, noua variantă *Spectre BHI* apărută recent, pentru a înțelege mai bine mecanismele din spate care fac posibile rezultatele obținute. Noile cunoștinte dobândite, m-ar ajuta în contiuarea cercetării 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] Mark Campbell, *AMD releases response to Meltdown and Spectre exploits*, Accesat: 15.05.2022, 2018, URL: https://www.overclock3d.net/news/cpu_mainboard/amd_releases_response_to_meltdown_and_spectre_exploits/1.

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

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

[8] *Covert Channel: The Hidden Network*, Accesat: 15.05.2022, URL: https://www.hackingarticles.in/covert-channel-the-hidden-network/.

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

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

[11] *File Locks*, Accesat: 15.05.2022, URL: https://www.gnu.org/software/libc/manual/html_node/File-Locks.html.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] *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.

[19] *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.

[20] *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.

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

[22] 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.

[23] 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).

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

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

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

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

[28] 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/.

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

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

[31] 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.

[32] R. S. Shaw, *Call Stack Layout*, 2007, URL: https://commons.wikimedia.org/w/index.php?curid=1956587.

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

[34] 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.