

# Network and System Defense

Piercino Caliandro

22 agosto 2022

# Indice

<b>I</b>	<b>Parte Hardware</b>	<b>4</b>
<b>1</b>	<b>Blocco 1: Introduzione</b>	<b>5</b>
<b>2</b>	<b>Introduzione alla cybersecurity</b>	<b>6</b>
<b>3</b>	<b>Blocco 2: vulnerabilità intrinseche IP/TCP</b>	<b>8</b>
3.1	Architettura di IP . . . . .	8
3.1.1	Routing table . . . . .	9
3.2	Vulnerabilità di TCP/IP . . . . .	9
3.2.1	DNS spoofing . . . . .	10
3.2.2	Security requirements per le vulnerabilità . . . . .	10
<b>4</b>	<b>Blocco 3: Ethernet LAN security</b>	<b>11</b>
4.1	Ethernet LAN recap . . . . .	11
4.1.1	Hub e Switch . . . . .	11
4.1.2	Recap DHCP ed ARP . . . . .	12
4.2	Ethernet LAN vulnerabilities . . . . .	12
4.2.1	Network and system access . . . . .	12
4.2.2	Traffic integrity . . . . .	13
4.2.3	Contromisure . . . . .	14
4.3	IPv6 SEcure Neighbour Discovery . . . . .	16
4.3.1	IPv6 neighbour discovery . . . . .	16
4.3.2	Router Authorization . . . . .	18
4.3.3	Conclusioni . . . . .	18
<b>5</b>	<b>Blocco 4: Virtual LANs</b>	<b>19</b>
5.1	Background . . . . .	19
5.2	Background . . . . .	19
5.2.1	VLAN membership . . . . .	19
5.2.2	VLAN e sottoreti IP . . . . .	20
5.2.3	Formati per VLAN . . . . .	21
5.3	Sicurezza delle VLAN (per device CISCO) . . . . .	21
5.3.1	Media Access Control (MAC) . . . . .	21
5.3.2	Basic VLAN Hopping attack . . . . .	21
5.3.3	Double encapsulation . . . . .	22
5.3.4	ARP attack . . . . .	22
5.3.5	VLAN Trunking Protocol Attack . . . . .	22
5.3.6	CDP attack . . . . .	22

5.3.7	Private VLAN attack . . . . .	22
<b>6</b>	<b>802.1X</b>	<b>23</b>
6.1	Introduzione . . . . .	23
6.2	EAP . . . . .	23
6.3	RADIUS . . . . .	24
6.4	802.1X overview . . . . .	24
6.4.1	EAPOL . . . . .	24
6.4.2	EAPOR . . . . .	25
6.5	EAP in relay mode . . . . .	25
6.5.1	EAP-TLS . . . . .	25
6.5.2	802.1X Authorization: VLAN . . . . .	25
6.5.3	802.1X Authorization: ACL . . . . .	25
6.5.4	802.1X Authorization: UCL . . . . .	26
<b>7</b>	<b>Firewall and packet classification algorithm</b>	<b>27</b>
7.1	Introduzione sul firewall . . . . .	27
7.1.1	Access Policy . . . . .	27
7.1.2	Filtering characteristics . . . . .	27
7.1.3	Limiti e capacità dei firewall . . . . .	28
7.1.4	Categorizzazione dei firewall . . . . .	28
7.2	NETFILTER . . . . .	30
7.2.1	ALG related state . . . . .	30
7.2.2	iptables . . . . .	30
7.2.3	NAT . . . . .	31
7.3	Packet classification algorithms and data structures . . . . .	32
7.3.1	Problema del packet classification . . . . .	32
7.3.2	Requisiti e metriche . . . . .	32
7.3.3	Schema uni-dimensionale . . . . .	33
7.3.4	Algoritmo generalizzato per matching esatto . . . . .	33
7.3.5	Hashing exact match . . . . .	34
7.4	Schema bi-dimensionale . . . . .	35
7.4.1	Set-Pruning Tries . . . . .	35
7.4.2	Estensione dello schema bi-dimensionale . . . . .	36
7.5	Linear Bit Vector Firewall con eBPF/XDP . . . . .	37
7.5.1	Hookse programmi . . . . .	37
<b>8</b>	<b>Secure protocols and overlay VPNs</b>	<b>39</b>
8.1	Recap: protocolli di sicurezza di rete . . . . .	39
8.1.1	Livello applicativo: SSH . . . . .	39
8.1.2	Transport layer: TLS . . . . .	41
8.1.3	Livello rete: IPSec . . . . .	41
8.1.4	Attacchi a TLS . . . . .	41
8.2	Overlay VPN . . . . .	42
8.2.1	OpenVPN . . . . .	43
8.2.2	Interazione con NETFLITER . . . . .	45

<b>9</b>	<b>BGP</b>	<b>47</b>
9.1	Introduzione . . . . .	47
9.2	BGP in a nutshell . . . . .	47
9.2.1	Protocolli di gateway esteriori . . . . .	48
9.2.2	Scelta del best path in BGP . . . . .	48
9.3	BGP configuration . . . . .	49
9.4	MPLS . . . . .	50
9.4.1	MPLS e BGP . . . . .	51
9.4.2	VPN con MPLS e BGP . . . . .	51
9.5	BGP Security . . . . .	52
9.5.1	Vulnerabilità legate a IP/TCP . . . . .	52
9.5.2	Vulnerabilità del control plane BGP . . . . .	54
9.5.3	Tipologie di BGP peers e relazioni fra essi . . . . .	55
9.6	RPKI e BGP origin validation . . . . .	56
9.7	Forged-origin hijacks . . . . .	57
9.8	Prefix filtering . . . . .	58
9.8.1	Route leaks . . . . .	58
9.8.2	Altri tipi di filtering . . . . .	59
<b>10</b>	<b>DNS Security</b>	<b>60</b>
10.1	Introduzione . . . . .	60
10.2	DNS vulnerabilities . . . . .	61
10.2.1	Rendere sicuro DNS . . . . .	62
10.3	DNSSEC . . . . .	62
10.3.1	Explicit denial of Existence . . . . .	63
10.4	Esempio reale . . . . .	63
10.5	Zone content exposure . . . . .	64
10.5.1	NSEC3 . . . . .	64
10.5.2	Cosa risolve DNSSEC e cosa No . . . . .	64
10.5.3	DNS over HTTPS o TLS . . . . .	65
10.6	Lab 14: FINALE . . . . .	65
<b>11</b>	<b>VXLAN ed EVPN</b>	<b>66</b>
11.1	Data center networks . . . . .	66
11.1.1	Clos topology - 2 tier . . . . .	67
11.1.2	Clos topology - 3 tier . . . . .	67
11.2	VXLAN . . . . .	67
11.2.1	Load balancing: ECMP . . . . .	68
11.2.2	VXLAN VTEPs . . . . .	69
11.2.3	Utilizzi di VXLAN . . . . .	69
11.3	EVPN . . . . .	69
11.3.1	EVPN type 2 . . . . .	70
11.3.2	EVPN tipo 3 . . . . .	70

# Parte I

## parte software

# Capitolo 1

## introduzione

approccio bottom up a partire dall'hardware fino al software. imparare i componenti interni di ogni livello dello stack prima di imparare l'attacco, in modo da capire meglio come funzionano. a questo punto, capire come e se è possibile incrementare il livello di sicurezza. perché non dire che renderemo il sistema sicuro? è impossibile rendere un sistema sicuro, **un computer sicuro è scollegato dalla rete, con l'hw che prende fuoco.** argomenti che vedremo:

- attacchi hardware-based e contromisure
- os e principi di sicurezza
- virtualizzazione
- malware detection
- db security

# Capitolo 2

## hardware based attacks e contromisure

perché siamo interessanti gli attacchi all'hardware: perché ogni sistema it si basa sull'hardware, quindi compromettendo questo si riesce a compromettere tutto lo stack it. quando scrivo sw, mi aspetto qualcosa dall'hw e quindi se lo altero per il software è molto difficile capire che è compromesso. i blocchi fondamentali per costruire degli attacchi hw based di successo:

- leggere primitive, per circumnavigare la sicurezza del sw
- scrivere primitive
- covertness: se l'attaccante può fare un attacco che non può essere scoperto è impossibile per il sw capire che c'è un attacco in corso

la prima cosa da fare è capire come funziona l'hw

### 2.1 storia dell'hw

legge di moore: il numero di transistor di un calcolatore raddoppia di ogni anno.

quindi possiamo aggiungere sempre più componenti nel chip: tempo fa si aveva il sw, si cambiava la frequenza del processore e il sw andava più veloce. ma c'è un upper bound nella potenza consumabile, che è 130w e quindi anche aggiungendo componenti non si può sfruttare più potenza. non si può aumentare la frequenza oltre il power wall, l'ammontare della potenza nel calcolatore è talmente alta che non si può aumentare la velocità della cpu.

le cpu inizialmente erano multi ciclo e single core, già complesse ma lente. quindi la prima cosa fu quella di cercare di rendere più veloci:

1. la prima cosa fatta negli 80' fu l'introduzione del pipelining: allo stesso tempo, in parallelo, vengono processare diverse istruzioni che sono in diversi stadi della pipeline. si può committare una istruzione per ogni ciclo, ma si velocizza l'esecuzione delle istruzioni. quindi la frequenza della cpu venne aumentata, ma si può sempre committare una istruzione alla volta
2. architettura super-scalare, aggiunge la possibilità di committare più istruzioni perché gli elementi base come alu, più elementi per il fetch etc... sono ridondati. quindi, c'è l'istruzione nella pipeline e serve della logica nella cpu che dica quale hardware è disponibile e può essere assegnato alle istruzioni ogni volta, quindi serve un run-time scheduling. in ogni caso, ci sono i problemi della pipeline come i branch condizionali e la necessità di dati quando non ancora disponibili

3. dai problemi sopra, nasce la speculazione:

```
1 a → b + c
2 if a ≥ 0 then
3   d ← b
4 else
5   d ← c
```

fetcho un'istruzione prima di avere conoscenza del fatto che devo fare un salto piuttosto che un altro. quindi ci sono una serie di bolle necessarie anche per un codice semplice, c'è una penalità per gli stalli enorme ed il codice fa molti salti.

quindi, serve cambiare le cose: branch prediction.

## 2.2 branch prediction

se la predizione è sbagliata, si scarta il risultato, quindi si flusha una serie di stadi della pipeline perché incorretti, altrimenti si continua con l'esecuzione delle istruzioni. ci sono delle parti dedicate nel chip del processore apposite per il branch prediction, migliore sarà la predizione e meno stalli si osserveranno e quindi migliori prestazioni. le performance dipendono da due fattori: quanto presto si può controllare se la predizione è corretta e se la predizione è corretta.

ci sono due tipi di predizione:

- **dynamic prediction:** implementata in hw, la predizione cambia in base al comportamento del sw. basata tipicamente sulla storia dei branch
- **static branch prediction:** determinata a compile time, come ad esempio il *likely* nel codice del kernel. viene riflessa a livello assembly con dei prefissi come ad esempio *0x2e* o *0x3e* ma nelle architetture moderne non funziona più perché la pipeline è troppo complessa.

la branch prediction table è una piccola memoria nella cpu utilizzabile per capire l'outcome di una branch instruction: si usano i bit meno significativi dell'istruzione per indicizzare la tabella e prendere la decisione, il risultato è binario (take or not take).

se la predizione è corretta ed è take ho un ciclo di penalità: questo perché si esegue l'istruzione di salto, ma nella pipeline c'è comunque la fase di fetch per cui semplicemente la carico, ma per poter fare la predizione devo arrivare almeno alla fase di decode.

se la predizione è not take ed è corretta non ho penalità

se invece la predizione è incorretta, devo

- cambiare la predizione
- aggiornare la tabella
- flushare la pipeline, la penalità è la stessa che si avrebbe se non ci fosse la branch prediction, perché ci sarebbero gli stalli.

l'automa a stati che rappresenta i cambi di stato in base alla predizione è il seguente:

Figura 2.1: predittore a due bit



si possono usare due bit per decidere come spostarsi in base alla predizione. come si comporta con dei cicli annidati?

```
1  mov $0, %ecx
2  .outerloop:
3    cmp $10, %ecx
4    je .done
5    mov $0, %ebx
6  .innerloop:
7    ; actual code
8    inc %ebx
9    cmp $10, %ebx
10   jnz .innerloop
11
12   inc %ecx
13   jmp .outerloop
14 .done:
```

l'unità per la branch prediction continuerà a dire branch taken finché non si arriva a 10, alla fine c'è una decisione sbagliata e si torna al loop esterno. si cambia continuamente idea per via dell'interazione fra loop interno e loop esterno, per via di questo il 2-bit saturating counter non funziona bene perché sbaglia sempre la prima predizione e l'ultima.

la branch prediction è importante, perché da un'analisi risulta che nel codice il 20% delle istruzioni sono branch condizionali, inoltre ci sono pipeline più complesse e super-scalari, quindi la possibilità di trovare delle istruzioni condizionali è più alta ed inoltre c'è la programmazione ad oggetti, dove per capire se una classe è figlia di un'altra bisogna navigare l'albero per capire la classe padre e quindi prendere decisioni.

per migliorare la branch prediction:

- migliorare la predizione
- determinare il target prima
- ridurre la penalità per predizioni errate

### 2.2.1 correlated prediction

l'idea è che alcuni outcome dei branch sono correlati fra loro, quindi serve poter correlare due branch condizionali in assembly. in un correlated predictor si usa una storia degli  $m$  branch passati e si può usare la storia per correlare i branch. si usa la path history table:

- si mette la storia globale in un registro globale della storia
- si usa il valore per accedere ad una pht di 2-bit saturating

Figura 2.2: predittore correlato

Figura 2.3: tournament predictor

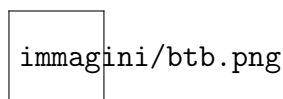


Figura 2.4: architettura del branch target buffer

### 2.2.2 tournament predictor

si usano due tipi di predittori:

- correlated predictor, basato sugli ultimi  $m$  branches, acceduto dalla storia locale
- local predictor, che viene selezionato usando l'indirizzo del branch

si usa poi un indicatore di quale è stato il miglior predittore per lo specifico branch in esecuzione, che è un contatore a 2 bit, incrementato per uno dei predittori e decrementato per l'altro. di seguito, uno schema dell'architettura del tournament predictor dec alpha 21264

#### branch target buffer

è relazionato ai jump indiretti, che sono difficili da predirre. esempio: la `ret` è l'istruzione di ritorno da una sub routine. il problema è capire dove si ritorna: qual è l'istruzione che si mette in pipeline, dovrebbe essere la prossima del chiamante ma è ancora più complicato. si può fare il jump nell'indirizzo contenuto in un registro, l'output dipende da cosa è scritto nel registro che si vuole usare per il salto.

il btb è una piccola cache che viene acceduta col program counter:

troviamo il prefetched target ed il prediction bit, che dice se conviene prendere il target oppure mettere uno stallo. è una cache, quindi ci può essere miss o hit, si può imparare se il prefetched target dovrebbe essere quello e più tardi si saprà se quel target è corretto o meno.

#### return address stack

l'istruzione di `ret` legge l'indirizzo dallo stack, quindi se possiamo leggerlo subito possiamo sapere il target del ritorno.

possiamo mettere uno stack nella cpu che manitene gli indirizzi di ritorno, ogni volta che c'è una chiamata, si salva in questo stack l'indirizzo di ritorno. in questo modo, ogni volta che si torna si conosce l'indirizzo di ritorno e questo è utilissimo perché l'85% delle istruzioni è di ritorno.

#### fetch both target

tecnica che costa molto perché riempi la pipeline con tutte le istruzioni, ma la metà verranno scaratate e non aiuta in casi di fetch multipli come lo `switch`, non viene usato nell'hardware off the shelf.



 immagini/trace\_cache\_hit.png

Figura 2.5: trace cache hit



 immagini/trace\_cache\_miss.png

Figura 2.6: trace cache miss

### 2.2.3 simultaneous multi-threading

c'è una cpu fisica ma è esposta al software come una coppia di cpu virtuali. l'idea è che mentre runni istruzioni nella pipeline usi un sotto-insieme delle risorse hardware, in questo caso nel simultaneous mt si può tenere traccia di che hw stanno usando le istruzioni, quindi si duplica lo stato dell'architettura, quindi ad esempio il pc in modo che il processore usi un algoritmo di scheduling in modo da dare il controllo ai differenti thread che girano. quindi è richiesto meno hw per far girare più processi ma servono dei controlli per evitare l'interferenza.

#### caso intel: hyperthreading

gli obiettivi sono molteplici:

- minimizzare la die area
- evitare che uno dei due processori logici produca stalli per l'altro
- "spegnere" l'hyperthreading se non necessario

oggi, anche l'architettura intel mostra delle istruzioni assembly al programmatore diverse da quelle effettivamente usate internamente, ci sono due parti:

- un front-end
- un engine di esecuzione out-of-order: se puoi eseguire una micro operazione va eseguita immediatamente.

**xeon frontend:** il ruolo è quello di tradurre istruzioni cisc in istruzioni risc usando una microcode rom. c'è una cache di micro operazioni ( $\mu$ -ops) che mantiene le traduzioni già fatte.

nella trace cache, sappiamo se una micro operazione è associata ad uno o all'altro execution program, grazie ai due program counter differenti.

ci sono quindi due code diverse di micro operation.

l'accesso alla trace cache è regolato dai processori logici ad ogni ciclo di clock e nel caso di una contesa l'accesso è alternato.

nel caso di hit avviene quanto segue:

nel caso di miss nella tc, serve fare la traduzione:

bisogna accedere all'Instruction TLB (ITLB) che permette di tradurre la corrispondenza fra indirizzi logici e fisici. ovviamente, ci sono due TLB associati alla cache L2, quindi si può pagare anche un miss dovuto alla L2. quindi, si mette la CISC instruction in coda, c'è poi il decoder ROM che mette le istruzioni in una coda usata per fillare la TC e quindi ora si può mettere la micro operation nella  $\mu$ op queue.

quindi: l'ITLB riceve una richiesta dalla trace cache per consegnare delle nuove istruzioni.

il next-instruction pointer viene tradotto in indirizzo fisico, mandando una richiesta alla cache L2, l'accesso è per processore logico ed è gestito su una base FIFO, con almeno uno slot per processore.

i byte delle istruzioni sono salvati in dei buffer da 64 byte.

**xeon out-of-order pipeline:** l'architettura la seguente:

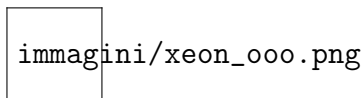


Figura 2.7: pipeline dei processori xeon

vogliamo eseguire le micro operazioni in ogni ordine quando è possibile farlo. nel primo stadio, vogliamo schedare i registri fisici della CPU: una  $\mu$ op cerca di accedere il registro ICX, non è uno solo ma sono molteplici ovvero un multi-registro e quindi ICX viene mappato su uno di questi. servono quindi un register rename ed un allocator per tenere traccia di tutto.

ora ci sono code multiple, per discriminare la semantica della  $\mu$ op in base al fatto che si acceda alla memoria etc...

lo stadio di execute ha diverse ALU, quindi l'allocation unit dice anche quale componente usare per eseguire l'istruzione. lo scheduler prende l'istruzione dalla coda nel momento esatto in cui il componente è disponibile. l'ordine è qualsiasi, le  $\mu$ op appartengono a diversi program flow, quindi data una CISC operation, le macro istruzioni saranno invertite nell'ordine.

alla fine, non vogliamo che le istruzioni siano materializzate all'esterno in ordine casuale, quindi c'è un reorder buffer per riordinare le istruzioni CISC nell'ordine in cui sono entrate e far credere all'utente siano state processate così.

non abbiamo quindi una idea di come siano state eseguite le istruzioni internamente, quindi ci può essere uno stato di inconsistenza per un tempo illimitato ma non è un problema perché questo non viene esposto a livello ISA. il problema è che stiamo interagendo con la memoria e questo è sfruttabile, possiamo ad esempio far sì che la CPU carichi delle istruzioni che non dovrebbe e quindi dei dati in cache che non dovrebbe: l'istruzione non verrà committata, ma la traccia sarà comunque esposta e sfruttabile.

## 2.2.4 multicores (2000s)

i vendor cominciano ad usare i molteplici transistor per creare delle CPU multicore, la prima CPU è l'IBM Power4, 2 core con una architettura interessante: una cache privata per ciascun core, interconnessione di un certo tipo fra i core e una cache di L2 ed L3, dove la L3 fa anche da memory controller. i core interagiscono con la L1 e così a salire, quindi per caricare dati dalla memoria. ma se i cores hanno una cache privata, come fanno ad essere sincronizzati con i dati contenuti in L1 e quindi visibili al core 2? serve un protocollo apposito per garantire la consistenza dei dati.

## 2.3 cache coherence

comportamento della cache da un punto di vista di utilizzo. la cc definisce la coerenza della cache con dei protocolli: quando scrivo codice, non mi rendo conto di interagire con la cache, ma i processori vi interagiscono di continuo. abbiamo ad esempio 2 cores, con due cache private ed una serie di operazioni: al tempo 0, nella cache del core 1 c'è la copia di a, a t=1 c2 carica a in un altro registro.

t	core c1	core c2	c1 cache	c2 cache
0	load r <sub>1</sub> , a		a:42	
1		load r <sub>1</sub> , a	a:42	a:42
2	add r <sub>1</sub> , r <sub>1</sub> , \$1			
	store r <sub>1</sub> , a		a:43	a: ??

poi c1 fa una una operazione su a, quindi il valore di a nella cache di c2 quale sarà? non c'è una sola risposta, dipenderà dal protocollo di consistenza

### 2.3.1 strong cc protol

il protocollo più semplice, tutte le cache dei vari core vedono gli stessi dati, come se non ci fosse una separazione fra le cache in un qualunque istante di tempo. quindi, serve avere un agreement fra i cores su chi fa l'operazione sul dato, per cui c'è un impatto sulla concorrenza e le performance.

per assicurare tale consistenza serve:

- single write/multiple readers: ad ogni istante di tempo, per ogni locazione di memoria un solo core può leggere e scrivere, oppure un certo numero di cores possono solo leggere dalla locazione;
- data-value (dv): il valore della locazione di memoria all'inizio dell'epoca è lo stesso che avrà alla fine dell'ultima epoca di lettura/scrittura

bisogna continuamente sincronizzare le cache quando un core fa un update in memoria, quindi non è molto performante

### 2.3.2 coerenza debole

c'è meno limitazione sulle cache, in quanto devono sincronizzarsi meno, quindi viene persa una delle invarianti dette sopra.

quindi due cores possono scrivere su una stessa area di memoria e questo può far sì che accada che il programmatore osservi dei comportamenti inattesi e quindi per implementare correttamente l'algoritmo occorre implementare delle primitive software per avere lo stesso outcome che si avrebbe se tutto accadesse sequenzialmente.

c'è una **time window** di un certo numero di epoche temporali in cui i dati non vengono propagati e questo può mostrare la complessità dell'hardware al programmatore.

è anche possibile non avere coerenza, che è il modo per avere le cache più veloci. non ci sono ovviamente garanzie, quindi le cache vanno sincronizzate esplicitamente lato software, ad esempio con delle api c/assembly.

### 2.3.3 cc protocol

un protocollo di consistenza per la cache è un algoritmo distribuito basato su primitive di tipo message-passing. ci sono due principali tipi di richieste in memoria da servire:


- load(a) per leggere il valore dalla locazione di memoria a
- store(a, v) per scrivere il valore v nella locazione a

e coinvolgono 2 tipi di attori principali, ovvero i cache controllers ed i memory controllers. il ruolo è quello di rinforzare una certa nozione di coerenza.

qualunque cosa nei protocolli cc è basato sulle coherency transaction, ciascuna di queste genera dei messaggi. ci sono due tipi principali di transazioni:

- get: carica un blocco di cache (un dato) in una linea di cache
- put: cancella un blocco dalla linea di cache, la linea diventa disponibile per altri dati

l'organizzazione è la seguente:

 immagini/cc\_arch.png

ogni linea di cache ha associato un bit che dice se la linea è valida o memo, in questo caso abbiamo una macchina a stati finiti che definisce lo stato del blocco in cache. la macchina è leggermente più complessa di una classica fsm, perché per transitare dai vari stati ci possono essere dei delay dovuti al fatto che il protocollo è distribuito, quindi avremo differenza fra:

- stati stabili, ovvero osservati all'inizio ed alla fine di una transazione
- stati transienti, osservati nel mezzo di una transazione

. ci sono poi

- eventi remoti: viene ricevuto un messaggio di coerenza
- eventi locali, ricevuti dalla parent cache controller


infine abbiamo le azioni:

- azioni remote, che producono la segnalazione di un messaggio di coerenza
- azioni locali, visibili solo al cache controller parent.


ci sono una serie di protocolli per gestire la coerenza, definiamo

- invalidate protocols: protocolli, quando viene scritto un blocco invalidano tutte le altre copie, quindi solo lo scrittore avrà la copia aggiornata
- update protocols: ogni update viene mandato a tutti i core.

quindi, nell'esempio iniziale il valore di a dipende dal protocollo implementato. ci sono poi due famiglie di cc protocols diversi

immagini/snooping.png

- snooping cache protocols: tutti i controller osservano tutto il traffico, quindi il broadcast è totalmente ordinato. è molto veloce, perché tutti i controller possono fare decisioni da soli in quanto osservano tutti i messaggi ma non è scalabile se il numero di core aumenta. questo perché serve un arbitro del bus, e ciò comporta un degrado delle performance
- directory based: c'è una directory, ogni request è unicast alla directory. questa tiene traccia di chi è il controller destinatario e gli manda il messaggio, il che rende tutto più scalabile e permette di aumentare il numero di cores.

immagini/directory.png

### vi protocol


il protocollo di consistenza più semplice, un solo cache controller può leggere e/o scrivere il blocco in ogni timestamp.

la transaction **get** è usata per richiedere un blocco in read-mode per poterne diventare l'owner e potervi scrivere e leggere. la **put** è usata per togliere il blocco dalla cache e riscriverlo al llc controller. ci sono una serie di eventi:

- **own-get**: transazione di **get** emessa dal controller locale
- **other-get**: transazione di **get** emessa da un controller remoto
- **any-get**: transazione di **get** emessa da qualunque controller
- **own-put**: transazione di **put** emessa dal controller locale
- **other-put**: transazione di **put** emessa da un controller remoto
- **any-put**: transazione di **put** emessa da qualunque controller
- **dataresp**: il blocco dati è stato ricevuto con successo

il protocollo vi sta per valid/invalid, ma oltre a questi due stati ce n'è anche uno intermedio per cui si aspetta una copia del dato up-to-date, l'automa a stati finiti è riassunto in seguito:

se confrontata con la fsm del llc, vediamo che c'è differenza: ad un certo punto, il core associato

immagini/vi\_prot.png

al cache controller fa una operazione di load, quindi ottiene l'indirizzo e va nella cache e scopre che lo stato della cache è i. quindi, serve una own-get dal llc, manda un messaggio che viene ricevuto da tutti i cache controller nel sistema con un ordinamento totale. una volta che il messaggio viene mandato, il controller transita nello stato iv, la copia verrà inviata dal possessore attuale del dato, dopo averlo ricevuto transita in v.

dopo, un altro core vorrà diventare l'owner e quindi questo controller transiterà nello stato *i* e trasferirà il contenuto del blocco.

immaginiamo di essere in una situazione in cui c'è un conflitto, ovvero si vuole scrivere un blocco nella linea di cache che è piena. si manda una **any-put** al llc controller, l'llc transita nello stato *i* e se poi qualcuno richiede il dato transita nello stato *v*.

il protocollo ha una nozione implicita di dirtyness: se una linea di cache è in stato *v*, il controller *ll* può o leggere e scrivere o solo leggere. ha anche un concetto implicito di esclusività, in quanto se lo stato è *v*, nessun altro può accedere al blocco, quindi nessuno ha una copia valida; inoltre, c'è nozione del possesso del blocco: l'owner sarà quello che manda la copia aggiornata a tutti.

la cosa positiva del portocollo è che richiede pochi bits per rappresentare le fsm, ma ha diverse inefficienze; ciò che vogliamo catturare del blocco di cache sono aspetti di

- **validità**: un blocco valido ha il valore più aggiornato per quel blocco. può essere letto ma scritto solo in modo esclusivo
- **dirtyness**: un blocco è dirty se il suo valore è quello più aggiornato e differisce da ciò che è scritto nell'llc o nella memoria centrale
- **exclusivity**: un blocco di cache è esclusivo se è l'unica copia in cache privata di quel blocco
- **ownership**: un controller della cache è owner di un blocco se è il responsabile di rispondere delle richieste di coerenza per quel blocco.

vogliamo queste proprietà per avere un protocollo efficiente ma che consumi più bits per l'implementazione

### **moesi stable states**

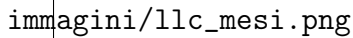
ogni lettera fa riferimento ad uno stato particolare, ognuno cattura una delle proprietà dette sopra. di nuovo, è una fsm per ogni blocco. i possibili stati per il blocco sono:

- **modified**: il blocco è valido, esclusivo, posseduto e potenzialmente dirty. può essere scritto o letto, la cache ha l'unica copia valida del blocco
- **owned**: blocco valido, posseduto potenzialmente dirty ma non esclusivo. può essere letto o scritto ma occorre rispondere alle richieste per quel bloccos
- **shared**: valido, non esclusivo, non posseduto, non sporco. ci sono più copie read only del blocco, può essere solo letto
- **exclusive**: il blocco è valido, esclusivo è clean. può essere solo letto e nessuna altra cache ha una copia del blocco.  
la copia in llc o memoria è aggiornata
- **invalid**: il blocco non è valido. la cache può contenere o non contenere il blocco, o il blocco è potenzialmente stale. non può essere nè letto e nè scritto.

quali sono le transazioni per poter implementare il protocollo:

- **gets**: get del blocco in shared state
- **getm**: get in modo modified





- **upgr**: ho un blocco, so che il blocco ha la copia più aggiornata perché ho osservato le transazioni, quindi non voglio la copia del dato ma voglio la possibilità di scrivervi, passando quindi da shared/owned a modified
- **puts**: cancella un blocco in shared state
- **pute**: cancella un blocco in exclusive state
- **puto**: cancella un blocco in owner state
- **putm**: cancella un blocco in modified state

### mesi protocol

il controller della cache di livello inferiore (llc) ha uno schema simile al seguente:

m/e è uno stato congiunto di modified/exclusive. se in m/e e si osserva una any-get, si va in shared, se qualcuno richiede il blocco, si va in i.

### 2.3.4 cc e cache write through

nelle cache write through, si propaga direttamente l'update.

nelle architetture moderne, le cache non sono più write through, in quanto la complessità può esplodere.

inoltre, nelle architetture moderne, le cache possono essere sia inclusive che esclusive ai diversi livelli, e questo può essere sfruttato in attacchi che sono specificatamente indirizzati alla cache di livello l\*. la cc decide come i diversi controller delle cache interagiscono fra di loro, c'è quindi il concetto di **consistenza della memoria**

## 2.4 memory consistency

il mc model definisce il comportamento di una memoria condivisa, indipendentemente da come è implementata. vediamo ancora un esempio: in macchine multi core c'è il riordino degli accessi in

core c1	core c2
s1: store data = new	l1: load r1 = flag
s2: store flag = set	b1: if (r1 $\neq$ set) goto l1
	l2: load r2 = data

memoria: il core commetta le operazioni sulla memoria in un certo ordine ma gli altri cores possono osservarle in ordine diverso, quindi non c'è un ordinamento totale. il sotto-sistema può quindi riordinare le operazioni: ci sono 4 combinazioni di possibili riordini:

- store-store reordering
- load-load reordering

core c1	core c2
s1: x = new	s2: y = new
l1: r1 = y	l2: r2 = x

- store-load reordering
- load-store reordering

quindi se scriviamo codice concorrente, dobbiamo considerare il fatto che ci siano riordini delle operazioni.

questo ha a che fare con due ordinamenti diversi:

- ordiamento di programma, è per core ed è totale. cattura l'ordine in cui ogni core esegue logicamente le operazioni sulla memoria
- ordinamento di memoria che è un ordinamento che vale per tutto il sistema e cattura l'ordine con cui la memoria serializza logicamente le operazioni da tutti i core

possiamo avere diverse consistenze:

- sequenziale: ci sono determinate operazioni che verranno osservate identicamente tra ordine di programma ed ordine di memoria.  
nell'esempio di prima, solo due ordini sono consistenti sequenzialmente
- consistenza debole: modello implementato dalle architetture moderne. quella di intel è total order store. c'è un piccolo buffer che agisce come coda in cui vengono salvate le diverse operazioni di store e che verranno poi processate. le architetture intel sono fra quelle in cui c'è il minimo numero di riordini tra le operazioni

### 2.4.1 memory fence

se ho una qualunque operazioni di memoria, la fence forza l'ordine di una load/store, quindi possiamo creare dei punti in cui siamo sicuri che tutte le istruzioni prima della fence sarà ordinata e stessa cosa se accade prima, ma non si possono riordinare le fence. sono quindi operazioni speciali che permettono di mettere dei punti di riordino per poter ad esempio avere consistenza sequenziale in intel.

#### fence su architetture x86

ci sono 3 tipi di fence diverse su x86:

- **mfence**: barriera full memory
- **sfence**: barriera store/store
- **lfence**: barriere load/load e load/store

## 2.5 in memory transaction

le transazioni in memory permettono di realizzare la sincronizzazione in memoria più semplice. può essere esplicita, quindi per accedere ad un sezione critica si prende un lock, si accede e poi si esce. ci possono essere problemi sui locks come deadlock, i deadlocks non sono componibili e quindi possono esserci delle inconsistenze nei risultati di operazioni che usano i lock.

l'hardware transactional memory prevede di usare delle istruzioni di assembly specifiche per dire che si vuole accedere a dei dati in maniera esclusiva, così che il processore metta un lock privato sui dati e una volta finito, se il commit va a buon fine i dati vengano pubblicati all'esterno

per poter implementare le hw transaction, intel ha modificato il protocollo di cc: i tentativi vengono fatti solo al primo livello della cache e si marcano i blocchi di cache con lock, per poi committare se va tutto bene. ci sono 4 nuove istruzioni assembly

- **xbegin**: inizio transazione
- **xend**: termine transazione
- **xabort**: abort della transazione
- **xtest**: controlla se il codice sta girando all'interno di una transazione

esempio in c:

```
1  while(1){
2      int status = _xbegin();
3      if(status == _xbegin_started){
4          // transaction body starts here
5          ....
6          ....
7          // transaction body ends here
8          _xend();
9          break;
10     } else{
11         // actions on aborts
12         ....
13         ....
14     }
15 }
```

una transazione può andare in abort per vari motivi: le istruzioni all'interno di una transazione di read e write costituiscono il read set e write set nelle cache. il write set è marcato nelle cache l1, mentre il read set nella l1-l3.

- se un'altra cpu legge una locazione nel write set della cpu corrente;
- se avviene una scrittura in una locazione di locazione che è nel read e write set

la transazione può anche abortire per via di limitazioni hardware:

- context switch
- interrupts
- page faults

- aggiornamento dei dati per la traduzione da memoria virtuale a fisica
- limiti nella taglia della cache l1 ed l3.

da un punto di vista di performance il problema è per cosa si fa abort a livello hardware, ma la cosa interessante è che la transazione può essere mandata in abort da qualunque altro cpu core. se supponiamo di mettere del codice per fare delle transazioni, rischiamo che dopo se mettiamo del codice su quello scritto per fare i lock possiamo avere una esecuzione per cui in concorrenza le cose non tornano (vedi l'esempio dei depositi/ritiri sulle slides), perché i lock non si compongono.

si possono usare le transazioni, concetto preso dal dbms: quindi, vogliamo eseguire tutte le operazioni fra l'inizio e la fine di una transazione come all-or-nothing.


è possibile farlo sia usando un dbms, sia tramite l'uso di facilities offerte dall'hardware.

la facility hardware è quella delle **hardware transaction memory**: l'idea dietro è che se nulla va storto, si hanno delle performance migliori. usati molto negli engine per videogiochi, quasi tutti i processor off the shelf offrono transactional memory, intel li ha disabilitati quasi del tutto per via di una implementazione buggy.

il paradigma hdm lavora in questo modo:

- introdurre operazioni assembly per capire quando inizia e finisce la transazione
- utilizzo delle cache, i controller sono già in grado di capire se operazioni concorrenti stanno accedendo allo stesso dato e quindi se scoprono un conflitto sui dati, droppano il flusso di esecuzione ed anche i side effects sui dati. ci sarà poi un retry path

il tipico hardware off the shelf è mostrato di seguito, usato per fare il reverse engineering di come è fatto l'htm la l1 e la l3 possono essere usate per implementare le transazioni, in quanto possono

immagini/htm\_hw.png

essere usate per rappresentare le transazioni. ci sono due gruppi di dati, read set e write set. l'implementazione delle htm introduce un bit aggiuntivo per ogni cache line, che dice se un blocco è stato caricato o letto durante una transazione.

il protocollo di cc sa quali linee sono marcate come transaction access, se ci sono concorrenze e può agire per rendere lo stato coerente. (slides)

### 2.5.1 funzionamento

le istruzioni assembly usare per supportare htm sono

- xbegin
- xend
- xabort
- xtest

la cosa buona di questa implementazione è che sulla base di queste 4 operazioni è possibile scrivere delle transazioni annidate, il che rende la scrittura del codice molto più semplice; il cc controller tiene un contatore delle nested transaction fatte, quando il valore va a 0 pulirà i bit dalla cache.

tutti i compilatori moderni possono generare queste 4 transazioni, a partire da linguaggi di alto livello come c e c++, generando delle undefined instruction se htm è implementato in hardware.

è possibile capire anche se c'è un abort in corso, quindi è utile se abbiamo una transazione grossa capire se c'è un abort perché magari il processore non è in grado di portarla a commit.

il write set è vicino alla cpu, sta nella l1, mentre il read set è sia in l2 che in l3. quindi, il cache controller setta il t flag sia in l1 che in l3 quando si legge, mentre solo in l1 quando si scrive. supponiamo di avere l'esempio del bank account: se ci sono due diversi cache controllers che stanno gestendo le transazioni, non possono conoscere lo stato l'uno dell'altro ma possono verificare che ci sia un accesso concorrente e quindi se ogni altra cpu legge una location di un write set, la transazione andrà in abort: c'è un pezzo di dati che è marcato con t, non è ancora committata ed il controller deve mandare il dato più fresco e se c'è una read non può mandare il dato fresco e quindi la transazione deve abortire.

vale lo stesso per le altre cpu: se qualcuno scrive in un read o write set, la transazione va in abort, l'algoritmo è quindi locale al controller, può dare delle performance peggiori per via che si mandano in abort anche cose che non dovrebbero.

inoltre ci sono anche limitazioni hw, cosa accade se abbiamo un contex switch: non possiamo lasciare dei dati incommittati nella cache, quindi la transazione deve abortire e lo stesso vale per la ricezione di un interrupt, perché non si sa per quanto tempo si starà in questo stato e non si possono lasciare i dati inconsistenti; lo stesso vale per page faults.

quindi è necessario avere della logica di codice per gestire gli abort

i codici di errore possono essere vari **esempio:**

abbiamo una transazione, vediamo l'effetto sulla cache

```
1  xbegin:
2      a0 += 4;
3      a1 -= 4;
4  xend
```

abbiamo solo due locazioni di memoria coinvolte nella transazione, vediamo cosa accade in una possibile run concorrente:

- $cpu_0$  legge  $a_0$  dalla cache e setta il t bit
- $cpu_0$  scrive su  $a_0$ , portando quindi lo stato a modified, quindi  $a_0$  è nel write set
- $cpu_1$  legge  $a_1$  nella cache (in maniera esclusiva) e setta il t bit

lo stato della cache è il seguente:

(vedi slides)

quindi è possibile committare semplicemente pulendo lo stato dei bit.

se  $cpu_1$  entra in gioco e legge  $a_0$ , la  $cpu_0$  deve per forza fare abort: pulisce i t bit e rende le linee di cache invalide che erano modified. ora,  $cpu_1$  deve per forza leggere i dati dalla memoria, perché in cache sono invalide, questo avviene proprio quando  $cpu_0$  stava per fare il commit.


## 2.6 recap sulla gestione della memoria virtuale

la memoria virtuale serve per far sì che le applicazioni non accedano alla memoria fisica direttamente, bensì tramite degli indirizzi virtuali. l'idea è di astrarre cosa una applicazione vede e cosa il processore dovrebbe fare, per molteplici motivi


- rendere la scrittura del codice più facile;
- ingannare l'applicazione, facendole credere di avere più memoria in quanto si può fare lo swapping sulla memoria;
- isolamento per processo: se qualcuno nel sistema può trasformare un indirizzo virtuale in uno fisico ed il componente è abbastanza sicuro, ci sono delle delimitazioni che impediscono ai processi di vedere i dati di altri processi, a meno di usare memoria condivisa

il supporto alla memoria virtuale usa sia hardware che firmware, che os.

usiamo la terminologia di **virtual page** e di **segmento di memoria**: una pagina virtuale viene mappata su un frame fisico. è possibile che un certo range degli indirizzi virtuali condividano dei

immagini/frame\_pages\_mapping.png

frame fisici in memoria. il firmware deve fare diverse operazioni per poter usare le pagine virtuali, quindi viene introdotto una cache addizionale per facilitare la traduzione tra virtual e phisical address: è il tlb, che se ha la traduzione in memoria da l'indirizzo fisico corrispondente all'indirizzo virtuale. i passi della traduzione sono riassunti di seguito: si può avere un indirizzo virtuale valido, che però

immagini/virt\_to\_phis.png


non è presente nella memoria fisica per via dello swapping, quindi si ha un miss nel tlb o nella page table.

### 2.6.1 traduzione in x86

le cpu intel sono bloated, quindi cercano di fare di tutto per dare retro-compatibilità, per questo ogni volta che c'è una nuova facility viene disabilitata per default, quando parte un nuovo chip intel è come se fosse un 8086, ci pensa il so ad abilitare tutto ciò che serve:

- segmentazione di memoria
- segmentazione di memoria in protected mode
- paginazione

l'indirizzamento basato sulla segmentazione è fatta in base a questo schema: come funziona l'unità


immagini/seg\_based\_transl.png

di segmentazione: 4 registri a 16 bit (allo startup girano così) per i segmenti

- cs: code segment, dove trovare il codice in memoria
- ds: data segment, dove trovare i dati in memoria
- ss: stack segment
- es: extra segment

intel i386 ha aggiunto due nuovi registri, fs e gs senza uso predefinito.

basandosi sulla segmentazione, si possono definire dei segmenti di memoria in cui ogni segmento è un numero che indica l'inizio del segmento, quindi è possibile avere overlapping. un esempio è mostrato in seguito:


immagini/seg\_based\_transl.png

### registri x86\_64

fra i tanti registri, i due di interesse sono cr0 e cr3.

abbiamo bisogno di strutture dati, che è un virtual-to-physical tree (tabella delle pagine) che mantiene le tabelle di traduzione e la cui root è indicizzata da cr3.

## 2.6.2 segmentazione protected mode

immagini/seg\_prot\_mode.png

tra le informazioni che otteniamo, dovremmo avere la base dell'indirizzo ma in realtà si ha un indice che serve per accedere ad una tabella in memoria centrale. la tabella ha dei segment descriptor che mantengono la base address per i segmenti, può essere ovunque in memoria per cui si può avere l'indirizzo nel registro `gdtr`.

supponiamo di avere una `jmp` che ha un target: l'indirizzo diventa un offset, nel `cs` register si trova un indice nella global descriptor table. a quel punto si usa l'index come offset nella tabella e trova il segment descriptor, quindi si prende la base e si somma all'offset per ottenere un indirizzo logico.


la base è sempre 0, per via della retro-compatibilità, ma il processo va comunque fatto per rendere possibile la paginazione (lo fa il so). quando rendiamo possibile la paginazione, l'indirizzo è lineare e viene passato all'unità di paginazione (firmware) che lo trasforma in un indirizzo fisico e lo passa alla cpu

### unità di paginazione

l'idea è che l'indirizzo lineare viene diviso in 3 pezzi (32 bit):

- directory: indice alla tabella puntata da cr3
- table: indice nella page table
- offset, che viene usato per trovare l'indirizzo fisico del frame

ogni tabella ha nel record un puntatore al componente successivo, sono tutte mantenute in memoria e la gestione è fatta dal so. c'è un albero di tabelle per ogni processo. quando si fa context switch, il valore del cr3 non viene cambiato e quindi è possibile anche trovare indirizzi del kernel.



 immagini/i386\_paging.png

### 2.6.3 traduzione virtual-to-physical i386

l'istruzione assembly ha un indirizzo logico, che verrà usato come offset per la tabella dall'unità di segmentazione per ottenere un indirizzo logico, che verrà usato dalla page table per navigare una pagina ed ottenere un indirizzo fisico da passare alla cpu. ogni volta che si accede ad un singolo byte di memoria si triggera tutto il processo

quindi per questo è stato introdotto il caching, per cercare di ridurre la latenza. per usare l'indirizzo fisico, il cache controller deve aspettare che tutta la traduzione avvenga, se si usa solo l'indirizzo logico c'è un problema se le tabelle cambiano.

è possibile usare entrambe le cose: appena l'applicazione genera un indirizzo lineare, questo viene preso dal firmware del cache controller l1, e se il dato è in memoria c'è un hit. in parallelo c'è il controllo nel tlb per vedere se è possibile tradurre in indirizzo fisico. se c'è un miss nel tlb parte la traduzione per ottenere l'indirizzo. quindi diversi livelli di cache sono indicizzati con diversi indirizzi, per cercare di velocizzare il processo di traduzione si fanno diverse cose in parallelo.


 immagini/vtp\_riassunto.png

## 2.7 primitive di lettura

vogliamo trovare un modo per bypassare l'isolamento dei processi, la tecnica da usare sarà il timing: vogliamo mettere su delle primitive di lettura, l'idea fondamentale dietro la sicurezza è che non vogliamo lasciare delle informazioni, a volte anche il tempo impiegato per fare un'operazione dice qualcosa.

### 2.7.1 algorithm timing

prendiamo il seguente esempio:

```

1 int strcmp(char *t, char *s){
2   for( ; *t == *s ; s++, t++){
3     if(*t == '\0')
4       return 0;
5   }
6   return *t - *s;
7 }
```

passando le due stringhe "abcd" ed "abxde" possiamo inferire qualcosa dai dati in base al tempo impiegato per fare il confronto.

useremo degli approcci simili sulla cache: ogni volta che leggiamo i dati dalla cpu, questi vengono letti dalla cache e quindi ogni volta che si fa una load, a seconda della condizione, ci saranno differenti risultati

- supponiamo di avere la cache in modified, siamo i proprietari e possiamo leggere



- se il blocco è invalid, occorre fare altre azioni

dipende tutto dallo stato in cui è la fsm della cache. l'idea è che possiamo fare delle operazioni sulla cache privata in modo da portare la fsm in degli stati che conosciamo, quindi per fare sfruttare un **side channel**

- portiamo la cache in uno stato noto
- facciamo sì che qualcuno faccia delle operazioni sulla cache per cambiarne lo stato
- vediamo, tramite timing, come è cambiato lo stato

con l'hyperthreading è ancora più critico, perché ci sono più thread che girano sullo stesso core e quindi condividono la stessa cache l1 e quindi potremmo leggere i dati di qualcun altro e quindi violare la privacy della l1 per via del simultaneous mt; ((nb: se non ci fosse, la l1 rimarrebbe privata. inoltre, la l1 di usa indirizzi virtuali per indicizzare i blocchi dati, quindi è anche più semplice accedere ai dati.))

### 2.7.2 code path

ci sono degli algoritmi su cui non si può fare timing, perché l'algoritmo ci metterà sempre lo stesso tempo. ad esempio abbiamo il montgomery ladder, usato per fare operazioni sulle curve ellittiche e blocco chiave della crittografia. abbiamo delle nonce, che devono essere univoche altrimenti si potrebbe invertire la curva (ricavare la chiave privata), lo pseudocode è il seguente:

```

1  input: point p, scalar n, k bits
2  output: point np
3  r_0 ← o
4  r_1 ← p
5
6  for i from k to 0 do:
7      if n_i = 0 then
8          r_1 ← r_0 + r_1
9          r_0 ← 2r_0
10     else
11         r_0 ← r_0 + r_1
12         r_1 ← 2r_1

```

abbiamo k bit, ma il numero di operazioni è sempre lo stesso. accediamo allo scalar, 1 bit alla volta, ed a seconda del valore accediamo ad uno dei rami dell'if-else e quindi ci chiediamo se possiamo determinare in quale dei due branch siamo, sfruttando la l1 per ricavare la nonce. una volta che il codice è caricato, viene messo in cache ed alcune operazioni sono grandi e quindi finiscono su più linee di cache, quindi se riusciamo a portare lo stato della gerarchia della cache in modo da capire in quali linee di cache è stata eseguita l'operazione, possiamo sapere se è stato preso l'if o l'else. quindi, anche se l'algoritmo è time independent, possiamo osservare quali linee di cache sono state usate per accedere ai dati

## 2.8 side channel attacks

un side channel è una zona di memoria che permette di leggere un altro contenuto di memoria o accedere a dei pattern di dati. ci sono diverse tipologie di side channels

- prime + probe
- flush and reload
- flush + flush
- evict + time
- evict + reload
- prime + abort

l'idea è sempre che la prima parte viene usata per portare la cache in un certo stato, con la seconda si cerca di capire dopo che la vittima ha fatto qualcosa, qual è il side effect sullo stato della cache. non c'è nulla che il so può fare per garantire l'isolamento dei processi, perché lo stato dell'hardware è condiviso. un attacco passa per diversi stati, le differenti macchine cambiano per diversi aspetti quindi abbiamo

1. pre-attack: il pre-target attack serve per acquisire il target, quindi ad esempio la linea di cache o il cache set (nel caso di cache n-associative) stabilire eventuali timing threshold. dobbiamo considerare tanti elementi, come ad esempio il carico di lavoro della macchina, se è alto potremmo essere deschedulati e perdere ad esempio la cpu su cui stavamo lavorando venendo rischedulati su un'altra
2. active attack:
  - a) inizializzazione: portare il canale in uno stato noto
  - b) attendere che la vittima faccia un accesso in memoria
  - c) analizzare l'accesso, osservando i side effects lasciati dalla vittima
  - d) ripetere l'accesso fino al leak dei dati

ci sono diversi aspetti da considerare

- le cache sono cachate sia virtualmente che fisicamente
- le cache sono condivise in maniera differente in base al livello
- vanno considerati anche gli interleave di esecuzione fra i diversi processi

mettere su un vero side channel attack è molto più complesso di cosa si legge nei paper

### **evict + time**

il target è la cancellazione di una linea, quindi inizialmente la vittima gira e fa degli accessi in memoria, quindi carica i suoi dati in memoria. a questo punto, stabiliamo una baseline execution, ovvero sappiamo che la vittima girerà di nuovo e quanto ci metterà quando ha i dati in cache. l'attacker quindi cancella una linea di interesse dalla cache per scoprire se la vittima la usa: in algoritmi crittografici, si ci basa su tabelle di dati condivise che vengono usati in base alla chiave. quindi se sappiamo quale parte della tabella viene usata, possiamo fare una ricostruzione di discovery code path, quindi capire quale parte dell'algoritmo viene usato mentre gira. questo è stato effettivamente usato per rompere aes, possiamo scoprire quali linee di cache sono state usate per capire ad esempio

la chiave di cifratura, usando il tempo: vediamo quanto ci vuole per fare degli accessi per capire se la vittima sta usando una cache line cancellata o no.

la parte critica è poter far girare la vittima quando si vuole, ad esempio è stato usato per rompere aes perché è una libreria condivisa (nb: condivisa fra più processi) e quindi è possibile per l'attaccante far girare la vittima per molto tempo.

non usare solo l'istruzione assembly `rdtsc` per contare il tempo che passa, i chip del processore vengono spenti/accesi in base alla temperatura del processore (perché si tocca il power wall) così come anche viene **decrementata la frequenza del processore**, quindi il numero di cicli effettivi contati è falsato. usiamo l'istruzione `rdtscp` perché rispetta il program order, altrimenti bisognerebbe usare le fence (ricorda l'esempio di quaglia). la cosa interessante è che l'attacco è veloce quanto la vittima è veloce nel computare.

nota: nel codice, vengono usati buffer di caratteri di 4096 byte, perché? la gerarchia delle cache sono state introdotte nel calcolatore per sfruttare la località in modo da fare andare tutto più velocemente. uno dei principi è la **località spaziale**, quindi possibilmente se viene caricata una linea di cache, anche se viene caricata una sola linea e quella successiva è associata ad uno stato invalid, la cache potrebbe pre-fetchare una linea di cache. quindi, dobbiamo considerare il pre-fetching quando buildiamo un attacco, essendo sicuri che le linee pre-fetched vengano flushate. 4kb è una pagina ed è talmente grande che anche col pre-fetching siamo abbastanza sicuri che non ci siano effetti indesiderati dopo la flush della linea di cache.

### flush + reload

basato sull'abilità di usare memoria virtuale condivisa, quindi affinché funzioni il pre-requisito è che sia possibile condividere i dati virtualmente con un altro processo, quindi diverse applicazioni condividono pagine di memoria per risparmiare spazio (si usa la copy on write, appena si cambia un byte si duplica la pagina). flushamo una linea di cache, possiamo farlo con un indirizzo virtuale, eseguiamo la vittima e poi proviamo a ricaricare le linee di cache flushate prima e misurare il tempo. se flushamo la linea e viene ricaricata dopo un po', nessuno l'ha toccata, altrimenti vuol dire che una vittima ha toccato quella linea di cache e quindi ha usato una parte di dati. la vittima viene fatta girare una volta sola, se non c'è noise sulla gerarchia di cache basta girare una volta sola.

nel codice, viene allineata la pagina di cache, la prima cosa che si fa è materializzare il vettore di probe in memoria: l'approccio di linux è che il kernel non si fida dello sviluppatore, quindi se chiede molta memoria l'os ne dà di meno se si accorge che è fatto di zeri. quindi, l'indirizzo di memoria virtuale è valido, ma non è mappato in memoria fisica del tutto, c'è una pagina in linux che è piena di 0 (la zero page) che viene condivisa da tutte quelle strutture con tutti zeri. in questo modo si dice al kernel che si vogliono davvero le 256 pagine, semplicemente scrivendo un byte.

ora, bisogna discriminare fra cache hit e miss e mettiamo delle `lfence` per essere sicuri che non ci sia della noise aggiuntiva per via della memory consistency.

nb: nell'attacco, creiamo una pagina per ogni singolo valore del byte che andiamo a leggere. quindi nell'accesso in `make_side_effect` (codice) leggo una pagina della memoria prendendo uno qualsiasi dei valori del byte, quindi creo un pointer in una delle pagine del vettore che verrà portata in cache, che sarà associata al valore del byte. se troviamo la pagina numero 0 in cache, vuol dire che il valore del byte è 0, se invece trovo la pagina 1, vuol dire che il byte letto è 1 e così via.

quindi, mi baso su due cose

- uso la grandezza di una pagina per mitigare il cache pre-fetching

- carico un valore in memoria senza far girare la vittima. se ho l'indirizzo di memoria virtuale e posso leggere un valore in memoria, posso leggere un valore, portarlo in un vettore e poter vedere se quel byte viene acceduto


perché non leggere direttamente il valore del puntatore? l'indirizzo può essere valido, ma non è detto che siamo in grado di leggere perché non abbiamo privilegi di lettura in memoria.

otteniamo un seg fault se la memoria non è valida, ma lo possiamo gestire con un handler e continuare con l'esecuzione e quindi possiamo bypassare i privilegi di accesso perché abbiamo portato dei dati in memoria. questo è il fondamento di meltdown.

### **prime + probe**

si fa girare in cache l3, non serve per forza smt, si può anche lavorare fra diverse vm. la tecnica prevede di fare un priming della cache, ovvero portarla in uno stato noto caricando dei dati che conosco in cache, mettendo uno o più linee di cache che riempiano tutte le linee in quanto la cache è set-associativa.

una volta fatta girare la vittima, questa deve invalidare delle linee di cache e carica i suoi valori, a questo punto possiamo rigirare e fare il probing del cache set caricato prima tramite una operazione di probing: se il tempo di accesso aumenta, sappiamo che la vittima ha toccato quelle linee di cache. un attacco di successo di prime + probe non è così semplice perché mettere su un cache set non è così semplice in una cache set-associativa

immagini/prime\_probe.png

ci sono delle contromisure hardware, il mapping memory-to-cache cambia di continuo. l'unico modo per mettere su l'attacco è conoscere il mapping a run time:

- scegliere n indirizzi di memoria casuali. c'è la possibilità di un cache collision perché non conosciamo il mapping con la memoria
- misuriamo il tempo che ci vuole per caricare la cache accedendo agli indirizzi di memoria
- se becchiamo una self cache collision, rimuoviamo l'indirizzo dal set. possiamo usare cpuid per capire l'architettura della cache

### **prime + abort**

permette di fare tutto ciò che è stato detto fin ora senza timing. tramite le transactional memory, avremo delle hardware callback che ci dicono che c'è stato un accesso in memoria. l'idea è quella di fare una transazione, aspettare per un abort e nel momento in cui l'abort avviene sappiamo chi ha fatto l'accesso alla memoria

sulla cache l1:

- una linea di cache è stata scritta durante una transazione
- l'accesso transazionale comporta una scrittura in memoria
- possiamo quindi monitorare le cancellazioni dalla l1, potendo spiare solo i thread che girano sullo stesso core grazie al smt (simultaneous multithreading)

possiamo anche mettere su degli attacchi alla cache l3: una linea di cache scritta durante una transazione viene tolta dalla l1, quindi l'accesso transazionale fa una write in memoria.

### **flush + flush**

la **clflush** non è idempotente rispetto al timing, quindi a seconda dell'input ci metterà tempo diverso. facciamo una **clflush** e la cpu dice al cache controller di fare un flush ma a seconda dello stato della cache il tempo impiegato sarà diverso. è una versione del flush + reload che permette di capire se la cache line è stata invalidata da qualcuno. è molto stealthy come attacco, perché non si fa molto rumore ed inoltre è veloce. quindi, anche la **bandwidth** dell'attacco è aumentata, perché il numero di dati che si riescono a scoprire è maggiore.

## 2.9 out of order pipeline

usando i side channel attacks visti, vediamo cosa possiamo fare introducendo la out of order execution pipeline: possiamo fare il leak dei dati, se facciamo girare le micro-istruzioni in pipeline in maniera speculativa. l'architettura non è quindi sicura al 100% se le micro-istruzioni verranno portate a termine, lo fa per velocizzare la pipeline ed alla fine ci può essere il commit o lo squash della pipeline se il check sulla istruzione non torna: se viene fatto il commit, le conseguenze verranno esposte sull'architettura della cpu, altrimenti no. ma anche se la cpu capisce che una operazione fallisce e non deve essere portata a commit, le conseguenze sulla micro-architettura ci sono: se facciamo una micro-operazione che fa accesso sulla cache, gli effetti rimangono anche se l'istruzione viene tolta dalla pipeline. questo è l'effetto degli attacchi di tipo **transient execution**, attacchi come

- spectre
- meltdown

che sono usciti nel 2018, ma la comunità di ricerca aveva già intuito della loro potenzialità 10 anni prima. per riassumere

- eseguiamo istruzioni speculative nella pipeline
- possono essere rimosse, ma gli effetti rimangono nello stato micro-architetturale

### **meltdown primer**

costruiamo un probe array, abbiamo una pagina per ognuno dei possibili valori di un byte, quindi facciamo il leak di un byte alla volta. non vogliamo leggere il byte, ma misurare l'effetto delle esecuzioni transienti, per capire se alcuni dati sono stati acceduti durante una esecuzione speculativa. facciamo il flush del contenuto dell'array dalla cache. ora, cerchiamo di accedere degli indirizzi di memoria, ad esempio di un buffer del so e cerchiamo di de-referenziarlo, quindi di leggere un byte. il kernel non vuole esporre all'utente le strutture dati usate, quindi l'indirizzo del kernel non è leggibile dall'utente e quindi ogni volta che proviamo ad accedere c'è un **segfault**:

- cerchiamo di accedere ad un indirizzo
- passiamo un indirizzo virtuale

- viene tradotto in fisico tramite la page table, c'è un bit che dice che non si può accedere
- il so manda una trap e viene eseguito del codice per gestirlo

siccome il `sefault` è un segnale, si può intercettarlo e fare recovery, come ad esempio dire al so di continuare a girare.

l'attacco è utile in quanto la traduzione indirizzo virtuale-fisico ha bisogno di tempo, siccome giriamo speculativamente, l'istruzione viene lasciata in pipeline per un certo intervallo di tempo. l'attacco funziona per un bug della cpu, poiché se possiamo fare la traduzione dell'indirizzo del kernel il firmware sa già che quell'indirizzo non è accessibile a livello user, ma per via di un design errato della cpu questo controllo viene fatto a ret time. infatti, cpu come arm ed alcune amd questo check viene fatto subito e quindi non sono vulnerabili a questo tipo di attacco, cosa che non avviene in intel cpu: quindi, in questo caso viene comunque fatta la load nella cache, ma l'attacco in se non ha molto senso di esistere.

ora ci sono delle patch hardware per cui non si soffre più di questo problema: l'istruzione viene considerata phantom, quindi non completerà mai, alla fine dell'esecuzione verrà squishata.

quindi per via di un design sbagliato della cpu, il cache controller fa comunque operazioni di load dell'indirizzo per una istruzione che è doomed e quindi non verrà mai committata. abbiamo quindi un byte, lo moltiplichiamo per la taglia di una pagina, quindi il byte diventa la probe page nel probe array e quindi un offset con cui possiamo accedere all'address space dell'attacker, ma l'accesso dipende da un byte letto speculativamente. viene quindi caricata una pagina in cache, ora so che ho rimosso tutte le pagine tranne quella appena caricata e posso fare il timing sulla mia stessa cache di quale di queste pagine è stata caricata.

### 2.9.1 inganno della branch predicion unit

l'unità per la predizione dei branch serve per aiutare nella decisione delle istruzioni di salto, quindi dire quale è il migliore valore di guess per una istruzione di salto. il predittore impara dal passato recente, che però dipende dalla mia applicazione: quindi possiamo fare del poisoning della bpu per fargli fare quello che vogliamo. eseguiamo una istruzione voluta molteplici volte per fargli capire qual è l'outcome della istruzione, dopo del tempo la bpu diventerà stabile e quindi l'outcome sarà sempre lo stesso. all'improvviso, cambiamo il flusso del codice, quindi verrà fatto un guess errato che era quello che volevamo, ma se il guess è errato il check verrà fatto dopo nella pipeline, intanto verrà fetchata l'istruzione in pipeline e se possiamo controllare le istruzioni, siamo in uno scenario simile al precedente.

qui serve uno step addizionale rispetto a meltdown perché va fatto il trainign della cpu

#### spectre v1

facciamo girare del codice un numero elevato di volte

```
1 if (x < array1_size){
2   y = array2[array1[x] * 4096];
3 }
```

il valore di x però viene usato per accedere una pagina in un probe array, come in meltdown. per far si che funzioni, dobbiamo essere sicuri che `array1_size` non sia in cache, perché vogliamo che il cache controller starti una transazione per portare in cache il valore. la bpu farà un guess sull'outcome dell'if, in modo che se lo facciamo spesso la bpu dirà sempre branch taken, non avendo il valore in cache aumentiamo il tempo per cui il guess fatto dalla bpu sarà vero o no. `array1[x]` è il byte di interesse,

ma nel momento in cui `x` non sta nei limiti, saltiamo da qualche parte nell'address space, questa cosa viene fatta dopo un certo tempo. l'attacco è molto più difficile da patchare: non avendo array1 in cache, non ci sono check che la cpu può fare in anticipo, mentre nel caso di meltdown il bit di accesso è sempre disponibile. in questo caso il dato non è in cache, il controllo viene appositamente dilazionato, l'attacco va a cambiare il corretto funzionamento della bpu e del funzionamento interno della cpu.

## spectre v2

variante di spectre, fin ora abbiamo visto attacchi per cui il codice per l'attacco era nel processo dell'attaccante. possiamo fare una variante in cui il codice non è nell'address space: cerchiamo una porzione di codice in un altro processo, un buon candidato è codice di livello kernel, o librerie condivise, programmi ebpf. scegliamo un **gadget** dall'address space della vittima, ovvero un insieme di istruzioni valide generate dal compilatore per essere eseguite in un certo ordine. quindi, cosa succede se vengono eseguite in un altro ordine, ovvero ad esempio eseguendole all'interno di una funzione, possiamo fare jump ovunque cambiando il valore del pc, quindi anche fuori dalla logica del mio programma.

dobbiamo poter ispezionare il codice dell'applicazione e saltare da qualche parte, quindi al gadget nell'address space della vittima. di nuovo, facciamo il poison della bpu, possiamo far girare la funzione corretta in modo da avere un certo comportamento, dopo un certo numero di guess cambiamo le pre-condizioni del gadget, quindi saltiamo direttamente al gadget, quindi il vero payload dell'attaccante è il gadget della vittima che è codice legittimo, semplicemente lo usiamo in maniera impropria, ad esempio saltando nel mezzo di una funzione.

esempi:

```
1 adc edi, dword ptr [ebx + edx + 13be13bdh]
2 adc dl, byte ptr[edi]
```

istruzioni di win10, accediamo alla memoria con queste due istruzioni. le istruzioni sono parte di una funzione più grande, vengono usate per calcolare qualcosa nel kernel, l'unica cosa che mi interessa è che usano quei due registri: se riesco a controllare quei registri e fare il jump a quelle istruzioni, posso controllare i valori. per controllare le istruzioni, dobbiamo assicurarci che la bpu faccia un guess sul contenuto dei valori. la funzione gira molteplici volte, in modo che la bpu associ alla prima istruzione un certo outcome, la bpu fa il check sul fatto che io possa o meno accedere alla memoria, facendo così la bpu farà semplicemente girare l'istruzione, facendo side effect in cache, e poi la cpu si renderà conto che magari va squasata, ma ormai possiamo misurare i side effects.

per sfruttare le istruzioni, possiamo impostare `edi` all'indirizzo di base di un certo probe array: vogliamo accedere ad `m`, quindi dobbiamo sottrarre i valori che vengono sommati nell'istruzione:  $ebx = m - edx - 13be13bdh$ . quindi stiamo controllando come usare `m`, la prima istruzione leggerà sempre il mio indirizzo di memoria `m`, il valore viene aggiunto al registro `edi`, ma `edi` sta puntando al mio probe array. la seconda istruzione carica nel probe array il valore `m`, quindi possiamo far girare qualunque codice del kernel, shared library etc.. in modo da accedere al mio probe array. il codice è legittimo, accediamo la memoria su una struttura dati controllata da me, il probe array, quindi accediamo a nostro piacimento ed il codice non è semplice da patchare.

ci sono le convenzioni delle chiamate a funzione: se chiamiamo delle chiamate a funzione, il caller deve assicurarsi che i valori siano salvati, se saltiamo nel mezzo di una funzione, il compilatore non si cura del valore perché da per scontato che il chiamante abbiamo salvato lo stato dei registri. il compilatore genera il codice pensando che questo verrà usato sempre rispettando le convenzioni delle chiamate, ma se si salta nel mezzo della funzione, si può eseguire del codice gadget a nostro piacimento.

### 2.9.2 mitigare i side channels

in generale, gli attacchi basati su timing sono complicati da patchare perché si usano i meccanismi interni della cache. quindi, dovremmo poter implementare delle operazioni idem-potenti che non diano leaks sul tempo. ma cercare di nascondere la differenza fra hit/miss della cache e per poter riuscire a liberarsi di questi problemi occorrerebbe non usare la cache che non ha senso.

si potrebbero applicare una strada più dura, quindi ad esempio restringere dei timer ad alta risoluzione come la rdtsc, ma spesso le operazioni sono necessarie. si potrebbero marcare determinate regioni di memoria come non raggiungibili, ma è molto challenging dal punto di vista hardware.

abbiamo detto che spesso si sfrutta aes come "vittima" per questi attacchi, qui si potrebbe implementarlo completamente in hardware, ma per gli altri algoritmi di cifratura? o se la cpu non ha questa possibilità?

altra possibilità è implementare gli algoritmi in modo che ai dati segreti non sia permesso di influenzare gli accessi in memoria. per gli attacchi basati sulle istruzioni transazionali, si può disabilitare tsx oppure buttare la ooo-pipeline, che però è performance-critical.

fin ora questi attacchi sono sulla l1, per patchare gli attacchi a questa cache si può disabilitare il smt (sim multi-threading), quindi in pratica per rendere il sistema sicuro consiste nel disabilitare tutto quello che è stato introdotto per essere più veloci.

#### **detection way**

l'idea dietro la rilevazione dei side channel: il side channel spesso porta la cache in stato noto in una fase di prepare, quindi c'è molto rumore sulla cache, vengono fatte molte operazioni. le cpu moderne (dai 90) hanno degli hpc (hardware performance counter): perf è uno user space tool che aiuta a capire se ci sono dei bottle-necks nell'applicazione, per farlo il tool usa dei registri counter che sono programmabili dal so:

- contare il numero di miss in cache l1
- contare il numero di flush nella l3

i conter sono efficienti perché collegati tramite fili direttamente ai counter hardware. quindi, se verifichiamo gli effetti più comuni sull'architettura della cache, possiamo creare dei profili di utilizzo dell'architettura di cache per scoprire se sono state fatte determinate operazioni sulla cache. il problema è che questo tipo di tool sono molto volatili e cambiano spesso, inoltre hanno dei bug perché non sono critici per l'esecuzione delle applicazioni.

i problemi per un approccio basato sulla mitigazione: la cache funziona bene perché è basata sul principio di località, se l'applicazione lavora con una mole di dati eccessiva, continuerà a richiedere dati dalla memoria ed a portarli in cache e quindi non possiamo essere al 100% sicuri che l'applicazione sia malevola, quindi non possiamo usare questo tipo di tecnica ad esempio per far carshare le applicazioni. potremmo, se c'è smt attivo, impedire di far girare un altro processo sulla stessa cpu. l'attacker può aggirare questo tipo di controllo facendo sleep di 1h (ad esempio) dopo il leak di ogni byte, per diminuire il bandwidth dell'attacco, quindi di nuovo non possiamo scoprire l'attaccante con sicurezza.

quindi in generale non è possibile mitigare i side channel attacks.

#### **mitigare le transient execution: kernel isolation**

solitamente, nella memoria abbiamo la parte alta dedicata al kernel ed un'altra dedicata allo user space. nella page table, ci sono i bit che dicono se si può o meno accedere all'indirizzo fisico, ma questo



controllo viene fatto dopo nella pipeline, quindi il modo per evitare che l'indirizzo sia caricato è andare a rimuovere l'indirizzo dalla page table. l'unico modo a livello software è impedire all'applicazione il mapping fra memoria virtuale e fisica, così la cpu fermerà subito l'esecuzione perché non ha l'indirizzo della memoria valido. l'unico modo è cambiare come il so viene mappato in memoria virtuale, ovvero usare la kernel page table isolation, tradizionalmente l'organizzazione era del kernel sopra lo spazio user, oggi cambia se siamo in kernel mode o user mode:

- in kernel mode le strutture sono le stesse
- in user mode, la gran parte del kernel space non è mappata: se cerchiamo di fare leaking del byte, non otteniamo nulla perché la memoria non è mappata su memoria fisica e quindi non ci saranno side effects in memoria.

quindi

- come facciamo ad applicare questo meccanismo, ovvero cambiare la visione velocemente: la traduzione funziona perché cr3 mantiene l'indirizzo della page directory. il so riserva una sola entry per il kernel level, se abbiamo due versioni differenti della page directory, in cui in una la entry del kernel è valida e nell'altro no, possiamo switchare le tabelle a seconda della mode con cui stiamo girando. per cui, ogni volta che passiamo a kernel mode, nel codice della `cpu_entry_area`, il sistema aggiorna in contenuto del cr3 per cambiare la first level page table usata dal firmware. c'è un componente hardware addizionale, perché fare la traduzione tra indirizzo fisico a virtuale, ma c'è il tlb dove potrei aver cachato le traduzioni degli indirizzi del kernel: quindi ogni volta che si cambia il contenuto del cr3 va flushato il tlb. ogni volta che si chiama il kernel, si flusha il tlb due volte perché si chiama il cr3 due volte, quindi per questo le performance delle applicazioni droppano tantissimo se si leggono/scrivono parecchi dati su/dal disco usando molte syscall.
- perché c'è una piccola parte di kernel mappata: ci sono le pagine tradotte da indirizzo virtuale a fisico per lo user space, se c'è un miss nel tlb occorre fare la traduzione. si deve avere la possibilità di gestire un hardware interrupt, servono le informazioni minime per gestirle ed inoltre serve il codice minimo per entrare in kernel mode, in modo da fare la rimappatura kernel space.

quindi la struct mantenuta è la `cpu_entry_area`, dobbiamo avere la global description table, più lo stack da usare quando cambiamo da user a kernel mode più altro.

se giriamo su un sistema multi-core, possiamo avere che su un core l'app gira in modo kernel e quindi c'è una mappatura della memoria, su un altro core ci sarà la configurazione con lo user mode mappato ed il kernel no. per questo, la struttura dati è per cpu-core, altrimenti lanciamo un attacco multi-thread con cui uno entra in kernel mode e l'altro fa partire meltdown.

il cr3 è quindi un registro per-cpu, per ogni processo il kernel mantiene due page tables, in una di queste non c'è la traduzione virtual to physical degli indirizzi kernel.

ogni volta che si invoca un syscall si vede questo codice, ad esempio per processori x86\_64:

```
1 /arch/x86/entry/entry_64.s:
2 sym_code_start(entry_syscall_64)
3 ...
4 switch_to_kernel_cr3 scratch_reg=%rsp
5 ...
6 switch_to_user_cr3 scratch_reg=%rdi
```

```

7
8  /arch/x86/entry/calling.h:
9  .macro switch_to_kernel_cr3 scratch_reg:req
10 mov %cr3, \scratch_reg
11 andq $(~pti_user_pgtable_and_pcid_mask), \scratch_reg
12 mov \scratch_reg, %cr3
13 .endm

```

la transizione avviene flipando un singolo bit poiché le due page table sono contigue. c'è il tlb, quindi abbiamo delle traduzioni virtual-to-physical cachate e quindi ogni volta che si scrive in cr3 l'architettura intel flusha immediatamente il tlb.

### mitigazione con retpoline

è una costruzione software, l'idea è di una tecnica che permetta di isolare i branch indiretti da istruzioni speculative, quindi si evita di fare esecuzione speculativa di qualunque cosa che possa fare leaking dei dati. quindi in uno scenario in cui la branch prediction unit, una istruzione che fa salto indiretto esegue un loop infinito, si basa sulla tecnica dei thunks, ovvero fare il delay di una computazione solo al momento in cui fosse necessario. le istruzioni che possono soffrire di attacchi spectre sono

- `jmp *%rax`
- `call *%rax`

rax può contenere l'istruzione corretta oppure no, quindi si fa il fool della branch prediction in modo che si salti a codice che non dovrebbe eseguire.

vediamo quindi un esempio

```

1 jmp *%r11 jmp retpoline_r11_trampoline
2 call *%r11 call retpoline_r11_trampoline
3
4 retpoline_r11_trampoline:
5     call set_up_target
6 capture_spec:
7     pause
8     jmp capture_spec
9 set_up_target:
10    mov %r11, (%rsp)
11    ret

```

la `set_up_target` scrive sul top dello stack l'indirizzo di `%r11`, quindi mettiamo un certo target in `r11` che è quello che vogliamo. all'inizio della funzione modifichiamo il top dello stack con l'indirizzo contenuto nel registro. lo stack di return nella cpu deciderà l'istruzione da mandare in pipeline, che è la pause che dirà al processore che siamo in uno stato di looping. questo va avanti finché non siamo davvero in grado di determinare il contenuto di `%rsp`, a quel punto si può flushare dalla pipeline il codice della retpoline.

in questo modo un attaccante non può fare leaks, poiché ogni outcome del branch predictor viene controllato.


il compilatore deve generare un thunk per ogni registro general purpose per cui può accadere che ci sia un salto indiretto

## prevenire branch poisoning

ci sono delle capabilities delle cpu moderne per prevenire il poisoning del branch predictor:

- `ibrs`, istruzioni specifiche che permettono di entrare in una modalità per cui la bpu non viene influenzata dalle predizioni false. ci sono quindi delle bpe differenti a seconda della modalità
- `stibp`: ci sono molteplici bpe, quindi con hyperthread su un singolo core si usano bpu diverse per i diversi thread virtuali. non si possono quindi fare attacchi sullo stesso core, si duplica l'hardware ma aumenta la sicurezza
- `ibpb`: include nuove istruzioni a livello isa che creano una barriera di esecuzione per la bpu. quindi, ogni cosa che la bpe ha imparato prima della barriera non è più valido, flushando le informazioni imparate dalla bpu.

quindi, in pratica, applicando tutte le patch si torna ad una i386

 immagini/from\_i7\_to\_i386.png

a seconda dell'applicazione che si sta girando, occorre applicare alcune patch piuttosto che altre. non c'è un sistema sicuro, occorre mettere la sbarra di sicurezza al livello più adatto al caso specifico.

## esempio pratico: spectre + meltdown

vediamo il file del kernel linux `/proc/version`, che dà informazioni sul kernel di linux. l'informazione è mantenuta dal kernel, si contatta una api che copia in user space un'informazione da stampare, quindi c'è una variabile nel kernel space che contiene una informazione di stringa.

se lanciamo il programma, si vendono delle stringhe di formato. nel codice c'è hardcoded l'indirizzo della variabile nel kernel, c'è un file che dà gli indirizzi dei file nel kernel, ogni volta che si re-starta il sistema gli indirizzi cambiano, inoltre passando il cat senza root non si vede nulla (tutti i byte degli indirizzi sono 0 per questioni di sicurezza).

invochiamo una operazione per aprire il file in memoria, quindi vogliamo spostare in cache i dati da accedere. a questo punto c'è l'attacco: forziamo la vittima, in questo caso il kernel, a caricare i suoi dati in cache per accedervi.

si usa un array per calcolare un indirizzo di memoria come avviene in spectre, si allena quindi il branch predictor per fargli credere che sia tutto giusto. viene continuamente fatto un bound check, ovvero un tentativo di controllo sul probe array per ingannare l'unità di predizione dei branch in modo che questa creda che l'istruzione possa essere eseguita anche se non è così.

ora c'è un meltdown attack

- si prende il tempo di accesso alla cache
- se il tempo speso nella lettura del byte va oltre la threshold si può restituire il valore del byte misurato

perché si possono mescolare le due tecniche: meltdown è solo una variante di spectre, c'è un miss-uso implicito del branch predictor. quindi, la maggior parte degli attacchi che ci sono oggi sono varianti di spectre, quindi si possono usare insieme perché meltdown ne è solo una variante. è necessario disattivare la protezione


## 2.10 recap sulle dram - funzionamento della ram

la memoria funziona in termini di scrittura/lettura di celle: ogni banco è organizzato in piccole celle di memoria, ci sono 3 segnali che vi operano

- select: seleziona una cella di memoria
- control: scelta dell'operazione da fare
- data in/sense (??)

con i seguenti schemi:

in una ram, abbiamo un piccolo capacitore, che è carico / scarico: se carico porta valore 1, altrimenti

immagini/accesso\_cella\_memoria.png

0.

c'è un transistor controllore che dà accesso al bit mantenuto nel capacitore. quindi, se si vuole scrivere occorre dare corrente / rimuovere corrente, se si legge si cerca di scaricare il capacitore per vedere se era carico o no. il problema è che il livello di carica del capacitore dà il valore del bit, ma il capacitore si scarica nel tempo, quindi serve un modo per ricaricarlo per tenere il bit ad uno "vivo". ogni byte è fatto da 8 schemi come questo, ogni volta che si legge una linea occorre fare il write back di quella linea.

è interessante vedere il funzionamento per due motivi

- lo scaricamento del capacitore è una esponenziale negativa, misurata con una costante (ricorda la costante di decadimento)
- quindi, a seconda di dove si fa il sensing del capacitore non è detto che si riesca a dire con certezza se c'è un 1 o uno 0
- in base ad una threshold, si ricarica il capacitore

quindi, il memory controller legge periodicamente il contenuto della memoria per poi riscriverlo, anche quando non ha bisogno solo perché una lettura ricarica il capacitore.

tipicamente, ogni 64ms il memory controller rilegge le linee per evitare di perdere il contenuto del condensatore, ogni volta che si fa un refresh il contenuto della memoria non è disponibile e quindi occorre aspettare. questo è il motivo per cui la ram è più lenta del processore, ci sono diversi modi per implementare il refresh

### 2.10.1 refresh della ram

il refresh distribuito fa un refresh distribuito nel tempo dei 64 ms in modo da avere alcune aree di memoria disponibili mentre si fa il refresh, ma più si incrementa la grandezza della ram si spende più tempo per fare il refresh, quindi a 16 gb spendiamo circa il 20% del tempo operativo per refreshare la ram.

quindi questo spiega ancora il perché le ram sono più lente delle cpu.

## 2.11 primitive di lettura

siamo in grado di scrivere dei bit random in memoria. il problema è la densità dei chip: i condensatori sono molto molto vicini, organizzati in matrice e che vengono caricati / scaricati ed ogni volta che si legge una riga vanno ricaricati i condensatori. ogni volta che si legge quindi, la corrente che scorre c'è fluttuazione del voltaggio, che genera un emf indotto che si espande fra le diverse linee. quindi, questo genera effetti nei capacitori vicini, si possono quindi

- aumentare lo scaricamento del capacitore
- ricaricare

quindi, se si scarica prima, la deadline è più stretta ed il memory controller legge uno 0 piuttosto che un 1. se si ricarica invece, si riesce ad andare oltre la threshold e si può ricaricare il capacitore. quindi fare letture continue permette di flippare bit, nessuno se ne accorgerà e la cosa da fare è trovare delle righe adiacenti in memoria fisica. ma siccome l'organizzazione dei banchi di memoria è nota, vogliamo fare qualcosa del genere

```

1  code1a:
2      mov (x), %eax legge dall'indirizzo x
3      mov (y), %ebx
4      clflush (x)
5      clflush (y)
6      mfence
7      jmp code1a

```


questo è il blocco fondamentale di uno **rowhammer attack**, è semplice generarlo perché c'è duplicazione nel so in quanto diverse pagine di memoria sono condivise fra le applicazioni, la maggior parte degli attacchi sfruttano il js dei browser in quanto sono utilizzate molto le pagine condivise. quindi semplicemente leggendo pagine di memoria e conoscendo l'organizzazione della memoria, possiamo capire dove sono delle linee di memoria vicine.

una pagina interessante che può essere sharata è codice di libreria crittografica, quindi possiamo fare il flip di un bit di una chiave

### 2.11.1 mitigazioni possibili

le possibili mitigazioni

- inserire dei crc nel codice, ma gli attacchi possono flippare più bit e quindi non è detto che il crc riesca a correggere i dati
- ridurre l'intervallo dei 64 ms, ma già così il 64% dell'uso della memoria di 64 gb viene speso per il refresh e quindi è un prezzo troppo alto da pagare
- detection: creiamo molto rumore sulla cache, in quanto si fa load/flush dalla cache, ma una volta che è stato scoperta la posizione in memoria si genera meno rumore
- pseudo target row refresh: tracciamo l'accesso alle linee di memoria, così che il memory controller capisca che una linea di memoria è sotto attacco e cominci a refresharla più spesso. è però un tipo di approccio di "security by obscurity", quindi è fatto internamente dal memory controller ed inoltre è comunque possibile fare il lavoro su più di una linea di memoria adiacente.

 immagini/dram\_mc.png

fare un attacco pratico non è semplice, occorre capire quali blocchi di memoria sono più soggetti ad essere flippati. poi, si fa girare del codice shared e si aspetta che il so mappi il codice su una zona di memoria su cui si sa di poter fare bit flipping e poi si fa l'attacco. abbiamo tanto tempo per fare l'attacco, anche settimane, per capire l'organizzazione fisica della memoria, perché può girare anche in vm.

## 2.12 memory performance attacks

### 2.12.1 architettura di memoria

l'implementazione di un memory controller non è così semplice, è datata e non è semplice da re-ingegnerizzare.

ora c'è il multi-core, quindi è stato necessario un layer in più, lo scheduler. il ruolo dello scheduler è implementare una specie di caching a livello di memory controller

- il mc deve fare sempre il refresh, ogni volta che si legge si spreca tempo. si vuole quindi evitare di fare operazioni di memoria se non è necessaria
- c'è un row buffer nel mc: ogni volta che si legge una riga, questa viene cachata e se vi si ri-accede la si trova. può esserci hit o miss, ma pensando alla località è verosimile leggere tutti i byte della stessa riga

si cerca prima nel row buffer, se c'è miss si fa write back della riga e poi si fa l'operazione di lettura. questa è l'organizzazione di come funziona la memoria in un'architettura multi-core moderna

per avere un accesso fair alla memoria fra tutti i cores, tutto dipende dal protocollo usato all'interno del dram bus scheduler.

l'algoritmo è fr-fcfs

- il bank scheduler prende riordina le richieste in base allo stato corrente del row buffer, inoltre implementa una policy fcfs in cui thread che generano più richieste hanno priorità maggiore, perché mettono più richieste in coda
- bus scheduler

se abbiamo un thread che fa molte richieste con alta località nel row buffer, le sue richieste supereranno quelle degli altri threads. indipendentemente dal numero di richieste degli altri thread, sarà sempre lui ad avere la priorità rispetto a tutti gli altri per come funziona il protocollo. quindi, se un attacker ha diversi thread che fanno molteplici richieste sullo stessa linea di memoria dello stesso banco di memoria, tutte le richieste vengono prese dallo stesso memory bank e per questo motivo sfruttiamo la località.

per fare questo memory hogging attack, occorre fare flush cache per poter sempre leggere dalla memoria.

# Capitolo 3

## os security principles

### 3.1 principi di sicurezza

consideriamo due principi di sicurezza base

- il sistema deve essere usato solo da utenti legittimi
- l'accesso è permesso in base ad un'autorizzazione, data dal sysadmin

il primo punto sembra facile, ma come un attacker possiamo fare un dos per rendere il sistema non usabile anche ad utenti legittimi.


l'idea è concentrarsi sul primo punto

### 3.2 kernel user space api - meccanismo delle syscall

il kernel non vuole lasciar fare tutto all'utente, quindi si accede al kernel mediante syscall. il kernel gira sull'intel cpu in protected mode, possiamo sfruttare 4 ring di protezione, più basso il ring maggiore il privilegio.

all'inizio, a ring 0 c'è il kernel, a 1-2 i device hardware, a 3 lo user space. girando a livello 3 non si possono usare diverse operazioni privilegiate ma bastano per poter implementare sicurezza a livello kernel. in so comuni si usano i livelli 0 e 3, diverso se si installano hypervisor per la virtualizzazione. ad un certo punto dell'esecuzione, vorremmo cambiare ring, ad esempio passando da user space a kernel space etc... in ogni momento si può de-privilegiare l'utente in intel, ma l'inverso va controllato e possibilmente non permesso.

per accedere ad un ring più privilegiato si passa per un gate, come mostrato in figura:

 immagini/gates.png

da user space non si salta alla funzione kernel, si passa per una routine che è registrata per un security gate. questo si ottiene con i registri segmento, come ss,bs etc... questi non mantengono più numeri grezzi bensì descrittori di tabelle, che dicono quali sono le capability di accesso ad una area di codice.

ogni volta che si vuole passare per un gate si sovrascrive il contenuto del registro, ma prima di farlo il firmware fa dei controlli di sicurezza: ci sono due campi coinvolti

- si legge il campo cpl nel cs per sapere il ring corrente
- nella tabella dei descrittori, a cui si accede con l'indice dei registri, c'è il dpl che dice qual è il livello di privilegio a cui quel ring può essere acceduto

da user space siamo a  $cpl = 3$  e possiamo cambiare ring se il  $dpl = cpl$ . se vogliamo invece abbassare il privilegio possiamo semplicemente sovrascrivere il registro segmento.

### 3.2.1 gate

un gate è un segment descriptor, ce ne sono diversi tipi che descrivono il motivo per cui si cerca di passare per un security gate.

ce ne sono di diverso tipo

- call-gate, non più usati (?)
- interrupt-gate
- trap-gate: interrupt software asincrono
- task-gate

un gate, per poter fare una transizione dei privilegi, su architetture tradizionali, si basa su un trap-gate descriptor: si genera una trap sincrona dal software che triggerà l'esecuzione di un trap-descriptor. la tabella in memoria che mantiene i trap-gate descriptor è la idt, puntata dal registro idtr.

abbiamo quindi il seguente schema

immagini/idt\_gdt.png

l'interrupt handler è una routine, il suo address è specificato nel campo offset del sd.

l'entry ha anche un selettore, che specifica un sd nell gdt; entrambe le tabelle sono popolate a startup time dal so.


non si usano i call-gate descriptor, perché guardando lo schema vogliamo chiamare una funzione kernel dallo user space, poiché il descrittore andrebbe mantenuto nella idt che permette solo di mantenere 256 entry per costruzione, quindi ci sarebbero meno campi possibili per le syscall kernel.

### 3.2.2 syscall

si basano su software trap che vada a targettare una specifica entry dell'idt, l'idea è che quando gestiamo gli interrupt, ad un certo punto va scelta l'entry dell'idt da associare con la ricezione dello specifico interrupt. in linux, l'istruzione macchina corrispondente alla software trap è `0x80` (`0x2e` in windows), il codice che è nell 'interrupt handler è un preambolo che determina quale syscall va chiamata, dopo essere transitato in kernel mode.

è possibile quindi scrivere in un registro segmento solo se il  $dpl \geq$  del cpl attuale, un gate è un'eccezione alla regola: il dpl in questo caso è il livello di privilegio a cui si può girare. vediamo il seguente esempio:



 immagini/cpl\_change.png


se cerchiamo di sovrascrivere il contenuto del dpl, viene fatto il check mostrato, per evitare la transizione da ring 3 a ring 0. ma se viene generato un interrupt, come una trap, il check è diverso e quindi alla fine possiamo scrivere nel gate descriptor 3 e quindi il gate diviene attraversabile se giriamo a livello 3. c'è un segment selector che prende il segment descriptor trova un dpl pari a 0: un gate dice quindi qual è il privilegio minimo per cui si può transitare ad un livello di privilegio maggiore.

### 3.2.3 syscall table

agglomeriamo in un unico handler diverse syscall, che è il syscall handler che controlla qual è la syscall da accedere. questo è possibile controllando la system call table, che è una tabella di function pointer a funzioni di livello kernel.

quando si gira in kernel mode quindi, si prende dalla syscall table l'indirizzo della funzione da chiamare e quindi possiamo fare tutti i controlli di sicurezza nel syscall dispatcher, è possibile quindi semplificare la manutenibilità dell'os.

il dispatcher ha bisogno dell'offset per identificare la syscall corretta, si usa un intero che è il syscall number. la syscall si attiva con un meccanismo indiretto, quindi si può accedere su un attacco tipo spectre, perché si accede con un intero quindi si può trainare il branch predictor per far sì che poi si salti male e quindi molti os moderni usano le retpoline per fare la chiamata. lo schema è quindi il seguente

 immagini/disp\_scheme.png

#### dispatcher

attiviamo il dispatcher, troviamo sullo stack l'interrupt frame, che è come lo stack frame ma modificato dal firmware per indicare che questo è stato modificato da una trap. il dispatcher copia i contenuti dei registri sullo stack, tra cui ad esempio gli argomenti della syscall, poi si fa l'invocazione della syscall e quindi sotto lo stack della funzione corrente si può trovare uno snapshot completo della cpu. questo ha il ruolo fondamentale di tornare allo user space con l'effetto di ritrovare tutti i valori dei registri come se non fossero stati manipolati.

la struttura dati è in linux `pt_regs`, dove si trova tutto il contenuto sotto lo stack frame della syscall chiamata.

sono stati introdotti dei meccanismi di fast syscall per evitare tutto questo overhead


- su architetture a 32 bit vengono usati dei model-specific registers aggiuntivi, si usa il meccanismo della `sysenter`. un insieme di registri vengono usati per mantenere il valore di cs quando si transita kernel mode + vari altri registri
- `sysexit` è la controparte della precedente

non occorre più usare l'organizzazione vista prima, ma scrivere a startup i valori nei 4 registri per transitare in kernel mode e questo può essere fatto solo una volta a startup time.

l'implementazione è molto patchy, una volta che amd ha sviluppato l'architettura a 64 bit è stata introdotta la coppia `syscall/sysret` che sono basate di nuovo su registri model-specific.

allo startup del kernel, nella funzione di `syscall_init(void)` si trovano le istruzioni per scrivere i valori del code segment sia per user che kernel space. l'entry point per il syscall dispatcher viene scritto nell'ultima istruzione.

intel ha un approccio backward compatible, quindi entrambe gli approcci sono validi, quindi ad esempio per raggiungere la syscall di read abbiamo diversi execution flows

 immagini/syscall\_paths.png

### 3.3 user identification

un altro aspetto fondamentale, che dà la possibilità di identificare chi fa operazioni su quale file. questo viene fatto con due file:

- `/etc/passwd`
- `/etc/shadow`

sono vecchi db di tuple in cui si trovano le seguenti informazioni

- `/etc/passwd`: `username:passwd:gid:full_name:directory:shell`. la parte della password è critica perché si potrebbe copiare il file e cercare di fare bruteforcing di tutte le password degli utenti. con lo shadowing si rimpiazza il campo col placeholder `x`
- lo shadow file ha sempre delle tuple: `username:passwd:ult:can:must:note:exp:disab:reserved` dove:
  - `username` è l'utente;
  - `passwd` è la password cifrata;
  - `ult` sono i giorni trascorsi dal 1/1/1970 dall'ultimo aggiornamento della password;
  - `can` è l'intervallo di giorni dopo cui è possibile cambiare la password;
  - `must` è l'intervallo di giorni dopo cui bisogna cambiare la password;
  - `note` è l'intervallo di giorni dopo cui all'utente viene richiesto di cambiare password;
  - `exp` è l'intervallo di giorni dopo cui viene disabilitato l'account utente se la password;
  - `disab` giorni dall'1/1/1970 dopo cui l'account verrà disabilitato
  - `reserved` campo non usato.

la cosa importante è come il kernel guarda gli utenti: per il kernel la cosa importante è l'uid, che è un identificativo con cui il kernel fa i check di sicurezza, lo stesso vale per il gid

### 3.3.1 uid/gid in unix

il kernel associa ad ogni processo running diversi uid:

- l'uid reale è un intero che dice chi è l'utente reale che ha lanciato il programma
- il secondo uid è l'uid effettivo in termini di quale utente puoi diventare, quindi ad un certo punto si può cambiare l'utente con cui si gira
- il 3° è il saved uid, quindi chi si può nuovamente ridiventare, ad esempio di tornare all'utente originale

lo stesso vale per i gid.

ci sono delle syscall apposite per cambiare utente: l'utente root è sempre associato a uid pari a 0, abbiamo le syscall `setuid()/seteuid()` che possono essere invocate solo da applicazioni che girano con privilegi di root. se volgiamo scoprire i valori di uid e euid ci sono le `get`, chiamabili senza privilegi.

la `setuid()` non è reversibile, quindi di sovrascrivono tutti e 3 i valori, mentre la `seteuid()` lo permette. quindi se un'applicazione ha l'euid impostato a 0 permette di divenire temporaneamente root.

introduciamo come lavorano su `sudo`: vengono mantenute le informazioni dell'euid. se cerchiamo le informazioni legate al binario `sudo` c'è un bit aggiuntivo che dice che una volta lanciato l'applicativo l'euid va mantenuto dal `so` (è il bit "s") e quindi real e saved uid sono dell'account che ha chiamato ma l'euid è 0, quindi root ed a questo punto si possono chiamare le syscall viste prima. si può anche chiamare `su` con `sudo`, per cambiar utente e `su` richiederà la password per poter verificare nello `shadow file` se è corretta.

quindi avere un binario con `s` attivo è un grande problema se è possibile trovare un bug nell'applicazione perché si può fare di tutto.

### rottura del kernel

installiamo un modulo nel kernel:

- nella funzione di inizializzazione, possiamo fare cosa volgiamo perché siamo a ring 0. la prima cosa è prendere l'indirizzo della syscall table, per scandire la memoria un func pointer alla volta a partire da un certo indirizzo e troviamo la syscall table nel momento in cui troviamo l'indirizzo di una syscall. si può fare, in kernel mode, l'accesso al file `kallsyms`, ma è stato disabilitato.  
quindi si può fare profiling della funzione, si inserisce una funzione di callback (hook) tramite `kprobe` per poter leggere l'indirizzo della `kallsyms_lookup_name_t` e poterlo restituire.  
quindi, ogni volta che si incrementa la sicurezza si perde qualcosa, in questo caso si perderebbe la possibilità di fare kernel profiling
- per poter scrivere la nuova syscall nella syscall table si accede al `cr0`, dove c'è un bit che dice se occorre fare check della protezione della memoria, si flippa quindi il bit di protezione stando attenti a fare memory reordering delle operazioni.
- ri-scrittura della `kill` per gestire determinati segnali
- per nascondere un processo, basta ricordare che in linux tutto è un file, quindi si può nascondere anche un processo.

### 3.3.2 su e sudo

come è stato possibile riuscire nel rompere linux: il problema era che il modulo è stato montato come root, quindi la sicurezza di un sistema passa per l'account amministratore, c'è l'equivalente per windows e quindi il problema non sta nella sicurezza del so in sé ma nel fatto che l'account di amministratore possa fare qualsiasi cosa. non c'è modo di rendere un so sicuro, a meno di cambiare il modo in cui gli utenti vi interagiscono: vale il **principio dei privilegi minimi**: in ogni layer del sistema, qualcuno (utenti o applicazioni running) deve poter accedere solo alle informazioni e risorse necessarie per i suoi obiettivi legittimi.

quindi, servono solo i privilegi obbligatori e necessari per le proprie operazioni, quindi ad esempio se un'applicazione ha bisogno di privilegi di amministratore, deve ricevere solo i minimi indispensabili. è fondamentale sia per utenti che applicazioni

- gli utenti possono commettere errori;
- le applicazioni possono avere dei bug.

ci sono diversi benefici

- stabilità del sistema: è più facile testare le possibili azioni delle applicazioni
- migliora la sicurezza
- migliora la facilità di deployment per le applicazioni, quindi è anche fondamentale per il software engineering

### 3.3.3 access control

tutto questo si relaziona con l'idea di access control: i so moderni permettono di fare controllo sull'accesso a grana fine, è la capacità del so di controllare la possibilità per qualcuno di fare operazioni su un oggetto, quindi è possibile sfruttare i controlli di accesso per ridurre le capacità delle applicazioni. l'obiettivo non è ridurre le capacità degli utenti nel sistema, ma di limitare i danni causati da errori di utenti ma anche da utenti malevoli, codice malevolo etc... il mindset corretto è che qualcosa vada male, o per via di attacchi esterni o per via di un errore interno (esempio: downtime di fb e ig, hanno provato a lanciare un test sui data center che ha triggerato una policy di sicurezza)

per capire come migliorare la sicurezza in questo contesto, dobbiamo categorizzare le policy di sicurezza in due categorie: le policy descrivono cosa l'utente può fare e cosa no, ma anche chi gestisce le policy. distinguiamo

- discretionary ac: gli utenti ordinari sono coinvolti nella sicurezza nel sistema (come al pop-up window di windows per installare come admin). si possono cambiare permessi di accesso ai file etc...
- mandatory ac: l'opposto del dac, se siamo utenti regolari non possiamo cambiare le security policy, c'è un amministratore che deve gestire le policy ma quest'ultimo non è un utente del sistema, quindi l'applicazione o il sistema etc... viene gestito da qualcun altro. quindi nessuno può di sua volontà tamper col sistema.

è interessante che il più degli os off the shelf implementano per default dac, ci sono alcune capability che permettono di implementare una mac policy.

le security policy dicono sol chi può gestire la sp, ma occorre capire come implementarle, quindi come avere un'ac a grana fine:

- system wide
- directory
- file
- action on file
- byte offset of file

più la grana è fine, più la policy è complessa quindi in base all'applicazione si sceglie un livello di granularità ed in base all'azienda si sceglie la sp più adatta: se la sp è molto "forte", si può avere un downtime dell'applicazione molto alto (vedi sempre l'esempio di fb e ig) e quindi ogni sp ha un costo che può essere anche elevato ed è un lavoro a parte sceglierle.

### posix acl

le ac del posix standard sono le acl (access control list), quindi i file system implementano degli attributi estesi oltre agli ottetti rwe, si possono specificare delle access policy aggiuntive, rimane comunque uno schema di controllo discretionary, l'utente che è owner del file può cambiare le acl. non risolve il problema dell'installare un rootkit, qui si parla di chi può accedere al file del sistema. ogni volta che si vuole accedere ad un certo file o directory che è salvato su un ext file system, il so fa dei check: ci sono policy che descrivono cosa un utente/gruppo/altri possono fare

- far girare un algoritmo che controlla qual è l'entry della acl che dice se un utente può fare una certa operazione o può non farla: un utente lancia un programma con un certo euid, in base a quello viene controllato se il processo è l'owner o meno.
- se è l'owner, si controlla l'entry owner, altrimenti si controlla la entry user;
- se nessuno user matcha, il kernel controlla il gid e vede se si possono usare i permessi di gruppo, altrimenti si cerca una group entry nell'acl
- altrimenti si usa la entry other

questo comunque non risolve il problema

## 3.4 capabilities

se si guarda il codice del kernel, ci sono dei controlli sulle **capabilities**: unix tradizionale distingue due categorie di processi

- processi privilegiati, che girano con euid = 0
- processi non privilegiati

girando come utente privilegiato, ogni controllo di capability viene saltato, altrimenti si fanno i controlli per verificare le policy.

quindi come è possibile far girare una applicazione come root evitando che questo possa fare qualunque cosa: le capabilities possono essere usate per dividere i permessi di root in diversi domini, si possono avere dei processi che girano come root che possono configurare device di rete ma non cambiare i permessi sul fs, quindi l'obiettivo è quello di alzare la sicurezza del sistema. lo standard per

descrivere le capability è posix.1e, ma lo standard è stato ritirato perché le aziende non si mettevano d'accordo. linux comunque implementa questo standard: ci sono diverse capabilities descritte con macro che iniziano con `cap_` e definiscono bit tenuti in una bitmask che decidono cosa un processo può fare. le capabilities sono associate anche con delle capabilities salvate in un file nel file system. esempi

- `cap_chown`: permettere di cambiare ownership dei file, sovrascrive le dac,
- `cap_kill`
- `cap_net_raw`: per poter usare una socket di lvl 3;
- `cap_sys_nice`
- `cap_sys_time`
- `cap_sys_admin`: il "nuovo" root, c'è stato un grande effort nel kernel community per rimuovere questa capability

### 3.4.1 implementazione delle capabilities

l'idea è che stiamo riempiendo il kernel code con diversi check sulle capabilities che dicono quali azioni possono essere fatte dai processi quando girano, quindi implementarle propriamente non è semplice, vanno controllati tutti i possibili execution flows su una specifica risorsa per essere sicuri che la capability sia rispettata.

il vfs è complicato, le funzioni interne del kernel sono differenziate dal prefisso `vfs`, quindi sono funzioni chiamabili dal vfs. la seconda implicazione è che non c'è un'implementazione stabile, quindi in base ai cambi del kernel non è detto che l'implementazione delle capabilities le funzioni usate, ad esempio, siano disponibili nelle diverse release del kernel.

il so deve permettere quindi ad un thread di cambiare il proprio capability set, in modo da poterle cambiare o recuperare.

ci sono due syscall:

- `capget`
- `capset`

per verificare le capabilities da cambiare / quelle che si hanno. può diventare complicato gestirle, c'è una libreria `libcap` che fornisce un wrapper di alto livello per interagire con le syscall: questo è fondamentale perché se viene implementata una syscall, è un problema della libreria re-implementare la api e non dello user space.

inoltre, dobbiamo essere in grado di attaccare le capabilities ai file eseguibili, così come avveniva con lo sticky bit, in modo che il processo guadagni le capabilities quando il file viene eseguito.

le syscall permettono ad un thread di gestire le capabilities, quindi due thread differenti della stessa applicazione possono girare con diverse capabilities, sono quindi salvate da qualche parte nel tcb (task struct). ogni thread linux ha diversi capability sets

- *permitted*: vogliamo che dei thread, tramite un supporto del fs, abbia un insieme di capabilities ridotto, magari partire anche senza e possa aggiungere le capabilities presenti nel set permitted;

- *inheritable*: insieme di capabilities preservate una volta che un processo privilegiato esegue la `execve()`, possiamo quindi avere un processo con diverse capabilities nel set permitted ma 0 nel inheritable.
- *effective* slides
- *ambient*: l'insieme delle capabilities che vengono preservate dopo una `execve()` di un programma non privilegiato.

ogni processo figlio creato via `fork()` eredita la copia dei capability sets del processo genitore.

le altre capabilities sono quelle dei file, che vengono salvate negli extended attributes, gli stessi usati per le acl. se introduciamo le file capabilities, dobbiamo pensare a cosa accade se un'applicazione gira con delle capabilities ed usa dei file che hanno diverse file capabilities.

le file capabilities con le thread capabilities determinano le capabilities finali di un thread: magari abbiamo la `cap_sys_admin` ed eseguiamo in un file che non lo ha o viceversa: in generale le file capabilities appartengono a due insiemi diversi

- *permitted*: le capabilities di questo insieme sono permesse automaticamente ad un thread, indipendentemente dalle capabilities ereditabili dal thread
- *inheritable*: il set viene messo in and con il set inheritable del thread per determinare quale capability ereditabile vengono attivate nel insieme permitted dopo la `execve()`

occorre anche mimare lo sticky bit effective, che è il flag *effective* in modo da mimare il vecchio effetto di `sudo` (vedi slides).

ci sono diversi check fatti dal so quando avviene una `execve()`

- $p'(\text{ambient}) = (\text{il file è privilegiato}) ? 0 : p(\text{ambient});$
- $p'(\text{permitted}) = (p(\text{inheritable}) \& f(\text{inheritable})) \mid (f(\text{permitted}) \& \text{cap\_bset}) \mid p(\text{ambient});$
- $p'(\text{effective}) = f(\text{effective}) ? p'(\text{effective})$  slides

### capabilities bounding set

è per thread e permette di limitare ulteriormente le capabilities che vengono date ad un nuovo processo, sono utilizzate nei seguenti modi: durante una `execve()`, l'insieme bounding delle capabilities è messo in and con il file delle capabilities dell'insieme permitted ed il risultato dell'operazione è assegnato all'insieme delle capabilities permitted del thread.

il bounding set pone quindi un limite sulle capabilities permitted che potrebbero essere garantite da un file eseguibile. il bounding set agisce come superset di limitazione per le capabilities che un thread può aggiungere al suo insieme inheritable usando `capset()`.


questo vuol dire che se una capability particolare non è nel bounding set, non può essere aggiunta nell'inheritable set anche se era nel set permitted.

## 3.5 security modules

meccanismo del kernel linux per permettere a diverse aziende / sysadmin di implementare diversi schemi di sicurezza. non è stato implementato un mac, è stato difficile trovare un accordo su qualche schema usare, quindi è stato realizzato un framework di sicurezza ed è possibile caricare alcuni moduli di mac scheme

- selinux;
- smack;
- apparmor;
- tomoyo

linux ha quindi offerto un supporto per mac con un minimo overhead, quindi si può implementare il modulo che si vuole: un esempio ad alto livello di accesso al file è il seguente:

 immagini/ref\_monitor.png

l'lsn hook è un function pointer, che andrà a chiamare il modulo kernel che implementa la policy engine che farà il check ed alla fine l'outcome sarà si/no ed in base alla risposta si accede all'inode. se viene combinato questo con una disabilitazione dell'account di root allora viene implementato un vero mac, gli lsn hook sono restrittivi, c'è una sola struttura nel kernel che mantiene questi function pointer che è la `security_ops`, caricata a boot time e che può essere aggiornata con la syscall `register_security()` (c'è al duale per de-registrare), il kernel rimpiazza gli hooks con quelli salvati nella struttura passata come input ed è possibile caricare un solo modulo di sicurezza alla volta. questa è l'implementazione interna della `register_security`

```

1  int register_security(struct security_operations *ops)
2  {
3      if (verify(ops)) {
4          printk(kern_debug "%s could not verify "
5                 "security_operations structure.\n", __func__);
6          return -EINVAL;
7      }
8      if (security_ops != &default_security_ops)
9          return -EINVAL;
10     security_ops = ops;
11     return 0;
12 }
```

le security options sono function pointers, ma è possibile volere delle informazioni aggiuntive da poter usare e ci sono dei pointer generici che le security operations possono istanziare e gestire come vogliono, e sono presenti nelle seguenti struct

- task\_struct
- linux\_binprm
- super\_block
- inode
- file
- sk\_buff: singolo pacchetto di rete



- `net_device`
- `kern_ipc_perm`
- `msg_msg`: messaggio individuale in una message queue.

la struct `security_operations` è un grosso insieme di function pointers:

```

1  struct security_operations {
2      ...
3      int (*ptrace) (struct task_struct *parent, struct task_struct *child);
4      ...
5      int (*inode_setattr) (struct dentry *dentry, struct iattr *attr);
6      ...
7  };

```

ad esempio, prima di attaccarsi con `ptrace` ad un certo task verrà fatto un check, oppure controllare se qualcuno a è permesso cambiare qualcosa in uno specifico file.

i moduli di sicurezza linux possono essere molto complicati da implementare, per questo i dev kernel non li fanno per conto loro ed offrono un framework generico basato su function pointers.

per registrare un modulo di sicurezza si possono seguire questi passi:

```

1  my_inode_setattr(struct dentry *dentry, struct iattr *iattr)
2  {...}
3  static struct security_operations my_security_ops = {
4      inode_setattr = my_inode_setattr,
5  };

```

la logica di controllo starà nella `my_inode...`

ci sono quindi 5 categorie di security hooks

- *task hooks*
- *program loading hooks*: decidere quale programma può essere lanciato
- *interprocess memory hooks*: se due processi possono condividere un segmento di memoria
- *ipc hooks*
- *filesystem hooks*: possiamo decidere che un sysadmin possa cambiare i permessi di un file sono in una specifica finestra temporale
- *network hooks*: forniscono un interfaccia per gestire sia i device che gli oggetti di rete

il seguente è un esempio di hooks per il file system preso dal kernel legacy 2.4:

```

1  int vfs_mkdir(struct inode *dir, struct dentry *dentry, int mode)
2  {
3      int error;
4      down(&dir->i_zombie);
5      error = may_create(dir, dentry);
6      if (error)
7          goto exit_lock;
8      error = -eperm;
9      if (!dir->i_op || !dir->i_op->mkdir)

```

```

10     goto exit_lock;
11
12     mode &= (s_irwxugo | s_isvtx);
13     error = security_ops->inode_ops->mkdir(dir, dentry, mode);
14     if (error)
15         goto exit_lock;
16     dquot_init(dir);
17     lock_kernel();
18     error = dir->i_op->mkdir(dir, dentry, mode);
19     unlock_kernel();
20
21     exit_lock:
22     up(&dir->i_zombie);
23     if (!error) {
24         inode_dir_notify(dir, dn_create);
25         security_ops->inode_ops->post_mkdir(dir, dentry, mode);
26     }
27     return error;
28 }

```

`may_create` verifica se è possibile creare la directory nel `vfs`, altrimenti si verifica se è possibile eseguire la `mkdir` nel `vfs`. si esegue poi l'operazione e se c'è un errore il kernel ritorna ed altrimenti si crea la directory. si possono poi cambiare le modalità di accesso alla directory, quindi ci sono diversi kernel hooks all'interno del kernel.

### 3.5.1 mac on linux

#### s.m.a.c.k

vediamo `s.m.a.c.k`: è basato sulle labels, ogni label è associata a subject, object ed actions (vedi sopra), le labels sono stringhe senza una particolare semantica.

il security module cercherà di matchare la label per verificare la access rule (nella classica forma ottale), le labels associate agli oggetti sono salvate in attributi estesi, quindi serve un file system che li supporta (come `ext4`), ogni processo con una capability `cap_mac_admin` può cambiare le label ed è facile ricadere nelle dac se chi ha la capability può lanciare altri processi.

selinux permette di avere le stesse cose di `s.m.a.c.k`, ma è più complesso di quest'ultimo. ci sono delle label di default nel modulo `s.m.a.c.k`, che hanno le seguenti regole:

- ogni accesso da un subject identificato con `*` viene negato;
- un subject identificato con `_` può accedere in read e execute mode;
- se un oggetto è labellato con `_`, può essere acceduto in read mode o executed
- se un oggetto viene labellato con `*` si può fare tutto

per configurare `s.m.a.c.k`, si usa uno pseudo filesystem apposito, montato in `/sys/fs/smack`, ci sono diversi file possono essere usati per configurare il sistema (slides):

- `access2`: può essere usato per fare query riguardo i permessi di accesso;
- `change-rule`: è possibile cambiare i permessi di accesso per un subject associato ad una certa label

- **load2**: specifica il nuovo permesso di accesso
- **onlycap**: specifica labels aggiuntive che sono richieste dai processi per sfruttare la capability `cap_mac_admin`;
- **revoke-subject**: permette di revocare i permessi per un determinato subject. è utile se ad esempio si verifica nell'ambiente di deployment che c'è una violazione di qualche regola nel filesystem di deployment

la difficoltà di configurare s.m.a.c.k è dovuta al fatto che c'è questa organizzazione in labels.

### tomoyo linux

tomoyo linux è un modulo che non usa la e labels esplicitamente ma usa i path dei file come label, quindi possiamo specificare delle regole di accesso basate su file paths, non serve quindi salvare byte aggiuntivi ma può avere problemi ad esempio per i link simbolici, perché non è più semplice determinare il path corretto.


in tomoyo c'è il concetto di **domain transition**: se un processo a forka un processo b, le regole che possiamo voler usare per b dipendono dal fatto che è stato forkato da a: quindi se lanciamo b come utente deve essere considerato un dominio differente dal fatto che b viene lanciato dopo una fork. supponiamo di usare sshd, che se ha successo si forka in una shell remota: abbiamo una shell remota che deve essere diversa da una shell locale nella macchina, dove vogliamo avere delle ac più permissive. mentre invece se la shell viene generata dopo ssh, vogliamo permessi diversi: magari da **bash** si lancia **ls**, possiamo avere diversi domini:

- `<kernel>`, `</usr/bin/sshd>`, `</bin/bash>`
- `allow_execute /bin/ls`
- `allow_read /home/pierciro/.bashrc`
- `allow_read/write /home/pierciro/.bash_history`

ed avere invece

- `<kernel>`, `</usr/bin/sshd>`, `</bin/bash>` `/bin/ls`
- `allow_read /etc/group`
- `allow_read /etc/nsswitch.conf`
- `allow_read /etc/passwd`

quindi, abbiamo diversi domini ma diventa complicato pensare a tutti i possibili domini per permettere un accesso legittimo a tutti i possibili utenti.

 immagini/selinux\_schema.png

## apparmor

un terzo security module è apparmor, che segue un paradigma diverso in cui l'approccio è che processi individuali possono avere il loro profilo, che è essenzialmente un confinamento per un processo o un gruppo di processi. l'idea è simile a quella di s.m.a.c.k, data la complessità di definire i profili apparmor adotta la policy del **selective confinement**, quindi se un processo non matcha nessuno profilo, quel processo girerà solo con il dac tradizionale, quindi tende ad essere più permissivo. apparmor quindi confina i processi che sono considerati dannosi per il sistema, mentre gli altri sono lasciati liberi, quindi c'è più usabilità ma può lasciare dei buchi di sicurezza in mancanza delle policy. le policy sono in `/etc/apparmor.d`, le policy sono espresse nel linguaggio di apparmor che è poi compilato in una rappresentazione binaria nel kernel. questo è un esempio:

```
1 /bin/bash{
2   /bin/ls cx -> childprofile,
3   network raw,
4   profile childprofile{
5     capability setuid,
6     /home/pierciro/** rw,
7   }
8 }
```

impostare le policy può essere complesso come in tomoyo, ma a differenza fondamentale è che se manca una policy per un processo, apparmor permette di eseguire con i classici dac linux.

## selinux

selinux è il 4° modulo, che è il più complicato modulo di sicurezza linux perché è basato su diversi db (file di testo) e basato su security hooks. c'è una distinzione di invocazione già dallo user space, come mostrato in figura

ci sono una serie di tool forniti da selinux per cui si riconosce un certo servizio come trusted e permette l'accesso a determinati file in base alle regole di selinux. si invoca una syscall, l'hook chiama il build authorization module che costruisce la query per il db che verrà processata dal selinux policy store, che mantiene in memoria kernel associate alla configurazione mac e valuta la query in base alle mac mantenute in memoria.

le regole configurabili nel sistema appartengono a 3 diversi db, di nuovo si parla di labels

- il protection state specifica delle regole che è possibile associare
- il labeling state
- il protection state entra in gioco in quanto ci sono delle regole dinamiche che si attivano quando cambia il dominio

ogni volta che qualche security module hook viene eseguito, c'è la costruzione della query di autorizzazione. in selinux, i parametri per una call al lsm

- subject: il processo corrente

- object: inode, a differenza di tomoyo dove c'era il path e quindi il problema sui link simbolici. sfrutta le capacità di linux per capire già qual è il file finale.
- operazioni richieste

i subject sono convertiti in delle labels, che sono dei contesti in selinux, per un singolo utente ci possono essere diversi ruoli che possono essere associati a diversi contesti e che permettono di definire cosa un processo può fare.

il secondo step è trovare delle policy entry specifiche per un access request: ci sono nuovamente dei file, che vengono compilati in binari per il kernel una volta caricato il modulo. uno statement per la policy può essere di questo tipo: allow subject\_type- object\_type-:object\_class- operation\_set-.

per generare le label c'è il labelling state, che è un db aggiuntivo che mappa le risorse nel sistema a delle label specifiche, in particolare ogni volta che si lancia un processo o si esegue una risorsa, questo

viene mappato nel db di selinux. lo schema è: 

-file path expr-	context
/etc/shadow.*	system_u:object_r:shadow_t:s0

  
abbiamo anche i transition state, che definiscono quali label condizionali possono essere cambiate per determinati subject/object: se ad esempio creo un file in /etc/, avrà la label passwd\_exec ma se il file crea un certo file che verrà labellato come etc\_t, allora la policy farà sì che questo avrà un label diversa: un po' come accadeva in tomoyo per i diversi profili. possiamo anche avere delle transizioni

type_transition	<creator_type>	<default_type>:<class>	<resultant_type>
type_transition	passwd_t	etc_t:file	shadow_t

per gli utenti, quindi se un utente spawna un processo può accadere quanto segue

il punto fondamentale in selinux è identificare le label di default nel labelling state, ma a seconda

type_transition	<current_type>	<executable_file_type>:process	<resultant_type>
type_transition	user_t	passwd_exec_t:process	passwd_t

dell'utente che gira il transition state permette di avere delle regole a grana più fine ed il tipo finale avrà le regole definite nel policy state.

l'organizzazione può divenire complessa per setuppare un mac, abbiamo sempre dei trade off fra la semplicità della regola e la granularità con cui è possibile specificarla.

## 3.6 boot time security


ogni cosa vista fin ora riguardo la sicurezza del so è basata su due cose fondamentali:

- il sistema è live, quindi configurato e con i moduli caricati
- il kernel non è compromesso (ricorda il rootkit)

c'è il punto in cui si fa il booting che è ancora critico, in quanto i dac e mac lavorano prendono per certo che il sistema non sia compromesso. possiamo fare il boot in un kernel compromesso, in modo che nessuna acl riesca ad essere attivata.

ad alto livello, questo è come viene bootato un sistema:

il bootloader è il pezzo di software che carica l'immagine del kernel in memoria, quindi poi il kernel


 immagini/boot\_scheme.png

parte ed una volta che si è configurato il controllo passa all'init ram disk.

l'init ram disk tipicamente mantiene una copia di systemd, dove una volta loggati siamo nel so.

abbiamo secure boot, gli attacchi prima di secure boot erano semplici ma ora il bootloader è un'immagine firmata che viene verificata dal firmware, anche il kernel è firmata crittograficamente, quindi non si possono lanciare dei kernel non firmati.

una volta fatto partire il kernel, abbiamo le mac e quindi l'unico punto rimasto scoperto è l'init ram disk che ad oggi è suscettibile all'horse pills attack: il target è un kernel di os funzionante, al punto prima che parta systemd e sfrutta i container.

il ram disk deve portare nella macchina dei moduli aggiuntivi, questo è ad esempio vero se il kernel è firmato e non ci sono dei moduli per usare parte del mio hardware, che è necessario, trovo i moduli nell'init ram disk.

mentre l'init carica i moduli necessari, potrebbe rispondere a degli hotplug events, oppure setuppare la crittografia (ad esempio per un file system cifrato), inoltre occorrerà localizzare il filesystem da montare come root filesystem da qualche parte sul disco, dopo questo si può saltare al processo di init che pulisce quanto caricato dal ram disk e parte col boot vero e proprio.

### 3.6.1 horse pills

in ram disk ci sono degli init script, in cui è possibile nascondere delle back doors sfruttando i namespace (sì, come mettere una pillola in culo ad un cavallo). abbiamo l'init process, che è associato al pid 1 e ci saranno dei processi relativi al namespace dell'init, ma l'init non è il processo con pid 1 fuori dal suo namespace, quindi dal suo container.

se un init script compromesso crea un container per fare il boot del reale processo di init: questo avviene nell'horse pill attack: si fa tutto quello che avveniva precedentemente, ma poi si enumerano tutti i kernel threads che stanno girando, si chiama poi la `clone` syscall che è una variante della `fork` che permette di specificare dei flags per creare un nuovo namespace, quindi un container nuovo. nel container

- si può rimontare `/proc`, quindi si nasconde quello che c'è fuori dal container
- si creano dei fake kernel thread, che non fanno nulla ma hanno lo stesso nome dei kernel thread veri
- si fa il cleanup di `initrd`
- si esegue `init`

init non è però quello regolare, fuori dal container si può rimontare qualunque root filesystem, che sarà diverso da quello visto nel namespace creato prima, quindi si può fare qualunque cosa:

- montare qualunque filesystem
- fare `fork()` di qualunque programma come shell di backdoor
- `waitpid()`, ovvero si aspetta il completamento della `clone`, in modo che se avviene uno shutdown o reboot avviene il catch dell'evento e si fa sul proprio volere

per mitigare l'attacco:

- in `/proc/<pid>/ns links` si trovano delle informazioni che indicano che si sta eseguendo in un container e quindi è probabile che ci sia un attacco in corso
- verificare che le entries dei threads abbiano `ppid  $\neq$  0`
- fare auditing

per risolvere davvero è non assemblare il ramdisk sul sistema, che è quello che avviene quando si aggiorna il kernel ma è poco fattibile in quanto assemblare il ram disk viene fatto dalle distro per le differenti macchine. in questo modo, servirebbe un kernel firmato per ogni macchina diversa, quindi rendere sicuro il ram disk iniziale richiede una configurazione mac adatta che però può arrivare dopo che è avvenuto lo script di inizializzazione compromesso.

## 3.7 os-level network security

il trend corrente è che non vengono più usate, ma possono portare ad avere una sicurezza migliore sul sistema. ci sono diverse applicazioni di rete che aprono porte alla cazzo di cane e quindi impediscono a questi servizi di essere efficaci.

se sulla macchina gira un qualche servizio di rete, è possibile renderlo più sicuro con le acl. un approccio usato nelle architetture erano due servizi:

- super-servers
- tpc containers

che rispondevano alle richieste in ingresso con delle acl apposite

### 3.7.1 inetd

servizio legacy che installa una socket listening, riceve una connessione e spawna una applicazione trasferendo la socket su quella applicazione. c'è un file `/etc/services` che serve per configurare il mapping fra la porta/protocollo e l'applicazione. l'obiettivo di inetd (nei 70-80) era quello di salvare le risorse, in quanto se la macchina ha poca memoria e ci sono diversi servizi di rete, occorre lanciare diversi demoni di rete e così c'è un singolo demone. inetd può essere esteso con parti di sicurezza, in particolare c'è il file `/etc/inetd.conf` in cui si associano le applicazioni con una struttura su una linea:

- nome del servizio, come è espresso in `/etc/services`
- tipo di socket
- protocollo della socket
- flag di servizio (`wait/nowait`)
- l'id utente da associare all'istanza running del servizio
- il path del file eseguibile e gli argomenti (se ci sono)

inetd è stato quindi esteso in xinetd, dove ci sono diversi tipi di acl:

- address based
- time frame


è possibile specificare qualcosa come: è possibile ricevere una connessione da questo range di indirizzi o che duri questo tempo massimo.

tutti i log generati sono centralizzati e può essere usato per prevenire dos ad esempio limitando il numero massimo di istanze per servizio o il numero massimo di istanze totali di server.

### 3.7.2 tcp daemon: tcpd

è un demone arrivato dopo initd e che viene chiamato a sua volta da initd, per supportare regole di ac aggiuntive. è possibile specificare per ogni applicazione la lista di client che si sono connessi all'applicazione, ad esempio si può dire che è possibile accedere ad ogni servizio se connessi localmente, ogni riga è strutturata come `daemon_list: client_list`.

tcpd ha una capability che permette di controllare se un attacker sta cercando di bypassare tcpd con dns tampering, come mostrato in seguito

 immagini/tcpd\_tampering.png

quindi tcpd fa un forward name to ip ed anche un reverse ip to name, che viene fatto con diverse zone dns quindi con dei record dns diversi e frega l'attaccante che può fare tampering con solo una zona dns.

l'unico requirement dell'applicazione che lancia tcpd è di poter leggere dati da stdin e rimandarli su stdout, ci centralizza il controllo e si verifica chi può usare un certo servizio senza replicare il demone di sistema.



# Capitolo 4

## antivirus

concetti legati ai software anti-virus, in particolare alcune delle metodologie usate dai tool. un antivirus è un pezzo di software che gira in background ed il suo obiettivo è quello di catturare la presenza di software malevolo che può essere

- salvato sul disco
- running
- proveniente dalla rete, quindi c'è il check sui pacchetti
- euristiche per capire se un attacco arriva da un device remoto

non è software bullet proof, quindi se un malware passa, l'antivirus cerca di rimuoverlo.

l'idea è quella di essere un software di prevenzione ma ci possono essere ad esempio 0-days che saltano nella macchina. gli anti-virus combattono vari problemi, il meccanismo di detection è basato su delle tecniche per riconoscere il malware ma anche quest'ultimo mette in atto delle pratiche per non farsi riconoscere dalla detection del malware: solitamente, il developer del malware usa delle syscall non documentate per la costruzione del software, ad esempio per so windows e spesso ci sono delle syscall rimosse da una versione all'altra oppure c'è il supporto ma non sono documentate. il kernel è comunque un binario e quindi si può disassemblare e capire quali syscall non documentate possono essere sfruttate. un altro problema è legato alla superficie di attacco, che per sistemi moderni è molto grande:

- si può fare targeting delle applicazioni di sistema
- servizi di sistema, se c'è un bug può essere sfruttato dai virus. es: bug dei 90', un bug sfruttava il servizio che permetteva alle stampanti di condividerle sulla rete.
- os: grandi target dei virus
- email: gli utenti faranno azioni sciocche e renderanno la vita dell'infezione del virus semplice
- network: come ddos

un antivirus dovrebbe monitorare tutta la superficie di attacco del sistema, quindi sono resource-intensive. un problema addizionale è che il numero di software malevolo che viene sviluppato è molto ampio, quindi gli antivirus ricevono 100-1000 samples di applicazioni amlevole per day da dover analizzare. quindi ci sono reverse engineers che studiano a mano il software malevolo in quanto

l'antivirus consuma troppe risorse sulla macchina, quindi i security engineers ricevono i samples, li analizzano e se decidono che è malware installano a mano nell'antivirus un controllo automatico per quel malware.

prima, i malware erano scritti per mostrare che si poteva fare qualcosa di fico ma non era legato a fare danni veri sulla macchina, mentre oggi si fa per motivi di soldi: fai un sacco di cose zozze come ransomware o appropriazione di account di banca per fare riciclaggio (stamo su narcos in pratica), ci stanno poi di mezzo organizzazioni, governi etc...

inoltre, ultimo problema è che l'antivirus è software e quindi può avere dei bug, per questo l'antivirus deve essere robusto agli attacchi. tipicamente sono scritti in linguaggi compilati, come c e c++ per essere veloci e consumare poche risorse.

all'inizio, gli anti-virus erano solo dei software per fare pattern matching, oggi la situazione è cambiata e quindi nessun antivirus offre cli per fare ispezione manuale e sono basati su gui per essere semplici e cercano di fare la maggior parte delle azioni in background. quindi in qualche modo l'antivirus installato prende il controllo della macchina in quanto deve verificare diverse cose in background, installano inoltre in automatico dei firewall. anche i browser hanno dei meccanismi interessanti per proteggersi ma comunque se compromesso il browser può causare problemi e quindi anche in questo caso gli antivirus inseriscono dei controlli nel browser.

visto l'ammontare di samples che vengono ricevuti, ci sono diversi updates ogni giorno e quindi l'organizzazione interna dovrebbe essere il più modulare possibile e non è più possibile fare solo pattern matching:

- il malware può cercare di evadere tale pattern
- tenere in memoria tutti i pattern è costoso

gli antivirus usano quindi delle euristiche, ma vengono usate anche su malware non ancora scoperto dall'azienda stessa. come euristica di base usano la conoscenza del passato per determinare l'outcome futuro. le euristiche possono dare falsi positivi ma anche falsi negativi.

#### 4.0.1 componenti degli anti-virus

il componente principale è il kernel dell'antivirus, è un processo che parte allo startup della macchina, diviene residente in memoria e governa le azioni degli altri componenti. implementa dei meccanismi di self-protection, ad esempio intercetta segnali come sigkill per evitare di crashare. se questo componente viene compromesso, c'è poco che si può fare e quindi è quello che viene analizzato e revertito per essere attaccato.

c'è uno scanner, che controlla un file per determinare se è malevolo o no. è residente, quindi parla col so per vedere se ci sono file che vengono scaricati o creati sul disco: c'è sempre una fase in cui il virus viene scritto sul disco quindi se l'antivirus può detectarlo riesce ad intervenire per tempo.

ogni so moderno ha dei bus che sono sistemi pub-sub (publish subscriber) per notificare ai software quando cambia un file, inoltre gli antivirus rimpiazzano le applicazioni di sistema: ricevono tutte le richieste ai servizi di sistema e se le richieste sono legittime le passano ai servizi di sistema reali.

ci sono filter drivers per il file system: spesso, un antivirus installa un qualche tipo di driver per il file system in modo da scoprire subito dei problemi sul file system prima che il file venga scritto, stessa cosa può valere per la rete, ad esempio con ebpf che controlla i pacchetti prima della consegna.

ci sono poi dei signature db, gli antivirus moderni hanno dei db che contengono dei binari che vengono caricati nell'address space del kernel e fanno dei controlli sul presunto file malevolo. alcuni binari cercano fingerprint nel file in modo da dire al kernel che sono fiduciosi del fatto che il file è un

virus, applicazioni più semplici usano dei checksum basati su hash ma cambiando anche un solo bit il controllo salta.

spesso i virus vengono scaricati in formati packed per fregare l'antivirus, quindi deve esserci un qualche componente di unpacking per poter riconoscere questo tipo di packing.

è possibile mettere codice malevolo in diversi formati di file, quindi gli antivirus devono essere in grado di riconoscere diversi formati dei file ed interpretarne il contenuto, fra i più comuni

- pe
- elf
- png
- jpg
- docx: l'organizzazione interna è basata sugli oel object (??). non c'è documentazione del formato, quindi tutta la nozione viene dall'effort di libreoffice, staroffice etc... ma la conoscenza è comunque incompleta
- pdf: lo standard è di più di 2000 pagine, quindi ci possono essere delle sezioni non chiuse che non sono renderizzate e che contengono codice e quindi l'antivirus dovrebbe essere in grado di capire che il programma di interpretazione del pdf ha sbagliato.

tutti i controlli fatti sul file dipendono dall'ammontare di informazione che l'azienda di antivirus ha e sul totale del numero di controlli che il software fa: può capitare che alcuni antivirus identificano un file come malware ed altri no.


i filtri per i pacchetti possono essere ad esempio implementati con ebpf etc...

meccanismi di auto-protezione: oggi ci sono randomizzazione degli indirizzi etc... mentre prima non era così e quindi il kernel dell'antivirus doveva fare dei controlli automatici su se stesso e spendevano diversi cicli di clock per capire se qualcuno cercava di manipolare il kernel.

ci sono poi gli emulatori: l'idea è che se gli antivirus ricevono un binario e in base ad una euristica decidono che può essere malware ma non possono decidere con sicurezza che lo sia tirano su un ambiente containerizzato per verificare il comportamento del malware e per verificare un certo pattern nel comportamento. anche questo porta al consumo di risorse

## unpackers

l'immagine seguente è un'idea di alto livello di come il malware può evitare di essere analizzato:

 immagini/unpacker.png

un binario è organizzato in sezioni: dati, testo etc... ed ogni sezione da informazioni al so per capire come far girare il programma e quindi leggerà le sezioni ad esempio per capire i permessi da dare alle aree di memoria.

c'è quindi un header, in cui viene descritto quali sezioni ci sono, una cosa interessante è che ci possono essere sezioni sul disco che hanno un ammontare di spazio più piccolo di quello effettivo.

alcuni virus hanno istruzioni per far sì che girino in vm e per questo è utile che ci sia un ambiente emulato per l'antivirus ma ci sono comunque meccanismi per provare a scoprire se l'emulatore è buggato e sfuggire all'ambiente controllato.

una volta che lo stub prende il controllo, spacchetta l'eseguibile varo e lo carica in memoria per poi implementare i meccanismi del so (vedi appunti ma sui packer /unpacker) per far girare il nuovo pe. l'obiettivo è sempre quello di evitare che l'antivirus possa fare controlli basati su checksum, ci sono inoltre diverse indicazioni che il binario è stato impacchettato per il software antivirus:

- l'iat è piccola
- nomi delle sezioni non standard
- sezioni con una taglia raw piccola ma una taglia virtuale grande
- poche stringhe distinguibili
- sezioni con privilegi rwx: questo è dovuto al fatto che il binario verrà spacchettato ed il codice eseguito
- istruzioni di jmp o call a registri o indirizzi di memoria strani, poiché diversi packer convenzionali salvano l'indirizzo di dove spacchettare in un registro. (long jump in ma's appuntis)


## signatures

tipicamente sono pattern, sequenze di byte che cercano di fare matching di sequenze di byte usando ad esempio hashing crittografico. il problema è che se si cambia anche un solo bit siamo fregati, quindi gli antivirus moderni implementano dei plugin appositi per fare checking di un file. ci sono diversi plugin da lanciare per capire se un file specifico rispetta un pattern, tipicamente le regole sono scritte dai reverse engineers.

l'idea è quindi quella che i virus che l'antivirus comune riconosce sono magari auto-generati e che usano sempre le stesse tecniche e sono quindi più giocattoli.

### 4.0.2 tecniche di evasione

abbiamo il disassembly di un certo virus:

 immagini/poly\_code.png

aggiungiamo del codice che non fa nulla, non solo nop che sono facili da capire, ma proprio delle istruzioni che non fanno nulla. per le nop è facile skipparle nel calcolo della signature, ma le istruzioni che sembrano cambiare qualcosa ma in realtà non cambiano nulla fregano diversi degli antivirus gratis.

un'altra tecnica è rendere il codice **spaghetti code**: si aggiungono jump avanti ed indietro nel codice così che la signature venga resa inefficace in quanto cambiano le istruzioni. la tecnica di cambiamento più semplice è che il virus sappia quali siano i suoi blocchi fondamentali e quindi cambi alcune delle parti per rendere le firme non efficaci.

### **virus polimorfi/metamorfi**

il codice polimorfo è qualcosa che cambia se stesso ogni volta che viene lanciato. ogni virus genera una nuova variante di se stesso ogni volta che viene generato, in particolare l'implementazione più semplice è quella di cambiare ogni volta lo stub quando si infetta qualcuno: un virus polimorfo ha diverse versioni di packer/unpacker logic, centinaia, in modo di fare repack di se stessi ogni volta che infettano qualche target scegliendo un algoritmo diverso ogni volta. spesso vengono quindi detti virus oligomorfi.

abbiamo poi i virus metamorfi, che generano varianti di se stesso che fanno le stesse cose ma con istruzioni differenti: implementano nella loro logica delle tecniche usate dai compilatori:

- si decompilano
- c'è una versione intermedia, cambiano del codice per far sì che la logica sia la stessa

il problema è che il codice generato dopo un certo numero di iterazioni aumenta di molto la sua taglia. quindi, dopo un certo numero di iterazioni, si applicano delle tecniche di shrinking, ad esempio convertendo due istruzioni in una sola. l'interpretazione intermedia si passa ad un assembler che restituisce il binario finale ed inoltre è possibile avere che tale binario sia eseguibile su diverse architetture.

### **firme comportamentali**

le behavioral signatures sfruttano il fatto che ci possono essere diverse versioni dello stesso programma, ma alla fine lo scopo dell'algoritmo attuale sarà sempre lo stesso. quindi una behavioral signatures guarda cosa il virus fa e non come è fatto, quindi gioca un ruolo fondamentale l'emulazione: possiamo misurare

- quali istruzioni vengono eseguite nell'ambiente
- quali syscall chiama

si può poi creare un istogramma per raccogliere i dati e mantenere i bin dell'istogramma nel db al posto della firma. si può complementare questa tecnica con un sistema di scoring


### **engine di euristiche**

gli engine per le euristiche sono dovuti al fatto che i samples ricevuti sono troppi e non riescono ad essere analizzati velocemente, quindi implementano dei classificatori e tecniche di ml per capire se un file è malevolo o no.

la prima versione era basata su le sequenze di api chiamate come una feature del malware. un'altra feature usata è l'opcode delle istruzioni, quindi si guarda alle sequenze o sotto-sequenze di opcodes. è possibile però avere un ammontare grande di sequenze, ad esempio architetture cisc come intel usano un grande ammontare di nop che quindi non serve salvare, si possono anche costruire dei pattern per riconoscere nop più complesse come quelle nell'immagine sopra per fare filtering sulle feature del dataset (tiè, come se vede che conosco il ml).

un'altra tecnica è quella di usare un call function graph, spesso generato dai compilatori per ottimizzare lì dove la funzione viene eseguita spesso, come mostrato in figura:

nell'analisi del malware gli antivirus li usano costruendoli guardando solo specifiche istruzioni

 immagini/flow\_graph.png

- jmp
- jcc
- call
- ret

poi si cerca di ridurre i nodi del grafo, quindi ogni nodo che è una sola istruzione o solo un jmp rimuove il nodo in modo da linearizzare il flusso del grafo. si cerca quindi di produrre un grafo in cui si possano identificare solo le istruzioni più importanti di jump. è usato come signature per riconoscere il malware, viene costruito a run time e matchato con quelli presenti nel database.

l'altra cosa usata sempre nel contesto del ml è l'n-gram ed è fondamentalmente una sequenza di n oggetti da un dato oggetto di testo o discorso, ovvero un **n-gram** ad esempio "the cow jumps over the moon"

è possibile ad esempio riconoscere coppie di istruzioni che realizzano nop. i classificatori più usati sono

- naive-bayes
- support vector machine
- decision tree

boosted decision tree sono quelli con risultati migliori per l'analisi del malware quando si usano gli n-grams. anche in questo caso vanno generati gli n-gram a run time girando in un ambiente simulato. abbiamo poi i byteplots, che trasformano binari in immagini: le immagini sono non-senso e ci sono tecniche di deep learning in cui si usano reti neurali che sono molto efficienti nel parsare e classificare immagini in 2d. quindi si possono classificare i binari con le reti neurali e la cosa ottima è che possono mettere il loro focus sul dettaglio della parte affetta dal malware e non su tutto il binario. questo richiede comunque tempo e consuma risorse.

### 4.0.3 emulators

le applicazioni malware cercano di fare del loro meglio per battere le tecniche messe in atto dagli antivirus, ad esempio mediante la compressione. un antivirus usa fra le sue tecniche l'emulazione, dove quindi viene messa su una vm per far girare l'applicazione in una sandbox. dentro questa sandbox, viene lanciato il presunto malware per verificarne il comportamento, l'obiettivo della tecnica è quello di lasciare che il virus faccia delle azioni sulla macchina virtuale per poter usare una delle tecniche di difesa viste sopra.

si può comunque scoprire che si sta girando in sandbox, in quanto l'antivirus non implementa davvero tutti i servizi del so, molti sono solo stub che danno dei magic values come risultato, ad esempio `openmtextw` che restituisce sempre lo stesso valore e quindi il malware può capire se è in un ambiente emulato facendo una sorta di finger printing, ad esempio chiamando questa syscall in modo da poter capire quanto è verosimile che stia girando in una sandbox.

in base a questo, il malware si comporta diversamente e quindi l'antivirus non riconosce il comportamento corretto.

si può anche scaricare l'antivirus e disassemblarlo per vedere se ci sono questi stub e si cerca di capire i fingerprint degli emulatori, le contromisure degli antivirus è di fare le stesse cose: se si cerca di fare debugging del kernel dell'antivirus ci sono tecniche anti-debugging oppure hanno i binari cifrati e quindi usano le stesse tecniche del malware.

#### 4.0.4 analisi del traffico cifrato

molti antivirus possono proteggere il traffico anche se questo è cifrato, ad esempio cifrato con tls ma sappiamo che tls non dovrebbe permettere a qualche sorgente esterna di vedere i dati. si usano le stesse cose che farebbe il malware per ispezionare la rete: montano un mitm, l'antiviru stesso lo fa sulla macchina, ed installa un certificato firmato con una ca trusted, è complicato in quanto devono installare una serie di certificato on-the-fly. quindi, quello che alcuni antivirus fanno è installare una nuova lista di root ca, quindi tutto il traffico viene rediretto verso l'antivirus ed ispezionato. questa tecnica non è sicura per diversi motivi:

- molti browser usano http pkp
- se l'implementazione non è corretta si può essere attaccati con altri mitm
- alcune implementazioni accettano dh key exchange di 8 bit
- una implementazione debole di tls può essere vulnerabile anche per gli update dell'antivirus

non usare mai l'https inspection sull'antivirus.

#### 4.0.5 ci si può fidare del software av?

la guida generale di sicurezza dice di aggiornare la password spesso ed aggiornare gli av. ma gli av usano spesso linguaggi compilati memory-unsafe, possono essere fregati dai malware quindi fidarsi al 100% dell'antivirus è sbagliato, l'unica cosa che si può fare è alzare la barra di sicurezza per il proprio sistema. se si crea un falso senso di sicurezza negli utenti ci può essere un problema, gli av giocano un ruolo chiave nel contrastare i malware ma non siamo in un sistema sicuro.

# Capitolo 5

## database security

### 5.1 db security tools

uno dei maggiori target degli attacchi sono i dataset, in quanto questi hanno un grande valore sia per le aziende che per gli attaccanti.

ancora oggi il datastore più comune è un database relazionale, c'è poca educazione e sforzo per rendere i database sicuri.

si parla di proteggere diverse cose:

- il sistema su cui gira il dbms
- i dati
- l'accesso al db

rendere quindi un db più sicuro implica essere robusti sia ad attacchi tradizionali ma anche a danni accidentali legati a fenomeni naturali.

abbiamo diversi tool da usare:

- security policies: dicono come si cerca di ottenere la sicurezza del dbms. ci sono dei tool offerti dal dbms per aumentare la sicurezza, quindi la policy riguarda sia i meccanismi del dbms che le policy "umane". non basta solo questo, quindi occorre considerare anche qualcosa di esterno al db

#### 5.1.1 requisiti ed obiettivi

ci sono quindi dei requisiti e degli obiettivi, il curriculum di sicurezza è abbastanza leggero, le persone considerano i dbms software che gira da qualche parte senza problemi.

un sacco di effort per rendere i db sicuri riguarda come rendere la connessione al db sicura che però non rendono globalmente il db più sicuro, quindi va considerato il sistema complessivo e non solo i dbms.

occorre quindi capire

- i problemi di sicurezza
- la specifica implementazione del dbms
- principi di sicurezza generali come le mac



- policies, least privilege etc...
- i dati sono salvati su hard drive, il dbms li gestisce solamente: è software server side che manda i dati

### 5.1.2 `sql`i

un dbms riceve connessioni dove vengono richieste delle stringhe, che se non sanitizzate correttamente possono portare a diversi problemi. c'è poco da dire, usa almeno gli escape sui caratteri speciali ma la cosa importante è che occorre usare i **prepare statements**: è l'unico modo corretto di interagire con un dbms: si manda una query non completa, con dei placeholders al posto dei veri parametri, quindi ogni volta la query incompleta è parsata, analizzata e poi compilata. si mandano al dbms solo i parametri che realmente si vogliono usare. la tecnica è stata introdotta per motivi di performance, in quanto per connessioni a lungo termine si aveva la query compilata una volta sola, oggi sono usate per rendere il sistema più sicuro e resiliente.

### 5.1.3 database authentication ed authorization

sono delle tecniche che permettono di implementare dac o anche mac se usate bene. sono quindi tool fondamentali che permettono di rendere l'applicazione più sicura: avendo una sola connessione con un singolo utente non abbiamo nemmeno dac. è un mindset complesso perché molte librerie che si usano tendono a rendere la vita più facile ma al prezzo di avere meno sicurezza: se si usa una sola connessione con jdbc si fa authentication ed authorization male. quindi:

- quando si parla di autenticazione, si parla di un meccanismo che determina se un utente è chi dice di essere. un amministratore di sistema è il responsabile per permettere agli utenti accesso al sistema creando account utenti individuali. almeno, il sys admin può dare permessi specifici agli utenti, ed è importante non riciclare le credenziali per diversi utenti, sia umani che non.
- l'autorizzazione permette di determinare, per un utente legittimato ad usare il db, quali azioni può fare. quindi, ogni volta che un utente cerca di fare un accesso ad un oggetto in rwe mode va verificato l'accesso e quindi questo è il modo di realizzare il principio di minimi privilegi. ci sono moltissime applicazioni 3-tiered che usano l'account di root per entrare nel db, quindi si tentano `sql`i per fare dumping dei db

### 5.1.4 `sql` access control e privilegi

`sql` standard permette di specificare chi e come può usare una porzione del db. gli oggetti privilegio sono tipicamente tabelle, ma è possibile avere acl a grana molto più fine per attributi, viste, procedure etc...

c'è un utente predefinito che ha accesso a tutto per default ma questo non è compliant con le mac e quindi molti db permettono di cambiare questa cosa. un aspetto da ricordare è che l'utente che crea una risorsa ha accesso completo (pieni privilegi) a cosa crea, quindi anche il deployment del db è problematico e va gestito correttamente.

un privilegio in `sql` è caratterizzato da

- la risorsa di riferimento

- il soggetto, ovvero l'utente che riceve il privilegio
- chi è l'utente che ha dato il privilegio
- l'azione permessa
- la capacità di trasferire il privilegio ad altri utenti, quindi è possibile avere la propagazione dei privilegi e che può causare problemi.

questi sono i privilegi basilari in sql:

- **insert**: permette l'inserimento di una tupla
- **update**:
- **delete**
- **select**
- **reference**
- **privilege**
- **usage**
- **execute**: permette di lanciare una specifica stored procedure

per poter dare privilegi, si usa il comando **grants**, dove la parte delle **grant options** permette di propagare il privilegio. per revocarlo, si usa **revoke**. è una pratica di acl a grana molto fine per la gestione dei dati, in modo che se questi sono molto critici, si può pensare di usare un dbms anche per l'accesso ai file. sql-99 ha introdotto l'idea di **ruolo**, in modo da associare i privilegi ad uno specifico ruolo e gestire la vita dell'applicazione cambiando i privilegi

### 5.1.5 views

le viste sono state introdotte in sql un meccanismo di sicurezza avanzato per estrarre i dati: si crea una tabella virtuale che restringe l'ammontare degli attributi che un utente vede. in questo modo, se non vogliamo che un utente osservi determinati attributi costruiamo una vista e l'utente non si renderà nemmeno conto dell'esistenza di quegli attributi. possiamo quindi leggere dati da una vista come se fosse una tabella, inoltre la vista può essere aggiornata solo se definita su una singola tabella. se il db usa le viste e viene compromesso, c'è comunque un livello di sicurezza in più

## 5.2 altri principi di sicurezza

i dischi che mantengono i dati possono essere attaccati, bisogna considerare anche la possibilità che qualcuno salta nella sala server e spacca i dischi con un martello (matto? ncazzato? capiamo), il down di facebook whatsapp e ig è durato così tanto per via del fatto che un cristiano è dovuto volare dall'altra parte del mondo per risolvere il problema.

### 5.2.1 raid


i dischi possono essere combinati in gruppi o array ed il pc che fa girare il software del dbms vedrà questo come un unico hard drive ma i dati in realtà verranno scatterati sui diversi dischi. il modo in cui si configurano i raid darà un certo livello di protezione. si usa la ridondanza, in questo caso la replicazione dei dati in modo da avere almeno una copia in caso di fallimento. anche in questo caso, il tradeoff è fra performance e sicurezza, i problemi fondamentali quando si configura un raid è tenere conto delle due operazioni fondamentali (letture e scritture) ma queste possono essere fatte in modi diversi. abbiamo dei controller per la gestione dei dischi, quindi se riusciamo ad avere un certo livello di concorrenza sulla lettura /scrittura dei dischi abbiamo diversi risultati.

il **data striping** consiste nel salvare i blocchi di un certo file su più dischi, lo striping è la ragione per cui abbiamo questo tradeoff fra throughput e concorrenza in accesso:

- uno stripe piccolo vuol dire avere throughput alto ma poca concorrenza
- uno stripe alto il contrario.

#### raid-0


ci sono diversi livelli che usano i molteplici dischi in base a come si fa lo striping dei dati. in raid-0 non c'è ridondanza: abbiamo diversi dischi su cui si scrive una porzione del file su ciascun disco, quindi la bandwidth dei dischi è incrementata. è il miglior modo per salvare i dati, ma non c'è tolleranza ai guasti quindi se uno solo fallisce si perdono tutti i dati.



immagini/raid0.png

#### raid-1

anche detto disk mirror, abbiamo due dischi che mantengono la stessa esatta copia dei dati.




immagini/raid1.png

le performance in lettura sono le migliori, perché si può scegliere una copia piuttosto che un'altra, ma a scapito dell'operazione di scrittura. la tolleranza ai guasti è buona, inoltre è possibile rifare il pairing in modo che se una delle due copie fallisce se ne mette una nuova senza fare rebooting della macchina

#### raid-2

non usa lo striping dei blocchi ma dei bit, quindi ogni bit viene scritto su una porzione dell'array.


il controller del disco soffre tantissimo, quindi perché è stato fatto: c'erano bit addizionali salvati sui dischi che erano bit di parità implementati con codici di hamming in modo che se ci fosse un

immagini/raid2.png

fallimento si potesse ricostruire l'informazione. è correntemente obsoleto in quanto servono un botto di dischi e l'implementazione è complessa.

### **raid-3**


implementa striping byte-level, quindi nell'immagine abbiamo i byte del file

immagini/raid3.png

non serve error correction perché il controller del disco sa quale disco fallisce, ci sono i bit di parità per poter ricostruire l'informazione in caso di fallimento, c'è un solo disco aggiuntivo. abbiamo il throughput migliore ma la concorrenza è terribile perché per leggere si prende il lock su tutto il disco e quindi non si usa molto specialmente per supportare dbms

### **raid-4**


si fa striping di blocchi di file

immagini/raid0.png

i diversi blocchi sono scritti sui diversi dischi e ce n'è uno addizionale che mantiene informazioni sulla parità, il problema è che questo disco diventa un bottleneck: piccole operazioni di scrittura scrivono un solo bit su un solo disco, ad esempio, ma va comunque aggiornata l'informazione sulla parità.

### **raid-5**


una delle configurazioni tipicamente usate nei data center. abbiamo sempre block level striping, buoni in quanto le scritture piccole possono essere fatte su un solo disco, inoltre la parità è distribuita

immagini/raid5.png

quindi, se abbiamo piccole scritture, i bit di parità vengono aggiornati solo su alcuni dischi e quindi aumenta la concorrenza sul disco, possiamo però tollerare solo il fallimento di un disco

## raid-6

block striping con due insiemi di parità: per ogni stipe di dati abbiamo due blocchi distribuiti che mantengono i bit di parità. la doppia parità è utile in quanto se si perde uno dei dischi, il raid-6 diventa un raid-5, quindi abbiamo tolleranze a due guasti


immagini/raid6.png

le scritture sono peggiori, in quanto vanno aggiornati due blocchi ma siccome abbiamo questa ridondanza possiamo leggere da diversi dischi se alcuni sono occupati.

avere più tolleranza alla rottura dei dischi è utile in quanto se si compra un batch di dischi dallo stesso vendor la possibilità che due dischi falliscano nello stesso momento è alta.

## raid-10

si creano due livelli di array ridondati: abbiamo due raid-1, uniti insieme in uno raid-10

immagini/raid10.png

questa è l'idea tipica per dbms relazionali che fanno diverse letture/scritture quindi sono avvantaggiati dei raid-1 sottostanti ma hanno anche ridondanza.

## 5.3 backups

abbiamo visto il concetto di array ridondanti di dischi, alla fine abbiamo detto che una delle organizzazioni migliori è la raid-10.

molta della reliability è legata al backup: in ogni data storage dovrebbe essere possibile avere dei modi per recuperare i dati anche se si è perso l'accesso.

l'idea di un backup è che si può avere uno snapshot periodico di tutti i dati salvati da qualche parte ma c'è un problema di autorizzazione: un utente non legittimo non dovrebbe potere vedere tutti i dati, ma un backup contiene tutti i dati, quindi questo allarga la superficie di attacco.

si potrebbero usare dischi che sono economici, ma chi ha accesso alla stanza fisica dove sono i cassette? quindi sia logico che fisico, il backup del sistema nell'infrastruttura ha problemi sulle policies, che vanno ben definite.

### 5.3.1 distater recovery

vogliamo far sì che il sistema sia più sicuro anche contro eventi catastrofici:

- possiamo salvare i dati in qualche disco messo in qualche stanza con qualcuno che può entrare. ma se l'edificio va a fuoco, si perde sia il main site che il backup.

- copie geograficamente distribuiti, pensando ad esempio a banca d'italia, questi hanno 3 siti differenti per tutta la nazione.

la disaster recovery andrebbe quindi realizzata tenendo presente la dislocazione geografica, servizi cloud permettono di realizzare questo tipo di dr: ad esempio le availability zones e le regioni di aws, per implementare correttamente la dr.

dobbiamo quindi considerare tutti i possibili avvenimenti: un ransomware ha poca probabilità di colpire le copie dei dati localizzate in punti geografici distanti.

avere quindi un piano di dr prevede di copiare i dati in almeno 3 zone differenti, le copie devono avere le loro policy di accesso e verranno usate solo in caso di attacchi/perdita di dati per avere continuità di business.

consideriamo 2 misure classiche, per capire quanto è buono il piano di dr:

- rpo: qual è l'ammontare di dati che ci si può permettere di perdere. ad esempio quindi, ogni quanto facciamo lo snapshot del db. ovviamente, più si fanno snapshot più costa
- rto: quanto ci vuole a tornare up&running. ad esempio quindi, abbiamo una copia del codice, quindi delle vm da poter ripristinare. possiamo ad esempio avere una vm già operativa che non fa nulla e che nel momento in cui la principale fallisce è pronta per essere attaccata al db e per ricevere il traffico utenti.

tipicamente, le aziende non fanno ingegnerizzazione appropriata della dr, l'idea è che ogni volta che si progetta un sistema complesso bisogna farsi queste domande e progettare anche la dr.

## 5.4 encryption

abbiamo comunque un problema: il dr è ok, le policy anche, ma abbiamo un data breach ad un certo punto. può essere colpa di un bug o altro ed i dati sono compromessi da qualcuno.

nei db i dati sono tipicamente non cifrati, tranne per eccezioni come le password, ma in un modo ideale vorremmo avere tutti i dati cifrati quindi si potrebbe permettere solo agli utenti autorizzati di accedere ai dati.

il problema è che il costo di processare i dati aumenta, ma si aumenta il livello di sicurezza inoltre il problema riguarda anche il canale che si usa per trasmettere i dati, che dovrebbe essere cifrato a sua volta (possibile usando ssl).

non basta comunque: siamo in uno scenario distribuito, abbiamo due db che usano cifratura ma sono di due aziende diverse e questo può creare dei problemi. l'approccio più semplice sarebbe avere delle chiavi per ogni azienda, ma se non c'è fiducia? in ogni caso si vuole cercare di poter scambiare per leggere i dati

### 5.4.1 cifratura omomorfa

si possono fare delle operazioni sui dati cifrati senza dover per forza leggere i dati de-cifrati. è un modo per far scambiare i dati fra delle aziende che non si fidano l'una delle altre.

tipicamente, abbiamo 3 operazioni:

- key generation
- data encryption

- data decryption

si usano delle funzioni che hanno una "trapdoor", quindi la chiave permette di fare operazioni computazionalmente fattibili.

in uno schema di encryption, c'è un algoritmo di **evaluate** che accetta una chiave pubblica **pk**, una funzione  $f$  ed un insieme di testi cifrati e fa:

$$c_f \leftarrow \text{evaluate}(\mathbf{pk}, f, c_1, \dots, c_t) \quad (5.1)$$

la prima proprietà per lo schema omeomorfo è la correttezza: se abbiamo la chiave condivisa e guardiamo a qualche funzione che produce il cipher text, decifrandola otteniamo lo stesso risultato che otterremmo computando la funzione sui dati in chiaro:

$$\text{decrypt}(\mathbf{sk}, \text{evaluate}(\mathbf{pk}, f, c_1, \dots, c_t)) = f(m_1, \dots, m_t) \quad (5.2)$$

non basta, perché se lo schema fosse solo così la funzione di **evaluate** potrebbe anche essere la semplice funzione identità. dobbiamo poter avere un modo per dire che la  $f$  non sia qualcosa che è l'identità

### somewhat homomorphic crypto

lo schema permette di supportare addizione e moltiplicazione di interi, utilizza i seguenti parametri:

- $\eta$ : la lunghezza in bit della chiave segreta
- $\gamma$  la lunghezza in bit degli interi nella chiave pubblica
- $\rho$  la lunghezza in bit nella noise

lo schema lavora così:

- la chiave segreta **sk** diventa un intero dispari che usa  $\eta$  bits scelti nell'intervallo  $[2^{\eta-1}, 2^\eta]$
- la chiave pubblica **pk** è il prodotto di  $p \cdot q$ , dove  $q \in [0, \frac{2^\gamma}{p}]$
- se cifriamo dei dati, facciamo **encryption**(**pk**,  $m$ ), che restituisce  $c \leftarrow m + p \cdot q + 2r$  dove  $r$  è chiamato "noise", ( $r \in (-2^\rho, 2^\rho)$ ) e quindi prima di fare l'operazione si aggiunge del rumore ai dati. in questo modo non si può fare l'operazione troppe volte altrimenti la noise diventerebbe troppa
- la decifrazione è **decrypt**(**sk**,  $c_i$ ) e ritorna  $m_f \leftarrow (c_f \bmod p) \bmod 2$
- abbiamo poi **evaluate**(**pk**,  $f, c_1, \dots, c_t$ ) che applica  $f$  ai ciphertexts.

per quanto riguarda il funzionamento, se semplicemente si sommano due ciphertext si raddoppia il fattore  $r$ :

$$c_1 + c_2 = (m_1 + m_2) + p(q_1 + q_2) + 2(r_1 + r_2) \quad (5.3)$$

mentre se si moltiplicano i due ciphertext

$$c_1 \cdot c_2 = (m_1 \cdot m_2) + p(q_1 c_2 + q_2 c_1 - p q_1 q_2) + 2(2 r_1 r_2 + r_1 m_2 + r_2 m_1) \quad (5.4)$$

otteniamo che l'ultimo membro diventa quadratico rispetto alla noise e quindi ancora peggio, si raggiunge facilmente il punto in cui non si riesce più a decifrare.

se il polinomio quindi ha un grado troppo alto, il fattore gioca un ruolo fondamentale e rende il ciphertext non decifrabile, quindi non si possono fare molte addizioni o moltiplicazioni.

### leveld fully homomorphic crypto

trasformiamo lo schema di prima, questo schema è organizzato in livelli ed ogni livello computa una delle parti della **evaluate**.

ogni keygen genera una sequenza di coppie di chiavi pubbliche e private:  $\langle \mathbf{pk}_1, \dots, \mathbf{pk}_{k+1} \rangle$  e  $\langle \mathbf{sk}_1, \dots, \mathbf{sk}_{k+1} \rangle$  e poi:

- si cifrano le chiavi segrete usando le chiavi pubbliche: si ottiene  $\langle \bar{\mathbf{sk}}_1, \dots, \bar{\mathbf{sk}}_{k+1} \rangle$  dove  $\bar{\mathbf{sk}}_i = \mathbf{encryption}(\mathbf{pk}_{i+1}, \mathbf{sk}_i)$ . on the fly, si possono anche decifrare.
- stiamo quindi implementando la **evaluate** come una funzione a livelli ed ogni passo stiamo decifrando i dati e ri-valutandoli, il che fa sì che l'operazione divenga molto lenta:

$$c_{i+1} \leftarrow \mathbf{evaluate}(\mathbf{pk}_{i+1}, \mathbf{decrypt}, \bar{\mathbf{sk}}_i, c_i) \quad (5.5)$$

- possiamo quindi ad ogni passo rimuovere la noise

### private set intersection

è una primitiva di computazione, molto usata nelle applicazioni odierne, che è una sorta di secure multiparty computation: abbiamo due aziende, ad esempio l'agenzia delle tasse e la polizia che vogliono fare un controllo incrociato e vogliamo che sia garantita la privacy, vogliamo solo computare l'intersezione dei dataset senza decifrare i dati, così che ognuno dei due party sappia quali parti del dataset è anche nell'altro.

in generale

- abbiamo un sender ed un receiver
- il sender ha un dataset  $s$  di taglia  $n_s$ , il receiver ha  $s'$  di taglia  $n_{s'}$  e supponiamo che siano fatti di stringhe di bit di taglia fissa,  $\sigma$ ;
- i valori  $n_s, n_{s'}$  e  $\sigma$  sono pubblici;
- abbiamo una fase di setup: entrambe i party devono concordare su uno schema di cifratura omeomorfa. il receiver genera un keypair, tenendole segrete. successivamente, manda i dati cifrati al sender: manderà  $n_s$  ciphertexts  $(c_1, \dots, c_{n_s})$ ;
- vi è poi la fase di **intersection computation**, dove per ogni  $c_i$ , il sender genera un valore  $r_i$  random diverso da 0 ed effettua la computazione omeomorfa di

$$d_i = r_i \cdot \prod_{s' \in s'} \lim (c_i - s) \quad (5.6)$$

il ciphertext ricevuto è sottratto omeomorficamente da un elemento dell'insieme.

facendolo per ogni elemento, solo se il cipher è lo stesso dell'insieme del sender, il risultato è 0, questo non implica che nel mondo dell'encryption il risultato sia 0, quindi il sender non conosce i valori che sono nell'intersezione. il valore della sottrazione sarà 0 se e solo se i due valori sono uguali. si moltiplica per un valore random in quanto se i due elementi non sono uguali, si aggiunge della noise che solo il sender conosce e quindi anche lui non può conoscere nulla sugli elementi del dataset del sender se gli elementi non sono uguali

- si manda il risultato al receiver, che a questo punto può calcolare l'intersezione, tramite



- il receiver decifra  $(d_1, \dots, d_{n'_s})$  e calcola:

$$i = s \cap s' = \{s_i, \text{decrypt}(\mathbf{sk}, d_i) = 0\} \quad (5.7)$$

e quindi ottiene tutti gli elementi per cui il valore della sottrazione è 0.

nessuno dei due conosce l'intersezione, siamo quindi in grado di forzare la privacy permettendo a dei parties di scambiarsi pezzi di dati.

ci sono delle tecniche in cui si può diminuire la privacy, ad esempio aggiungendo ad  $s$  più elementi, così da far sì che l'intersezione con  $s'$  prima tutto  $s'$ , ma occorre conoscere qualcosa dell'altro party oppure si può far girare l'algoritmo più volte e poi fare una analisi di tutte le intersezioni.

lo schema si chiama tipicamente "honest but curious", quindi non è un infame bastardo che vuole fregarti ma se gira più volte l'algoritmo può scoprire qualcosa.

## and that's all folks !!