25/09/2023 (lez.1)

Introduzione:

E' impossibile rendere un sistema sicuro, qualsiasi cosa può essere usata per portare avanti un attacco ed un attacco può venire da chiunque.
Si possono avere attacchi ad ogni livello dello stack, anche quando si compila del codice sorgente.
Le applicazioni full stack sono anche peggio, perché importano librerie esterne scaricate da internet.
Si può solo fare del nostro meglio per alzare l'asticella della sicurezza.
Non si può essere sviluppatori full stack nella sicurezza perchè si dovrebbe conoscere al meglio ogni livello dello stack IT.
Inoltre la sicurezza deve interfacciarsi anche con dei compromessi: tempo di sviluppo e di debugging, costo di sviluppo, vendor lock in e manutenibilità.
Non c'è un trade off corretto, dipende dal contesto in cui si lavora.

Rationale behind security:

Vogliamo che il nostro sistema sia utilizzabile solo dagli utenti autorizzati, si fa questo tramite meccanismi di autorizzazione, messi in piedi dagli amministratori di sistema.
Un attaccante potrebbe rendere un sistema inutilizzabile dagli utenti legittimi.

Principi fondamentali di sicurezza e privacy:

Un falso senso di sicurezza è molto pericoloso. Si deve prestare attenzione alla sicurezza ad ogni livello dello stack e sia alle persone interne che esterne al sistema da proteggere.
Il cloud computing ha reso la vita degli attaccanti più facile, perché possono spawnare molte VMs per performare attacchi brute force.
Ci sono principi di sicurezza di base, come cambiare le proprie password, aggiornare il proprio antivirus e usare firewalls, ma un approccio alla sicurezza di questo genere è un approccio naive, per aumentare la sicurezza si deve: guardare al sistema da un punto di vista del sistema e dal punto di vista degli esseri umani (utenti interni o esterni al sistema).
Ci sono degli ingredienti per alzare l'asticella della sicurezza: crittografia, autenticazione e sistemi operativi orientati alla sicurezza.

Principle of least privileges→ad un certo livello dello stack un utente deve avere accesso solo alle informazioni necessarie.
Ad esempio accedere ad un DB come root viola questo principio, anche se per l'amministratore è più semplice perché deve gestire una sola connessione.
Questo principio si applica ad ogni livello.
Esso conferisce: maggiore stabilità (riduce le azioni da testare), maggiore sicurezza (le vulnerabilità non possono essere sfruttate per attaccare il resto del sistema) e facilità di sviluppo.
Need to know principle→ se in un certo momento non si ha bisogno di accedere ad una certa informazione non si deve poter accedere
Se un utente ha bisogno di accedere a due sistemi in momenti diversi deve avere due profili diversi.
Questo risolve alcuni problemi di ingegneria sociale.

Cifratura: si cerca di oscurare i dati. Bisogna immaginare che eventualmente il sistema possa essere "rotto" e qualcuno possa accedere ai nostri dati. E' probabilisticamente sicuro,

computazionalmente non fattibile. Gli algoritmi di cifratura e decifratura sono noti, la sicurezza sta nelle chiavi.

CIA→ confidentiality:non puoi accedere dati se non ti è permesso, integrity, non puoi alterare dati se non ti è permesso and availability, qualsiasi cosa accada ad un sistema deve rimanere utilizzabile dagli utenti.
Dependability→ sistema disponibile, affidabile,sicuro,confidenziale, integro e manutenibile.

Altri obiettivi di sicurezza sono: autenticità (autenticazione + non repudiation), controllo degli accessi (autorizzazione) e "accountability" (tenere traccia delle azioni di un'entità e memorizzarle in un file di log).

09/10/2023 (lez.7)
Dependability significa avere un sistema sicuro da attacchi interni ed esterni.
Unico principio: prevedere ciò che può avvenire in futuro; è poco applicato dal punto di vista della sicurezza, mentre andrebbe considerato sin dall'inizio.
"Accountability" è fondamentale per garantire la sicurezza a posteriori; bisogna pensare che ad un certo punto le cose andranno male, quindi è utile tenere traccia di ciò che è accaduto: come e perché è avvenuto un attacco. Ciò è utile anche perchè se un sistema è vulnerabile a qualche tipo di attacco anche altri sistemi lo saranno allo stesso modo.
Tutto ciò che è stato detto finore è di carattere generale, vedremo come applicarlo a diversi livelli dello stack IT.

Security frameworks:
Un modo per aiutare le persone ad aumentare la sicurezza dei loro sistemi.
Ci sono concetti di base che possono essere standardizzati.
I security frameworks offrono dei controlli standardizzati per rafforzare i sistemi, una sorta di checklist.

FIPS 2000→ tentativo di raccogliere un insieme minimale di principi di sicurezza richiesti.
Alza l'asticella ad un minimo livello accettabile.

- Access control→per limitare l'accesso ai soli utenti autorizzati
- Aspetti relativi agli utenti in quanto esseri umani, quindi hanno a che fare con la consapevolezza e l'apprendimento. Si vuole che gli utenti utilizzino il sistema in maniera da conoscerne i rischi
- Revisione e "accountability" → se qualcosa va storto permette di capire quali parti del sistema sono coinvolte
- Certificazione, accredito e valutazione di sicurezza. La sicurezza non è statica, per ridurre l'insorgere di nuove vulnerabilità bisogna monitorare il sistema continuamente
- Contingency planning→soprattutto in grandi compagnie è difficile ricordare tutte le macchine che si possiedono, quindi bisogna tenerne traccia, in modo tale che se in parte del sistema si scopre una vulnerabilità essa non vada ad impattare altre parti del sistema. Bisogna avere checklist, copie, backups per fronteggiare incidenti. Spesso, soprattutto nelle piccole compagnie, si ignorano i backups, visti come un costo aggiuntivo, ma quando servono è troppo tardi
- Identificazione e autenticazione
- Risposta agli incidenti→ legata al contingency planning e accountability, bisogna tenere traccia degli incidenti con documentazione

- Manutenibilità→ legata al fatto che la sicurezza non è statica. Un sistema sicuro è un sistema “accountable” e di conseguenza manutenibile. Bisogna prevedere che qualcosa in futuro si romperà. Ha a che fare sia con il sw che con l’hw.

CIS→sono irganizzati in controlli generici e controlli applicati a situazioni pratiche. Sono semplici da applicare, ci sono diversi controlli a seconda delle aziende.
Sono suddivisi in diversi livelli, i primi 5 sono:

- ***CSC 1:*** Inventory of Authorized and Unauthorized Devices
- ***CSC 2:*** Inventory of Authorized and Unauthorized Software
- ***CSC 3:*** Secure Configurations for Hardware and Software on Mobile Devices, Laptops, Workstations, and Servers
- ***CSC 4:*** Continuous Vulnerability Assessment and Remediation
- ***CSC 5:*** Controlled Use of Administrative Privileges

CSC3 è suddiviso in diversi dispositivi perchè diversi tipi di utenti sono collegati a diversi tipi di dispositivi.
I controlli dal livello 6 in su non sono importanti come i primi 5, ma sono più specifici.

ISO/IEC 2700→E’ una famiglia di standard. Si ottiene una certificazione da terze parti: il sistema è sicuro secondo una serie di principi.
Sono nati nel 2018 e continuano ad essere aggiornati.
In ciascun livello dello standard ci sono controlli addizionali, per alcuni livelli bisogna pagare.
In base all’azienda si devono avere alcuni requisiti di questo standard.

Hardware Primer:
La legge di Moore è una legge empirica, fino al 2003 abbiamo assistito ad una crescita che seguiva questa legge, poi non più a causa del “Power wall”: aumentando la frequenza di lavoro della CPU la potenza da dover dissipare sarebbe aumentata troppo così da mandare a fuoco le CPU.
Per reagire a ciò abbiamo un cambiamento nelle CPU: il single core era troppo lento, si è passati quindi al multicore: sono stati realizzati componenti più piccoli che potessero coesistere nella stessa CPU. Questo ha aumentato il conflitto tra istruzioni, quindi è stato introdotto lo scheduling a runtime; le CPU riorganizzano l’esecuzione delle istruzioni in modo da utilizzare meglio l’hardware sottostante.

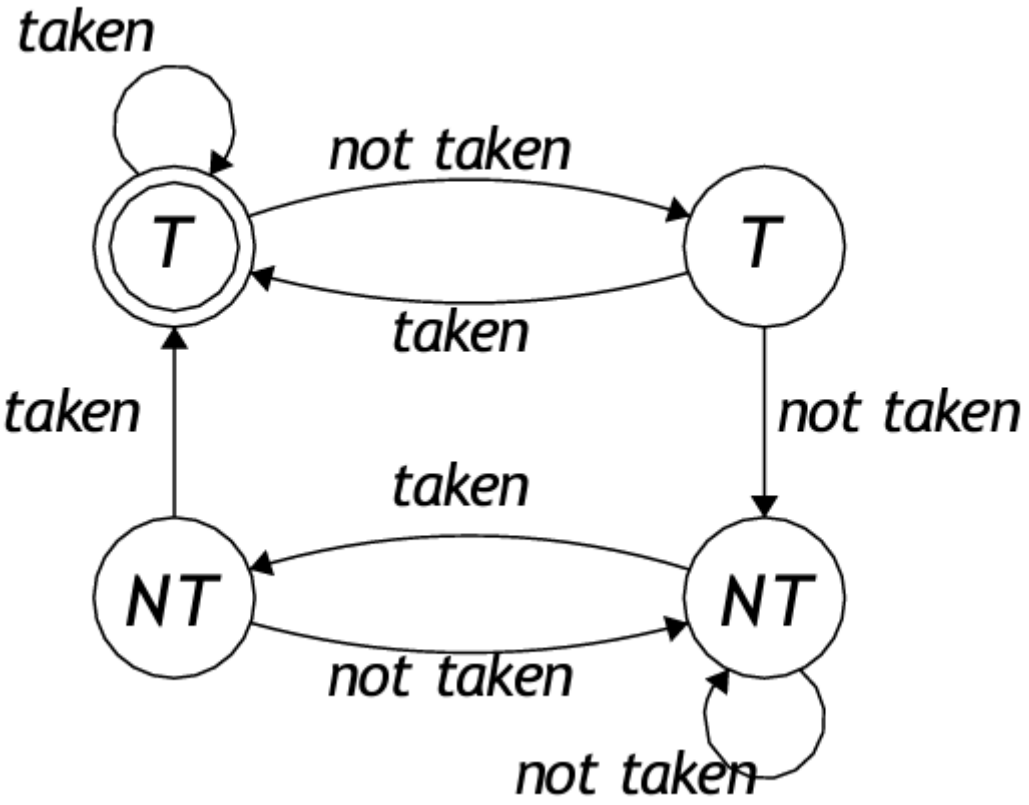
16/10/2023 (lez.10)
Speculazione: la CPU tenta di indovinare la prossima istruzione da mettere in pipeline. Nel caso in cui la decisione presa sia sbagliata flush della pipeline, che può essere costosa.
Se usassimo gli stalli il costo sarebbe troppo elevato. Il costo dipende dal numero di salti (che sono dappertutto) e dal numero di cicli da attendere (il numero di cicli corrisponde al numero di stages della pipeline).
Grazie alla speculazione se l’assunzione è giusta non c’è alcun costo da pagare, nessun tempo da attendere, se è sbagliata si paga il costo del flush della pipeline (costo di aggiungere bolle alla pipeline).
La miglioria delle prestazioni dipende da diversi fattori, tra cui:

Correttezza della Predizione: L'accuratezza della predizione del salto condizionale è fondamentale per migliorare le prestazioni. Se la predizione è corretta, il processore può continuare ad eseguire istruzioni senza interruzioni.
Tempismo nella Verifica della Predizione: Il momento in cui è possibile verificare la predizione influisce sulle prestazioni complessive. Ridurre il tempo necessario per controllare la predizione consente una maggiore efficienza nell'esecuzione delle istruzioni.

La predizione del salto può essere di due categorie:
Predizione Dinamica del Salto: Questo approccio prevede che la predizione del salto cambi in base al comportamento del programma. Solitamente implementato a livello hardware, si basa spesso sulla storia dei salti passati, prevedendo se il salto verrà preso in base alle occorrenze precedenti.
Predizione Statica del Salto: In questo caso, la predizione è determinata a livello di compilazione del codice. Può essere assistita dall'utente, ad esempio, tramite annotazioni nel codice sorgente del kernel, come nel caso dei prefissi 0x2e, 0x3e per il Pentium 4.
L'obiettivo principale è migliorare l'esecuzione del programma prevedendo in modo accurato le istruzioni di salto, sia in modo dinamico che statico, al fine di ottimizzare le prestazioni complessive del sistema.

Per far sì che le decisioni prese siano corrette si utilizza la “Branch Prediction Unit”.
Branch prediction table:
Piccola memoria indicizzata dai bit meno significativi dell'indirizzo dell'istruzione di salto condizionale.
Ogni istruzione è associata a una previsione: eseguire o non eseguire il salto.
In pipeline abbiamo il fetch della prossima istruzione, nella fase di decode della precedente capisce se deve fare o meno il salto: se la decisione è di farlo dobbiamo comunque pagare un costo perché la prossima istruzione non è quella da eseguire, quindi penalità di un solo ciclo. Se la previsione è di non eseguire il salto ed è corretta: nessuna penalità.
Se la previsione è errata: modificare la previsione, flush della pipeline; la penalità è la stessa come se non si fosse previsto il salto.
Inoltre se la predizione è sbagliata noi la possiamo modificare anche per gli altri salti, quindi stiamo imparando anche qualcosa relativo ad altri salti; mentre se è giusta stiamo rafforzando la nostra conoscenza, quindi impariamo sempre qualcosa.

Two bit saturating counter:
Possiamo usare una FSM. Decidiamo per ogni istruzione basandosi sulle decisioni prese per ogni altro salto, quindi prendiamo in considerazione tutte le istruzioni. Abbiamo due stati: Salto “preso” o “non preso”. Le transizioni di stato sono date dal fatto che si faccia o meno il salto. Lo stato non cambia se la predizione era giusta. Se la predizione è sbagliata si cambia stato, ma sempre in stato con stesse caratteristiche, perché cambiare predizione una sola volta non è indicativo del fatto che tutte le successive predizioni debbano cambiare.
Nel caso di cicli annidati questa FSM oscilla tra i due stati “T”.
Il problema è che qui creiamo una correlazione inesistente, perché vogliamo imparare il comportamento di un’istruzione da quello di un’altra.



Il 20% delle istruzioni un programma è un salto, quindi la predizione dei salti è molto importante. Anche nella programmazione object oriented ci sono molti salti, nascosti dall'ereditarietà, infatti per trovare un metodo reimplementato da un oggetto bisogna esaminare tutta la gerarchia.
Bisogna quindi trovare modi per migliorare le prestazioni delle previsioni dei salti.
a. Migliorare la previsione:
   1. Previsioni correlate (a due livelli)
   2. Previsioni ibride locali/globali

b. Determinare il target in anticipo:
   1. Branch target buffer
   2. Prossimo indirizzo nella cache delle istruzioni
   3. Stack degli indirizzi di ritorno (si usa in tutte le architetture)

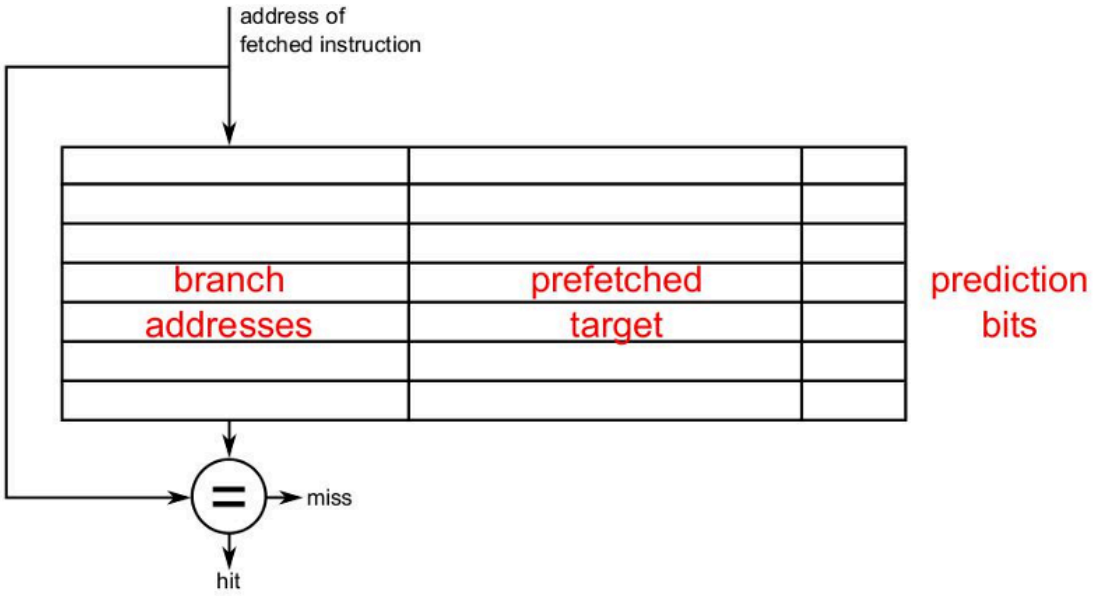
c. Ridurre la penalità per le previsioni errate: fetchare entrambi i flussi di istruzioni.

Predittori correlati:
Cerchiamo di capire quali salti sono correlati: stessa variabile nella condizione oppure una variabile modificata nel corpo di un if statement dopo essere stata usata come variabile per un altro if.
Usano una storia dei passati m salti che rappresentano un percorso all'interno del programma.

Pattern history table:
serie di FSM per predire l'outcome di un salto in base alla storia passata recentemente.
I bit provano ad identificare dei pattern nella storia dei salti, che rappresentano tracce di parti diverse del programma con comportamenti diversi.
Shiftiamo a sinistra il registri scartando i bit più vecchi quando un nuovo salto viene intrapreso o meno.
Questo meccanismo non è privo di errori, perchè ci sono istruzioni non correlate.

Tournament predictors:
Ci sono due predittori, locale e correlato. Quello locale usa l'indirizzo di una singola istruzione di salto. Quello correlato si basa sugli ultimi m salti, accedendo alla storia locale.
Un altro contatore a due bit (choice predictor) dice quale dei due funziona meglio per ciascuna istruzione.

Branch target buffer:
I salti indiretti sono difficili da predire. Sono diverse le destinazioni per questo tipo di istruzioni, per esempio "return": possiamo avere infiniti chiamanti per una certa funzione, quindi dobbiamo sapere dove ritornare.
Lo stesso vale per i jump ai registri (e.g. jmp *rax, ret).
Il branch target buffer è implementato nella CPU tramite una piccola cache che viene acceduta nella fase di fetch. Esso tiene traccia di alcuni target predetti.
Per differenziare tra i target prefetched: bit di previsione.
Non è una scelta binaria, sono possibili molteplici "Outcomes".
Può funzionare per una return? Già conosci il chiamante per una return, non c'è bisogno di statistiche. L'85% dei salti indiretti sono return.



Return address stack:
Ogni volta che si invoca una funzione si tiene conto dell'indirizzo del chiamante e si mette in cima ad uno stack. In questo modo si rende deterministico l'85% dei salti indiretti.

Branch target buffer e return address stack vengono usati insieme: se per qualche ragione il return address stack ha scartato un indirizzo di ritorno ci si può risalire tramite il branch target buffer.

Fetch both targets:
Nelle architetture superscalari si fetchano entrambi i targets.
non funziona in caso di switch cases.
Riduce le predizioni errate.
Necessita di molta banda nella cache delle istruzioni.

Nelle architetture multicore abbiamo molteplici cores nella stessa CPU fisica, si utilizza, dunque, il “multithreading simultaneo” o “hyperthreading”.
Vengono introdotti dei cores virtuali, questa è una virtualizzazione a livello HW.
Si possono condividere risorse tra diversi programmi o threads.
Lo stato dell’architettura identifica unicamente un thread, se lo si può copiare si può riusare per diversi programmi. Ne viene fuori un HW minimale, ma c’è bisogno di una qualche forma di “arbitrato”.

Scopi:

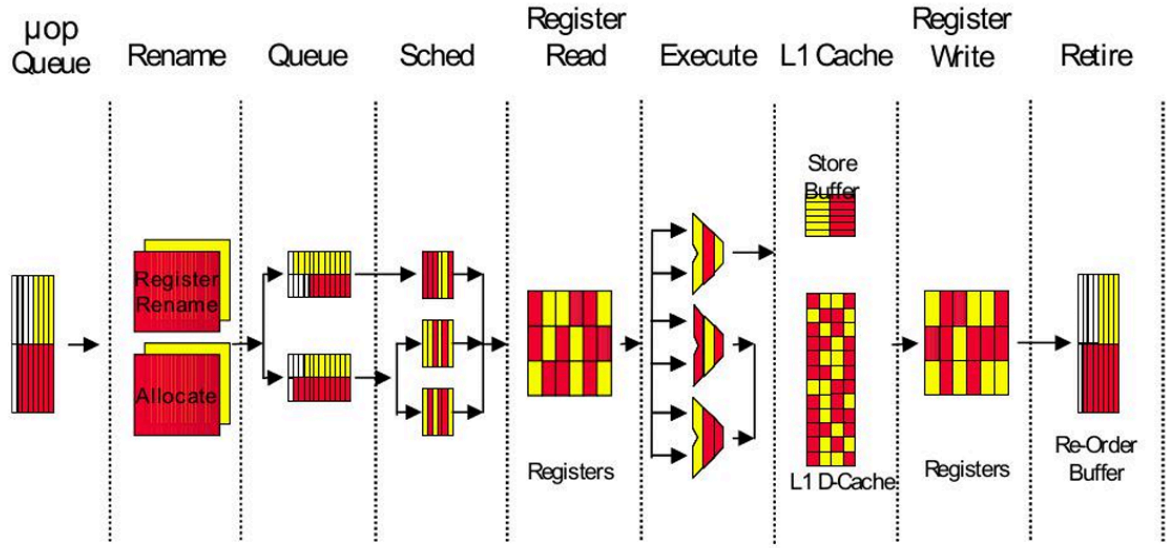
- Condividere il più possibile risorse fisiche
- Se più programmi sono in esecuzione sullo stesso HW vogliamo che le performance di uno non siano affette da quelle dell’altro
- Se si disattiva l’hyperthreading l’HW deve continuare a funzionare come se non fosse mai stato implementato

Le CPU tradizionali Intel internamente hanno un set di istruzioni ristretto, anche se le istruzioni provenienti dal compilatore non lo sono, quindi l'architettura è divisa in due parti principali: front-end e motore di esecuzione “out of order”.
Il front-end si occupa di tradurre un linguaggio dell’instruction set nell’altro, mentre il motore out of order si occupa del fatto che se alcune risorse non sono disponibili le istruzioni possono essere riordinate nella pipeline, per ridurre le attese.
La traduzione del microcodice da RISC a CISC non è semplice, quindi alcune traduzioni sono messe in cache. Ad un’istruzione RISC corrispondono molteplici istruzioni CISC.
Trace-cache: qui si tiene traccia di micro-operazioni associate a diversi programmi, quindi vengono segnate con le informazioni relative ai diversi threads.
Micro-operation-queue: anche qui le micro-operazioni sono segnate con le informazioni relative ai vari programmi, perchè si vuole che se un programma necessita di alcune micro-operazioni non abbia effetti sulle performance dell’altro.

Se c’è un miss della trace cache se le istruzioni vanno caricate dalla memoria.

L’out of order pipeline parla l’assembly RISC.
Cosa accade se due programmi vogliono usare lo stesso registro?
Avviene un conflitto, non ha senso fermarne uno; allora c’è un trucco: i registri acceduti non sono quelli fisici; per questo nella pipeline ci sono le fasi di “register rename” ed “allocate”.
I registri fisici general purpose sono circa 60 nelle architetture a 64 bit, nell’implementazione della CPU ne troviamo circa 200. Se accediamo ad un registro esso è una label che verrà associata ad un registro fisico.

Nel primo stage di rename abbiamo il rename register che tiene traccia delle allocazioni fisiche correntemente associate ad un flusso di esecuzione. Nella fase “allocate” guarda alle microoperazioni e alloca una delle componenti disponibili.



Scheduler: legge la micro operazione e se la risorsa richiesta non è disponibile ritarda la mop: è qui che avviene il riordinamento. Esso interagisce con le risorse fisiche e vede se sono disponibili.

Questi 4 stages sono simili a quelli della pipeline tradizionale.
Store buffer: mantiene le operazioni di “store”, dal punto di vista della CPU una micro-operazione che tenta di scrivere un valore in memoria.
Questo serve perchè l’accesso in memoria è più lento rispetto alla velocità della CPU quindi anziché scrivere di volta in volta in memoria si va ad utilizzare lo store buffer.
Dato che posso avere operazioni memory intensive non ha senso mettere in stallo la pipeline.
Se si hanno diversi cores, gli outcomes di diverse operazioni di scrittura, provenienti dai diversi cores, possono andare nello store buffer in ordine differente, alla fine avviene il riordinamento, nella fase di “retire”, quindi il compilatore non se ne cura.
Questo velocizza le performance, ma c’è un bug latente in questo design.

18/10/2023 (lez.12)
Gerarchia di Cache:
Il gap tra la velocità della CPU e della memoria aumenta nel tempo, questo rallenta anche la CPU, perchè essa ha bisogno di interagire con la memoria per fare calcoli.
La soluzione ottimale per un sistema di memoria sarebbe: costo minimo, massima capacità e minimo accesso in termini di tempo; una soluzione approssimata è quella della gerarchia di memoria.
Se si volesse realizzare una memoria super veloce bisognerebbe investire molti soldi, la soluzione a ciò è la gerarchia di memoria.
La CPU non vede direttamente la memoria ma dei livelli di cache.
La CPU parla con il primo livello di cache.

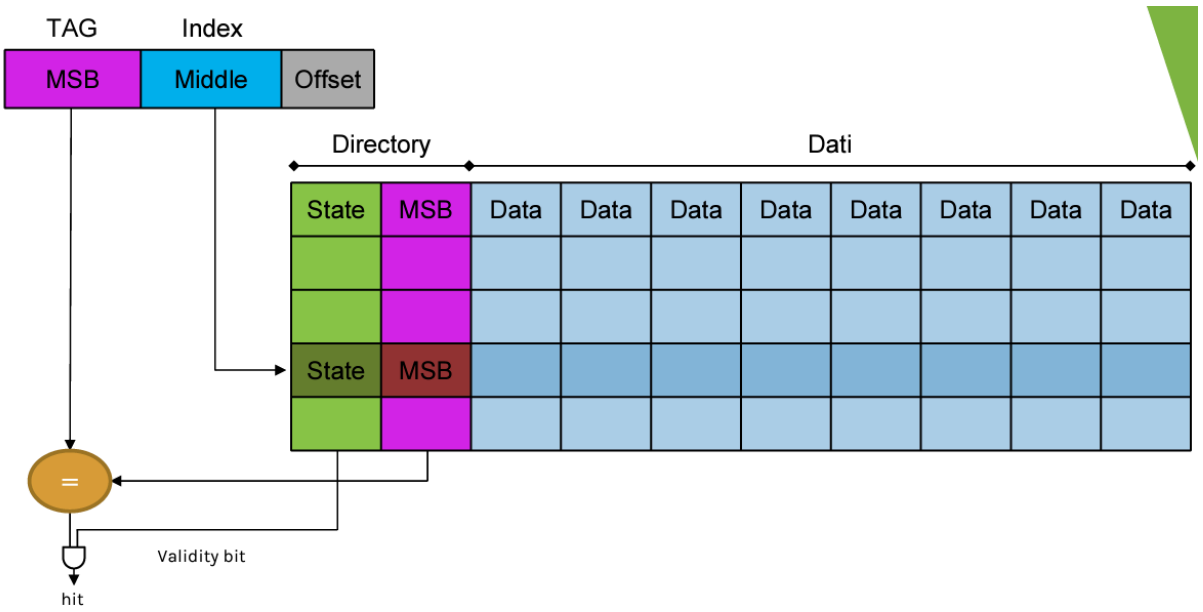
Ci sono 3 ordini di grandezza tra la velocità della memoria e le capabilities della CPU.
Dobbiamo far sì che la CPU trovi i dati di cui ha bisogno nella gerarchia di memoria quando ne ha bisogno.
All'aumentare della distanza dalla CPU il costo diminuisce e la capacità di archiviazione aumenta; in ogni istante di tempo i dati vengono copiati solo tra ogni coppia di strati adiacenti.
Organizzare questa gerarchia a livello di bytes sarebbe complicato, per questo viene organizzata a livello di linee di cache.
Linea= minimo ammontare di dati gestibile da una cach.
Blocco=dati che si inseriscono in una linea, è la più piccola informazione che si può scambiare tra due livelli della gerarchia.
Ogni livello fa una copia del dato dalla memoria principale e poi inoltra il dato al livello superiore.
Se si trova il dato in qualche livello della cache si parla di CACHE HIT, altrimenti se ad un certo livello non si trova il dato parliamo di CACHE MISS.
La granularità è superiore ad 1B perchè probabilmente la CPU accederà più di un dato.

Politiche:

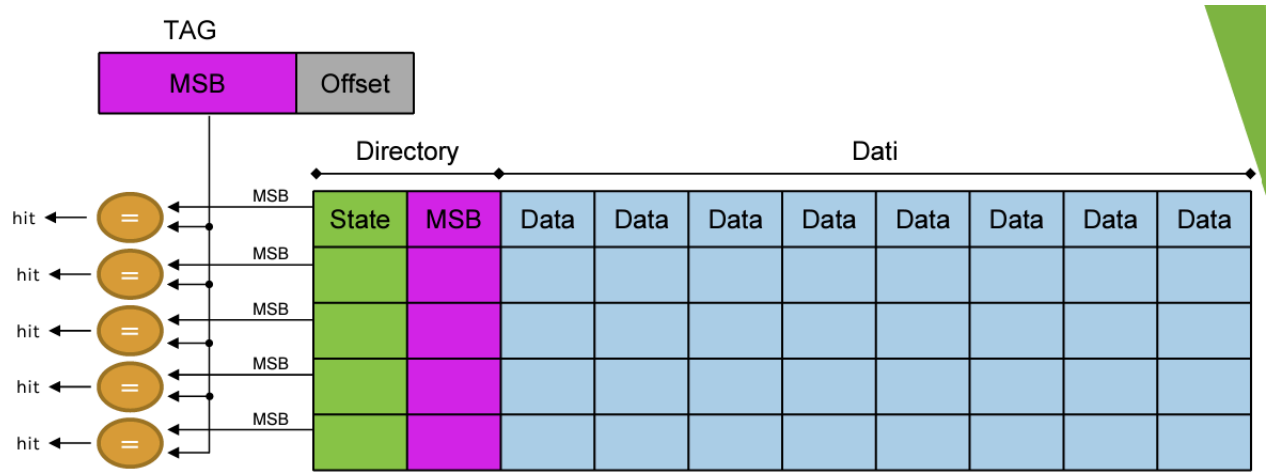
1. Cache inclusive → I dati disponibili ad un livello più alto sono disponibili anche nel livelli più bassi
2. Cache non inclusive → Alcuni o tutti i livello possono essere non inclusivi. Un miss ai livelli superiori può risultare in un accesso diretto alla memoria.

Oggi le cache sono non inclusive, con alcuni livelli inclusivi ed altri no.
Si deve definire un mapping tra un indirizzo di memoria ed una linea di cache. in una linea c'è più di un blocco perchè le linee di cache sono più piccole della memoria, diversi indirizzi in memoria possono collidere nello stesso blocco di cache.
Dobbiamo trovare un modo per distinguere i sinonimi.

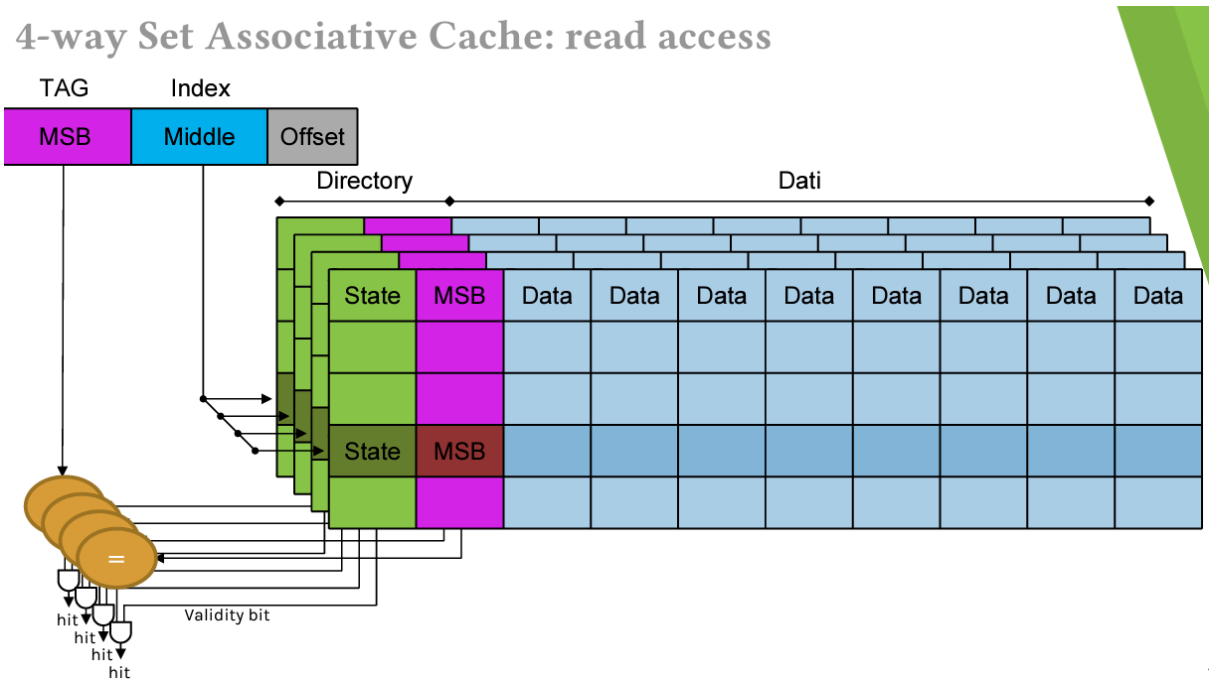
Direct mapped cache:
Approccio basato su tag
I bits di mezzo sono usati per identificare una linea di cache.
Un blocco sono 64B, quindi in una linea ci sono 64B.
Dobbiamo dunque scartare gli ultimi 6 bits.
I primi 10 bit sono usati come tag, perché i bits più significativi possono distinguere tra due indirizzi che collidono. I 16 bits di mezzo sono usati come indice per la cache line.
Se il tag non matcha abbiamo un cache miss.
Bit di stato→ Danno ulteriori informazioni sullo stato di una linea di cache. Quando si avvia una macchina la cache è totalmente riempita di 0, che rappresentano un tag valido, quindi abbiamo bisogno di informazioni di stato che dicano se un blocco è valido o meno (se può essere usato o no). Necessitiamo di un bit.
Questo schema non può funzionare perchè se si leggono due sinonimi alternativamente si avranno una serie di miss.



Cache completamente associativa:
Si può memorizzare un blocco in ogni linea di cache.
Pro: massima utilizzazione della cache e riduzione dei conflitti
Contro: per trovare qualcosa bisogna navigare in tutta la cache. Per velocizzare si fa ricerca in parallelo inserendo un comparatore per ogni linea, ma risulta costoso.



Cache set-associative:
Implementazione utilizzata attualmente.
Tradeoff tra cache direttamente mappate (troppi conflitti) e cache completamente associative (troppo costose).
Ogni blocco può essere inserito in un certo numero (>=2) di righe di cache (set).
Servono dei comparatori, il numero di comparatore dipende dal numero di righe in ciascun set.
Si riducono i conflitti e si ha un migliore utilizzo della cache.



Eviction policies:
Quando più di un'entry della cache è disponibile bisogna selezionare una vittima da rimpiazzare.
Ci sono diverse politiche:
LRU→Nei bit di stato possiamo memorizzare ulteriori informazioni, ad esempio un contatore sulle operazioni eseguite su una linea di cache. Ogni volta che si carica un blocco si resetta l' "age counter", ogni volta che si accede un blocco si incrementa. Questa policy cerca di catturare la località temporale. Vuole catturare il fatto che un dato sia acceduto ripetutamente.
Round robin→Si tiene traccia di un punto della linea seguente da rimpiazzare. Questo non cattura informazioni riguardo il comportamento di un programma.
FIFO→simile a LRU ma l'incremento avviene solo a valle di un rimpiazzo
LFU→ Difficile da implementare
Random→simile a round robin

Gestione delle operazioni di scrittura:
Cosa succede quando si scrive in memoria? Si scrive il dato in un blocco, quindi si hanno dei "mismatch" a diversi livelli della cache, ovvero per la stessa variabile allo stesso istante di tempo si hanno valori diversi.
Per supportare gli aggiornamenti della memoria ci sono due strategie:
write through→Ogni volta che aggiorni un livello fai una copia sugli altri livelli. Le performance sono più basse ma è semplice da implementare. In questo modo si ha il costo più elevato per un'operazione di write.
write back→Si propagano gli aggiornamenti quando si hanno dei replacement. L'implementazione è più complessa perché si deve tenere traccia del livello che mantiene la copia più recente, però le performance sono migliori.

Multi-cores:

Ogni core di CPU ha la propria cache L1, si possono avere anche cache multiple di livelli inferiori.
La strategia del write back risulta più complicata. L'HW ad un certo punto deve interfacciarsi con dei problemi, dunque vengono sviluppati i protocolli di "cache coherence".

Protocolli di cache coherence:
Strong coherence → La cache lavora come se ci fosse una singola esecuzione su di essa.
L'HW implementa un ordinamento totale sia sulle operazioni di lettura che di scrittura.
Tutte le cache si devono sincronizzare su ciascun core.
Soluzione molto costosa.

1. Single writer multiple readers → Un solo core può performare una store ad un certo istante di tempo, nessun altra nella stessa epoca.
   Se più cores stanno cercando di aggiornare una variabile allo stesso tempo si deve permettere ad un solo core per epoca. Le operazioni di lettura possono essere più di una per volta.

2. Data value invariant→ Il valore mantenuto in una variabile all'inizio di un'epoca è lo stesso che alla fine dell'epoca.
   Se qualcuno sta aggiornando una variabile non si possono permettere delle operazioni di lettura finché tutte le cache non hanno il valore aggiornato. E' permessa una sola operazione di scrittura e si ritardano le operazioni di lettura ad un momento in cui gli aggiornamenti sono terminati (propagati a tutti i livelli di cache).

   Il costo di implementare strong coherence a livello HW è elevatissimo.
   La soluzione a questo che abbiamo nelle nostre architetture HW è di rilassare i vincoli.

Weak coherence:
Si può permettere di fare diverse scritture allo stesso tempo, sta al programmatore sincronizzare rispetto al valore corretto, è il sw che prende questa decisione.

L'invariante SWMR potrebbe essere completamente eliminata: più cores potrebbero scrivere nella stessa posizione A, un core potrebbe leggere A mentre un altro core sta scrivendo su di esso.
L'invariante DV potrebbe valere solo eventualmente: si garantisce che le memorizzazioni si propaghino a tutti i core in d epoche, i caricamenti vedono l'ultimo valore scritto solo dopo d epoche.
Se si ha bisogno di sincronizzazione a livello sw si usano mutex o semafori.
Si possono propagare i valori aggiornati di una variabile dopo un certo tempo, ad esempio dopo un certo numero di letture.
Scrivere un programma concorrente è difficile perché bisogna interfacciarsi con specifiche implementazioni dell'hardware.

No coherence:
E' la cache più veloce, tutti possono scrivere e leggere qualsiasi cosa, no garanzie su ciò che andremo a leggere/scrivere.

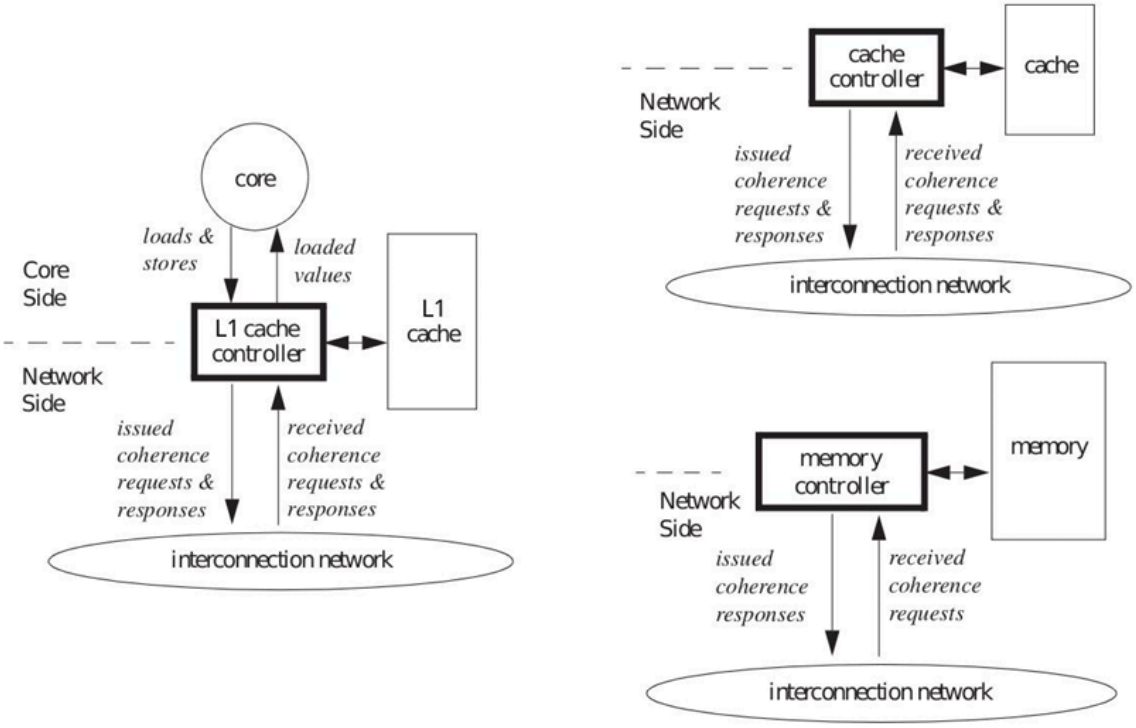
Tutto sta nelle mani del programmatore, che deve occuparsi della concorrenza esplicitamente.

23/20/2023 (lez.13)
CC protocols:
Solitamente i protocolli di coerenza sono distribuiti. Implementano un sistema distribuito all'interno della CPU. C'è uno scambio di messaggi tra i diversi attori HW nel sistema.
Cache controller→Ogni cache level ha un cache controller, esso è una parte dell'HW che genera eventi nel sistema, riceve messaggi su eventi, notifica agli attori cosa i cores chiedono di fare e prende decisioni su cosa fare nelle linee di cache in base a ciò che avviene nel sistema.
Servono solo due richieste di memoria: load e store.
I messaggi tra gli attori vengono scambiati viaggiando su un bus.
Questo tipo di protocollo può essere difficile da realizzare perché in caso di strong coherence si deve garantire un ordinamento totale.
Spesso gli scambi di messaggi sono incapsulati in "coherence transitions".
Le richieste load store da cpu core vengono trasformate in queste transazioni che vengono scambiate tra i diversi coherence controllers.
A seconda dello stato delle linee di cache possiamo essere in stato in cui blocco può essere caricato o meno, può anche essere stato "evicted", quindi dobbiamo pensare a due tipi di transazioni get e put.
Get→ il cache controller vuole caricare un blocco da una linea di cache
Put→il cache controller vuole togliere un blocco da una linea di cache.
Queste due transazioni sono sufficienti a caricare e eliminare blocchi.

Astrazioni dei diversi attori:
Immaginiamo una rete di interconnessione. Abbiamo due lati: core e network side. Ciascun core ha la propria cache L1 e per interagirvi esso deve comunicare con L1 controller.
Se qualche valore non si trova nella cache L1 il controller L1 interagisce con i livelli inferiori ed infine manda la risposta al core.
Dal lato della rete la rete di interconnessione è condivisa tra gli altri attori. Il dato di cui ha bisogno un core può essere posseduto da qualcun altro nel sistema, chi lo ha risponde e lo manda a colui che ha fatto la richiesta di load.
il memory controller non chiede blocchi perchè in memoria c'è tutto.
Tutti devono conoscere globalmente lo stato delle copie. Soltanto colui che ha aggiornato il valore deve rispondere alla richiesta cache coherence.
Nella directory della cache ci sono dei bit che tengono traccia dello stato e del contenuto del blocco.

FSM:

Cache and Memory Controllers



Un'astrazione per tenere traccia dell'evoluzione temporale del sistema è la FSM.
FSM: set di porte logiche che permettono ad un cache controller di conoscere lo stato di tutte le linee di cache.
Descrivono lo stato di ciascuna copia del blocco. Le transizioni da uno stato all'altro avvengono in base a degli eventi. Si può generare un output tramite le azioni.

Stati:

Stable state: all'inizio e alla fine di una transazione
Transient state: nel mezzo di una transazione; ad esempio se un blocco è stato richiesto ma non ancora ricevuto

Eventi:

Eventi remoti: qualche controller sta osservando sul bus le richieste fatte da lui stesso aspettando un coherence message
Eventi locali: emessi dal controller della cache primaria

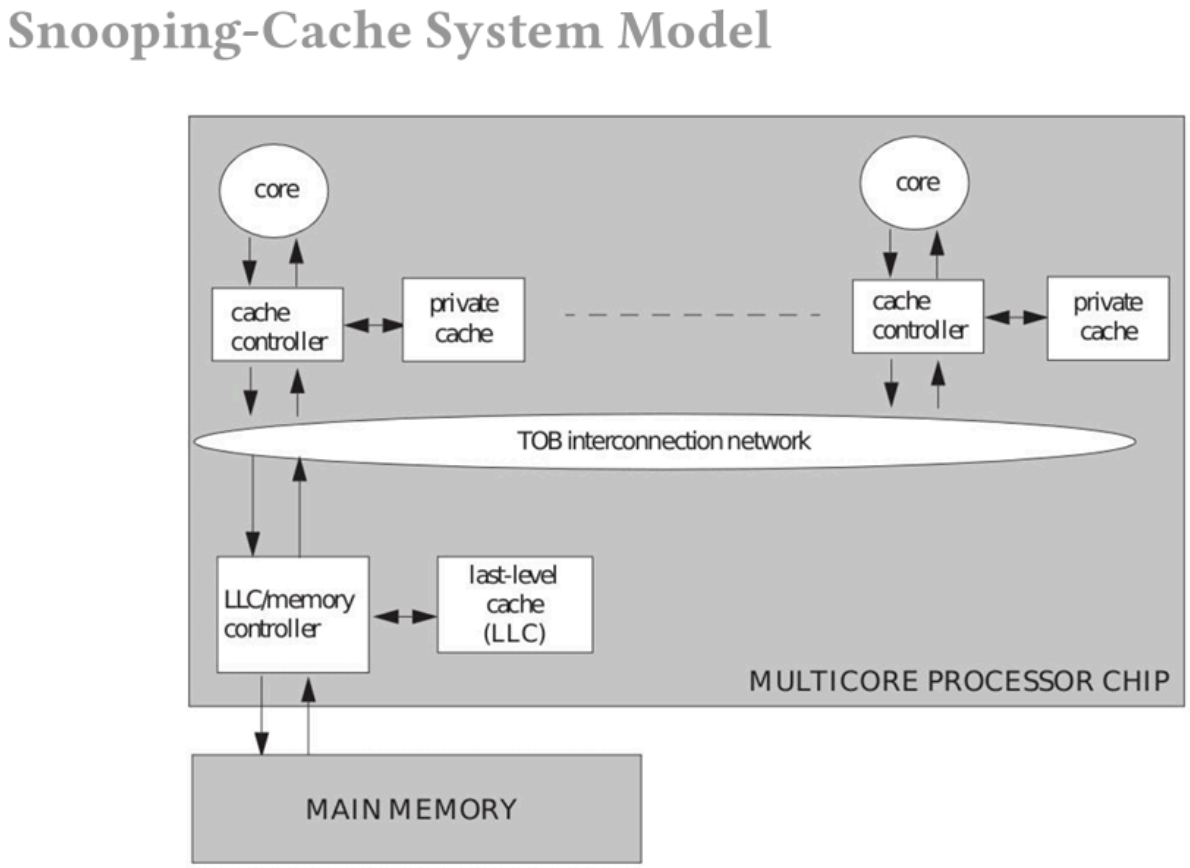
Azioni:

Azioni remote: producono la richiesta di un CC message
Azioni locali: sono visibili solo al controller della cache primaria.

Famiglie di coherence protocols:

1. Invalidate: abbiamo un blocco nella cache L1, lo vogliamo modificare, prima di fare l'aggiornamento chiediamo che chi ha copie di questo blocco vada ad invalidare la relativa linea di cache, quindi al bisogno dovranno ricaricare il blocco. Solo chi aggiorna ha la copia aggiornata del blocco.
2. Update: quando aggiorniamo un blocco dobbiamo propagare l'aggiornamento a tutti gli altri. Ciò comporta un grande consumo di banda, infatti molti aggiornamenti implicano molti messaggi di updates. Ogni update viene mandato in broadcast.

3. Snooping cache:tutti i cache controllers vedono tutti i messaggi di cache coherence essendo interconnessi da una rete. Questa soluzione non è scalabile, tutti devono vedere lo stesso ordine di eventi. Sulla rete c'è bisogno di un'ulteriore componente HW che svolga il ruolo di arbitro. Si parla di "total order broadcast interconnect network". Solo un controller per volta può accedere il bus, stiamo riducendo il parallelismo negli accessi. Anche se due cercano locazioni di memoria diverse non possono farlo parallelamente.



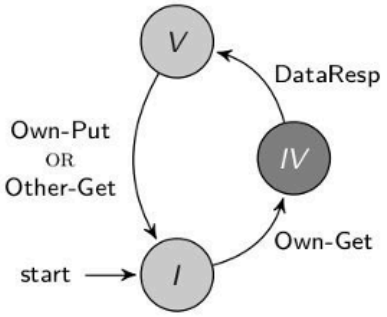
4.
5. Directory based: non ci sono assunzioni di interconnessione. Bus=routers, packet switching,... Ci sono diverse directories che indirizzano diverse regioni dell'address space. L'ordinamento totale si ha solo a livello delle directories. Scalabile, ma più lento. La scalabilità vince sulla velocità se si hanno molti controller.

Il protocollo VI:
Ci focalizziamo su un sistema con due soli livelli di cache.
Soltanto un cache controller può leggere o scrivere blocchi in un'epoca.
Le transazioni supportate sono:
•Get:per richiedere un blocco in read-write mode dal LLC controller
•Put: per scrivere dati sui blocchi verso LLC controller
List of events:
• Own-Get: get richiesta da cache controller locale
• Other-Get:get richiesta da cache controller remoto
• Any-Get: get richiesta da un cache controller qualunque
• Own-Put: put richiesta dal cache controller locale

• Other-Put: put richiesta da cache controller remoto
• Any-Put: put richiesta da qualsiasi cache controller
• DataResp: il blocco di dati è stato correttamente ricevuto. E' l'evento che si ha quando avviene il trasferimento del dato.

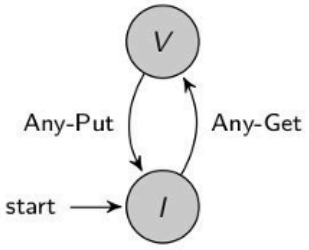
Abbiamo una FSM per ogni linea di cache. FSM implementato da cache controller e memory controller.

Cache controllers



V	Valid read-write copy
IV	Waiting for an up-to-date copy (transient state)
I	Invalid copy

LLC/Memory controller



V	One valid copy in L1
I	No valid copies in L1

FSM memory:
V= copia valida a qualche livello della gerarchia di cache
I= no copie valide a livello di gerarchia di cache, bisogna andare in memoria per ottenere il dato.
Questa FSM tiene traccia di ciò che avviene a livello delle cache.

FSM Cache controller:
Quando si effettua una get per ottenere un blocco si va in uno stato di transizione IV: la richiesta è stata inviata, ma il dato non è ancora stato ricevuto.
Other-get: qualche altro controller vuole fare una get sul dato, non può prendere il valore dalla memoria perché questo controller ha il valore aggiornato. Andiamo nello stato invalid perchè ora è il cache controller che ha fatto la richiesta a dover mandare il dato perchè possiede la copia aggiornata.

Nessuna architettura oggi implementa questo.
Se il cache controller vuole solo leggere un valore stiamo rimandando sempre lo stesso valore. Per risolvere questo ogni cores ha una sua copia del valore, così che si possano fare operazioni concorrenti sullo stesso dato.
Il problema di questa protocollo è che non sappiamo se qualcuno legge o scrive.

C’è una sola copia valida, quindi non ci sono differenze tra stato “sharing” o “exclusive”. Quest’unica copia è in read-write mode e soltanto chi sta leggendo può fornire la copia a chi la chiede.

Cosa dovremmo offrire a livello di un protocollo di CC?

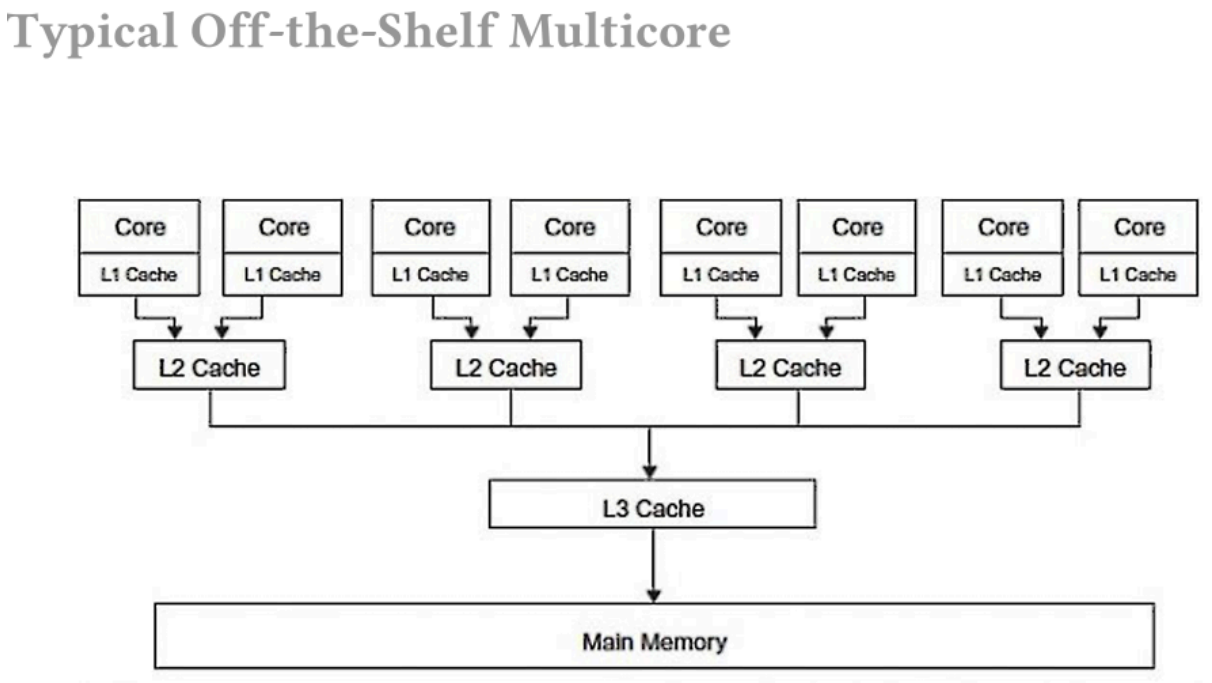
1. Validity→ un blocco è valido se ha il valore più aggiornato
2. Dirtiness→un blocco è sporco se qualcuno l’ha aggiornato
3. Exclusivity→solo una copia privata del blocco per una cache, le altre copie sono invalidate.
4. Ownership→avendo diverse copie di un dato bisogna distinguere chi è il proprietario.

Più proprietà garantite più stati e transizioni servono, dunque più spazio necessario.

MOESI:
Stati stabili di MOESI:

1. Modified→ il blocco è valido, esclusivo, di proprietà e potenzialmente sporco
2. Owned→ non è l’unica copia valida, ma il proprietario deve rispondere
3. Exclusive→ è l’unica copia presente, ma non è sporca, per poterla modificare si deve fare transizione nello stato Modified
4. Shared→ il contenuto è condiviso, le operazioni di lettura possono essere fatte da tutti gli altri cores
5. Invalid→ non si può leggere o scrivere il blocco.

TIpica architettura multi-core:
Il cache controller di livello più basso funge anche da memory controller.



In un’operazione di load si ha necessità di un indirizzo, esso è dato a livello di programma, quindi è un indirizzo logico, non fisico.

A che livello si deve fare la traduzione?
Dato che per fare la traduzione ci vuole del tempo farla a livelli L1 avrebbe poco senso e ridurrebbe notevolmente la velocità.
Sicuramente la cache L ha bisogno dell'indirizzo fisico perchè si comporta come "memory controller".
In realtà l'indirizzo fisico si usa anche in L2, c'è la cache TLB che lavora in parallelo rispetto alla cache L1. Se L1 ha un miss essa fa una richiesta a L2 usando l'indirizzo fisico calcolato dal TLB. Se si ha un'hit nel TLB non si paga alcun costo nel fare la richiesta a L2, si pagano solo i TLB miss.

Tempo di risposta: quanto tempo ci vuole per prelevare un dato dalla memoria?
statisticamente poco, dipende dagli hit a livello cache e TLB. possiamo sapere con certezza il tempo solo se conosciamo lo stato della gerarchia.

In-memory transactions:
Transazioni su DB implementate direttamente in HW.
Motivazioni:
• operazioni senza lock su strutture dati non saranno impedite se altri thread si bloccano durante l'esecuzione
• evitare problemi comuni con l'esclusione reciproca
• semplificare lo sviluppo del codice
• garantire buone prestazioni (a volte migliori rispetto all'utilizzo di blocchi).

Non usare i locks porta ad una notevole velocizzazione.
Per denotare le transazioni si usano delle istruzioni assembly addizionali.
Se si osserva un conflitto si fa lo squash.
Usiamo i protocolli CC per implementare le transazioni in HW.
Vengono effettuati dei cambiamenti a livello della cache L1.
Per disfare un aggiornamento si ricarica il blocco dalla cache L3.
Ogni cambiamento della memoria è committato atomicamente.
Le nuove istruzioni assembly introdotte sono:
• xbegin: inizio della transazione; tiene traccia dell'indirizzo della prossima istruzione e ritorna un codice
• xend: fine della transazione
• xabort:abort della transazione
• xtest: controlla se il codice sta eseguendo in una transazione

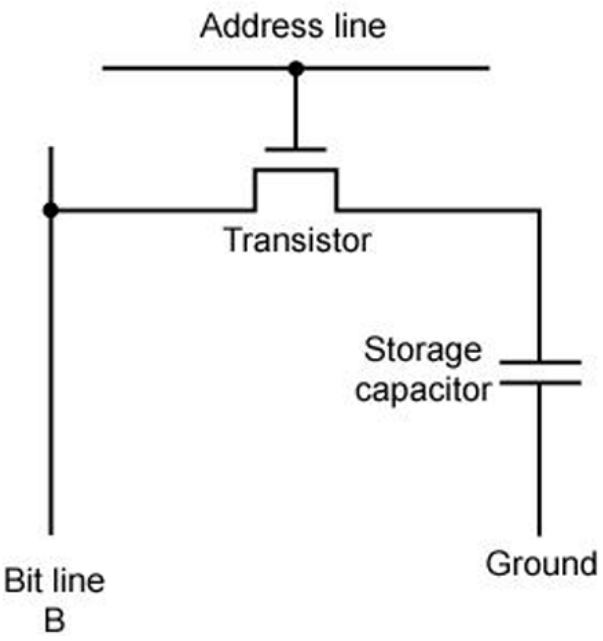
Le transazioni si possono annidare, c'è un contatore di annidamento, il commit può avvenire solo quando esso è pari a 0.
Il compilatore C supporta l'embedding di queste istruzioni nel codice.
Se qualche transazione va in abort lo stato del sistema torna indietro ed il SW viene notificato.
Si riprova la transazione finchè non va a buon fine.
Motivi per cui si può verificare un abort: se avviene un conflitto (ragioni SW), se si riceve un interrupt, un page fault, context switch, se la linea di cache ha dei limiti di taglia (ragioni HW).
Conflitto: qualcuno legge qualcosa che qualcun altro ha nel proprio writing set o qualcuno scrive qualcosa che qualcun altro ha nel proprio read o write set.

Nel caso dei limiti di taglia della cache size potremmo avere una serie di retries che non vanno a buon fine, perchè se la taglia della cache non basta per la nostra transazione non riusciremo mai ad andare in commit; per questo è sempre bene inserire anche un percorso alternativo che usa i locks anziché le transazioni.
Il codice ritornato dalla xbegin può darci idea del perché una transazione non è andata a buon fine, quindi lo possiamo usare nell'if per eseguire il percorso senza transazioni.

06/11/2023 (lez.16)
Ogni volta che si esegue una transazione quando si accede a una parte della memoria in lettura esso viene messo nel read set, se si accede in scrittura nel write set.
La CPU tiene traccia di chi sta leggendo/scrivendo una variabile.
L3 è condivisa tra le CPUs, quindi ha una visione condivisa dei dati. Il memory controller tiene traccia di chi sta accedendo alla memoria tramite le memory transactions.
Per fare questo bisogna aggiungere degli altri stati al modello MOESI.
Se la transazione va in abort c'è bisogno di rieseguirla di nuovo.
Nel caso degli abort dovuti a ragioni hardware il firmware ci dà un feedback riguardo l'abort.

Esempio slides 74-77:
Se una sola CPU esegue la transazione quando avrà modificato il registro a0 essa sarà l'unica che potrà dare alle altre questo valore qualora richiesto, in quanto sarà "exclusive owner". Il costo senza interferenze è minimo.
Se CPU0 inizia la transazione e poi CPU1 richiede l'accesso al registro a0, che sta nel write set di CPU0, CPU0 osserva un conflitto e abortisce la transazione.
Ad un certo punto CPU1 riuscirà a leggere il valore dalla memoria e CPU farà una retry.
Anche se CPU0 aveva quasi finito la transazione ha dovuto abortire, questo perché il firmware non ha idea di quanto durerà una transazione. In questo scenario le transazioni hanno priorità più bassa di tutti, quindi sarebbero scavalcate anche da operazioni che avvengono al di fuori di una transazione.

Recap on DRAM:
La RAM è composta da celle, ognuna può salvare un bit.
Cella di memoria che possiamo selezionare (read/write) tramite un indirizzo.
I bit di controllo dicono se si sta leggendo o scrivendo in una cella.
Per la write c'è un bus di dati bidirezionale usato dalla CPU per scrivere il valore nella cella, nel caso della read viene fatto il "sensing" del contenuto della cella e poi viene trasferito sul bus.
E' interessante vedere come sono fatte le celle internamente e come sono accoppiate tra loro per comporre la RAM ad alta densità che usiamo oggi.
Internamente la DRAM usa condensatori per mantenere i dati.
Un condensatore è molto semplice da costruire, ma disperde carica da solo, quindi quando leggiamo un valore dal condensatore esso piano piano si scaricherà, quindi periodicamente bisogna fare refresh: leggerlo e riscriverlo.
Quando si legge qualcosa chiudiamo il transistor, cortocircuito, quindi scarichiamo il condensatore. Quando leggiamo qualcosa stiamo distruggendo il valore in memoria, ma non vogliamo che accada ciò. Una read è un'operazione distruttiva.
Per memorizzare più di un bit si usa un array di celle.
Abbiamo una serie di transistors e capacitori.
Address line viene attivata dall'indirizzo per selezionare le celle che voglio accedere.



Operazioni sulla DRAM:
Con l'address line selezioniamo le celle da accedere, poi se vogliamo effettuare un'operazione di scrittura mettiamo voltaggio sulla bitline e cortocircuitiamo il transistor (segnalare la bitline significa questo).
Quando effettuiamo una read selezioniamo l'address line, prendiamo il valore del voltaggio del condensatore e lo confrontiamo con un valore soglia per vedere se è uno 0 o un 1.
La read è un'operazione distruttiva, perché leggendo scarico il condensatore, quindi periodicamente devo fare dei refresh, ovvero andare a riscrivere nella cella di memoria letta.
il problema è che i capacitori si scaricano velocemente, soprattutto all'inizio, quindi devo fare molte operazioni di refresh.
Esso viene fatto periodicamente ogni 64 ms, per questo motivo c'è un grande gap tra le performance della CPU e della memoria.
Se la memoria è grande fare il refresh ogni 64 ms richiede tempo.
La RAM è poco costosa, ma risulta molto inefficiente dal punto di vista energetico, perché si usa molta energia solo per fare il refresh, inoltre non si può accedere la memoria durante il refresh, quindi questo ha impatto anche dal punto di vista dei programmi che scriviamo.
Quindi le CPU eseguono a velocità molto elevate, mentre le DRAM hanno queste performances notevolmente inferiori, per questo gerarchia di cache, che non soffrono di questo problema.
Refresh distribuiti: anziché usare "burst refresh", ovvero bloccare le operazioni per fare il refresh suddividiamo i 64 ms nel tempo, in questo modo in momenti diversi facciamo il refresh di zone diverse della DRAM.
Quindi la probabilità di richiedere una parte di memoria non disponibile si riduce.
L'idea è che non sappiamo quando un refresh viene eseguito, quindi non sappiamo se accedere una cella di memoria porterà a un hit o meno.
Dato che all'aumentare della capacità di memoria il refresh introduce un overhead sempre più significativo, per questo aggiungiamo sempre più livelli di cache con livelli inferiori sempre più grandi, che significa che la CPU costa di più ma questo è l'unico modo per avere performances decenti nei nostri sistemi.

Hardware based attacks and countermeasures:
Un attaccante potrebbe voler: leggere dati in un'applicazione o scrivere dati in un'applicazione per cambiare il suo comportamento.
Quindi abbiamo due tipi di "primitive".
Reading primitives:
Ogni volta che si esegue un'operazione può o meno esserci una fuga di dati relativa al tempo.
Per esempio quando si effettua un confronto tra stringhe a seconda della lunghezza delle stringhe esso impiega più o meno tempo; quindi c'è una grande relazione tra il tempo speso nell'esecuzione di una funzione ed i dati processati.
Time invariant→ il tempo speso per eseguire un'operazione non ha nulla a che fare con il dato processato.
La gerarchia di cache è fortemente non-time invariant, infatti il tempo richiesto può variare dall'ordine dei ns a quello dei ms, nel caso in cui debba accedere in memoria.
Trucco: cercare di controllare lo stato della cache.
I livelli più bassi della gerarchia di cache sono condivisi, quindi le modifiche fatte da altri sono visibili a noi, possiamo misurarne il tempo ed estrapolarne delle informazioni.

Esempio:

Montgomery Ladder

Input: Point P, scalar n, k bits
Output: Point nP
$R_0 \leftarrow \mathcal{O}$
$R_1 \leftarrow P$
for *i from k to* 0 **do**
  if $n_i = 0$ **then**
    $R_1 \leftarrow R_0 + R_1$ — cache line A
    $R_0 \leftarrow 2R_0$ — cache line B
  else
    $R_0 \leftarrow R_0 + R_1$ — cache line C
    $R_1 \leftarrow 2R_1$ — cache line D
  end
end

Algoritmo di Montgomery, serve per calcolare ellyptic curves encryption. "n" è uno scalare, dovrebbe essere una nonce, se si conosce la nonce la decryption è semplice. Gli if-esle sono mappati su diverse linee di cache perchè sono più di 64 Bytes; se si invalida una linea di cache si avrà un miss, quindi più tempo di esecuzione. Quale blocco eseguito dipende dal valore della nonce.
L'attaccante induce una serie di conflitti per invalidare le linee di cache e misura il tempo di esecuzione del codice eseguendolo molteplici volte per ricostruire la nonce bit a bit.

Side channel attacks:
Un side channel è della memoria che mi permette di accederne altra o di trovare dei pattern di accesso.
Ci sono diversi modi per portare avanti un side channel attack:

• Prime + Probe
• Flush + Reload
• Flush + Flush
• Evict + Time
• Evict + Reload
• Prime + Abort

Usiamo la gerarchia di cache come side channel, l'idea è portare la gerarchia in uno stato noto ed osservare i side effects generati da altri processi.
Due fasi per portare a termine attacco:

1. Montare un side channel
2. Perpetrare l'attacco

Pre-attack phase:
Si vuole acquisire il target: l'area di memoria che si vuole rivelare.
Bisogna capire per l'applicazione in cosa consiste un hit ed un miss.
Per implementare un side channel attack si deve conoscere bene l'architettura.
La prima cosa da fare nella pre-attack phase è misurare la cache per sapere quanto tempo ci vuole per un hit o un miss.
Si porta il canale in un qualche stato noto, si attende che la vittima acceda il sistema, si osserva l'accesso della vittima e si ottiene qualche informazione da ciò.
Dobbiamo ricordare che nelle architetture di cache abbiamo sia indirizzi fisici che logici.
Bisogna trovare un modo per "rompere" il mapping.
Ad accedere alla cache non sono solo attaccante e vittima, quindi bisogna tener conto anche di questo, inoltre bisogna tener conto del fatto che si può essere deschedulati.

Evict+time:
Si utilizza il fatto che quando si vuole ricaricare una linea di cache dopo che è stata rimossa ci vuole del tempo. Si lascia che la vittima esegua, i dati di cui ha bisogno sono caricati nella cache, così l'attaccante può calcolare un tempo di "baseline", dopodichè l'attaccante elimina la linea di cache di interesse e misura il tempo che la vittima impiega per accedere la linea di cache, se è quella che l'attaccante ha eliminato impiegherà un tempo notevolmente superiore rispetto alla baseline.

13/11/2023 (lez.18)
→*Esempio codice evict_time.c*
La vittima e l'attaccante sono nello stesso programma.
L'obiettivo è scoprire qual è il segreto usato dalla vittima tra i segreti presenti qui.

Fase pre-attack→ ripetere l'esecuzione della vittima molteplici volte, scoprire il tempo necessario in caso di hit. Flush per acquisire il target in memoria, portiamo la xcache in uno stato noto. Scriviamo il dato in un buffer per avere l'accesso in memoria, dopo questa linea il dato è caricato in cache. Se non c'è interferenza di altri processi eseguendo di nuovo il target avremo un hit.

```
unsigned long pre_attack(void)
{
        unsigned long time1, time2;
        unsigned long baseline = 0;

        for(int i = 0; i < TRIALS; i++) {
                // Bring the cache into a known state
                flush_secrets();

                // Run target to load its working set
                run_target();

                // Working set is now loaded
                time1 = rdtsc();
                run_target();
                time2 = rdtsc();
                baseline += time2 - time1;
        }

        baseline /= TRIALS;
        return baseline;
}
```

Fase attacco→ rieseguiamo la vittima, prima cicliamo su tutti i segreti per flushare solo un segreto in particolare, portiamo la cache nello stato noto, così che se la vittima userà quel segreto ci sarà un miss.

```
int main(void)
{
        unsigned long time1, time2;
        unsigned long baseline, secret_time;
        printf("*** Evict+Time POC started***\n");

        baseline = pre_attack();
        printf("Baseline execution time is %lu\n", baseline);

        // Run again the target evicting a part of the set
        for(int i = 0; i < SECRETS; i++) {
                secret_time = 0;

                for(int j = 0; j < TRIALS; j++) {
                        clflush(secrets_table[i]);
                        time1 = rdtsc();
                        run_target();
                        time2 = rdtsc();

                        secret_time += (time2 - time1);
                }

                secret_time /= TRIALS;
                if(secret_time > baseline)
                        printf("Target is likely using secret %d\n", i);
        }


}
```

Maggiori difficoltà nell’eseguire questo attacco:

1. Implicitamente stiamo usando la cache L1 con indirizzi virtuali, per evincere l'indirizzo fisico dovremmo flushare e rimappare periodicamente e capire dalle collisioni qual è l'indirizzo fisico
2. Se il target non fosse deterministico no avrebbe senso misurare la baseline in questo modo
3. Assumiamo di poter controllare l'esecuzione del target, cosa non scontata. In realtà potrebbe volerci molto tempo per portare avanti questo attacco se non si ha il controllo della vittima.

Flush+reload:
Si basa sull'esistenza della memoria condivisa.
Il target è una libreria condivisa.
Se possiamo controllare la memoria in termini di flush di una linea, possiamo flushare la linea che la vittima sta usando e rieseguire la vittima.
Dopo di chè l'attaccante ricarica la linea, se ha un hit, quindi un reload veloce, significa che la vittima ha acceduto quella linea.
Cerchiamo di ottenere alcuni dati, non solo quale variabile ma anche il suo valore.

→*Esempio codice flush_reload.c*
Fase pre-attack→ il vettore dichiarato nell'attaccante è di 256 pagine, lo usiamo per trovare il valore della variabile. Per materializzare in memoria questo array lo dobbiamo inizializzare, quindi scriviamo almeno 1 byte per ogni pagina.
Dato che vogliamo ottenere un byte per volta dobbiamo sapere quanto tempo si impiega in caso di hit e di miss su ogni linea di cache.
Abbiamo due cicli for, il primo per misurare gli hit, infatti accediamo sempre la stessa variabile del ciclo esterno.
Nel secondo ciclo flushamo la linea di cache prima di accederla, quindi misuriamo il tempo di miss.
Alla fine si ottiene il tempo medio che si impiega per un hit e un miss.

Fase attacco→ vogliamo risalire a molteplici dati. Identifichiamo il target: la cache line che vogliamo attaccare. Flushiamo l'intero vettore così che non sia in cache, poi chiediamo alla vittima di accedere alla memoria che vogliamo.
Nella funzione "make_side_effect" accediamo un elemento diverso del vettore in base all'indirizzo che la vittima accede.
Accediamo una cache line che sia abbastanza distante da non essere stata precaricata dalla cache in qualche momento. Se si è in grado di accorgersi di un hit associato a qualche linea di cache si può risalire al valore della variabile acceduta dalla vittima.
Si mappano i valori in diverse locazioni di memoria. Si usa il valore per ricaricare una specifica linea di cache.

Maggiori difficoltà nell'eseguire questo attacco:
1. Assumiamo di poter far accede la memoria alla vittima
2. Se ci sono molte VMs in esecuzioni è verosimile avere lo stesso valore in diverse pagine, alcuni OS per risparmiare memoria fisica collassano più pagine fisiche nella stessa se sono accedute solo in read mode
3. Questo tipo di attacco richiede molto tempo

Nota (da ChatGPT):
L'attacco "Flush and Reload" è una tecnica di side-channel attack (attacco tramite canali secondari) che mira a ottenere informazioni sensibili da un sistema informatico sfruttando il modo in cui vengono gestiti i dati nella cache della CPU.

Ecco come funziona:
Flush: L'attaccante induce la vittima a eseguire un programma che accede a dati sensibili. Durante questo accesso, i dati vengono caricati nella cache della CPU.
Reload: L'attaccante monitora l'accesso alla cache della CPU per vedere se ci sono stati accessi ai dati sensibili. Ciò viene fatto guardando il tempo richiesto per accedere ai diversi blocchi di memoria nella cache.
Interpretazione: Utilizzando le informazioni raccolte durante il monitoraggio della cache, l'attaccante può dedurre quali dati sono stati acceduti e quindi inferire informazioni sensibili sul programma in esecuzione.

In sostanza, l'attaccante sfrutta la differenza di tempo richiesta per accedere ai dati nella cache della CPU per dedurre quali parti della memoria sono state utilizzate di recente, ottenendo così informazioni sulle attività in corso sul sistema.
L'attacco "Flush and Reload" è particolarmente efficace contro algoritmi crittografici o altri programmi che manipolano dati sensibili in modo prevedibile. È stato dimostrato che può essere utilizzato per recuperare chiavi crittografiche e altre informazioni riservate.

Prime+probe:
La prima versione funzionava solo su cache L1, poi è stato modificato per funzionare anche su cache L3, ciò significa che si possono ottenere dati di altre VMs.
Rileva quando la vittima sta accedendo ad uno specifico valore.
“Prime” significa che l’attaccante riempie la cache con qualche valore noto.
L’attaccante dopo che la vittima esegue riaccede la linea di cache, se ottiene un miss significa che la vittima ha acceduto quella linea di cache, c’è stato un conflitto quindi la linea è stata eliminata dalla cache.
Usiamo un “array di probe” per evincere il valore.
Nelle memorie set-associative però l’eviction di una linea di cache non avviene se non è stato riempito tutto il set, quindi si deve evincere l’intero set.
Bisogna generare da sé diverse eviction e generare un eviction set.
Per realizzare questo attacco in scenari reali ci vuole molto tempo.
Mapping dinamico memoria-cache: Nelle moderne architetture di cache, la mappatura della memoria principale (RAM) alla cache della CPU viene definita durante l'esecuzione del programma. Questo significa che la posizione dei dati nella cache può cambiare nel tempo in risposta ai diversi accessi alla memoria.
Poiché la mappatura della memoria alla cache può cambiare nel tempo, l'attaccante ha solo una finestra temporale limitata durante la quale può sfruttare una particolare configurazione della cache per condurre l'attacco Flush and Reload.
Per creare un insieme di dati da "espellere" dalla cache (eviction set) e indurre il ricaricamento di dati sensibili, l'attaccante procede come segue:

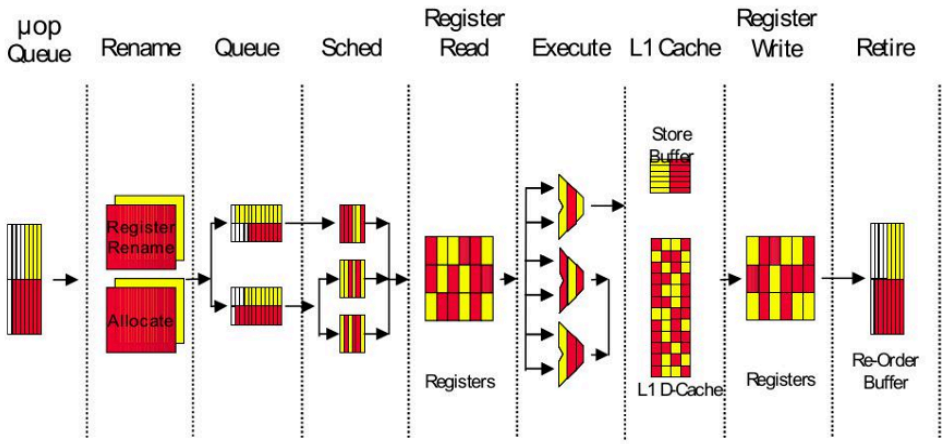
1. Scegli casualmente un insieme di N indirizzi di memoria.
   Carica questi indirizzi nella cache eseguendo accessi ad essi.

2. Durante questo processo, potrebbe verificarsi una "auto-collisione", ovvero un indirizzo di memoria precedentemente scelto potrebbe essere selezionato casualmente di nuovo.
   Gli indirizzi di auto-collisione devono essere rimossi dall'insieme, in quanto non sono utili per l'attacco.
3. Attacco classico: Una volta creato l'eviction set l'attaccante è pronto per l'attacco Flush and Reload, utilizzando la tecnica classica descritta in precedenza, che consiste nell'osservare il tempo richiesto per accedere ai dati nella cache e inferire così quali dati sono stati recentemente acceduti.

Prime+abort:
Si inizia una transazione, si porta in cache un dato, si aspetta che qualcuno acceda il dato e quando accadrà si verificherà l'abort della transazione.
Non c'è bisogno di una fase in cui si misura il tempo, perché le transazioni funzionano come callback hw.
Fix: disabilitare le transazioni hw.
Si può lavorare sia a livello di cache L1 che L3.
I read e write set sono a diversi livelli della cache: per spiare L1 si deve scrivere una variabile, per spiare L3 leggerla.

Flush+flush:
E' una variante di flush+reload.
Se si ha una linea in uno stato valido, se si fa il flush c'è bisogno di più tempo perché il memory controller deve decidere chi è il proprietario, quindi il flush può impiegare più tempo.
Se una linea viene ricaricata significa che la vittima l'ha acceduta e si può risalire al valore.
E' difficile identificare la malignità di un flush. Se si fa il flush di una linea già flushata non si crea alcun "rumore" sulla cache.

Meeting the out of order pipeline:
Nelle O.O.O pipeline se si fanno delle operazioni illecite restano dei side-effects.
Supponiamo di fare un accesso non valido, lo stage in cui il firmware se ne accorge è il "retirement stage", quindi un'istruzione non valida (phantom instruction) attraversa ogni altro stage della pipeline.
Per caricare dati dalla memoria si interagisce con il controller della cache L1, quindi questa istruzione manderà la richiesta alla cache L1 di caricare quella parte di memoria.
Quindi anche con delle phantom instructions possiamo lasciare dei side effects nel sistema: gli attacchi che sfruttano questi meccanismi sono i "transient execution attacks" e si basano su un bug di design.
I designers della CPU demandano la risoluzione ai designer della gerarchia di cache e viceversa, questo ha fatto sì che questo bug sia rimasto nascosto per 20 anni.



Meltdown:
Usiamo l'array di probe (abbiamo 256 pagine), poi si fa il flush come abbiamo fatto prima.
Quando facciamo un accesso non valido l'istruzione attraversa tutta la pipeline, dato che è O.O.O., quindi possiamo lasciare dei side-effects.
Quando si accede in maniera non valida la memoria otteniamo un segmentation fault e viene generato un interrupt.
Questo tipo di interrupt può essere intercettato dal programmatore nei sistemi operativi Linux, quindi possiamo far sì che l'applicazione continui ad eseguire.
Il firmware della CPU vuole che sia la gerarchia di cache a prendersi cura di questo accesso non valido e la gerarchia di cache vuole che sia la CPU.
L'istruzione solleva qualche segmentation fault, l'attaccante la intercetta e chiede al SO di far continuare l'esecuzione dell'applicazione, l'attaccante legge il valore anche se non avrebbe potuto leggerlo e lo moltiplica per 4K, lo usa per accedere al probe array e l'attacco è fatto, perché a seconda della linea che è stata acceduta si otterrà una hit (lo si capisce misurando i tempi di accesso).

Meltdown Primer

possible byte values

```
uint8_t *probe_array = new uint8_t[256 * 4096];
// Make sure probe_array is not cached
uint8_t kernel_memory = *(uint8_t*)(kernel_address);
uint64_t final_kernel_memory = kernel_memory * 4096;
uint8_t dummy = probe_array[final_kernel_memory];
// handle (eventual) SEGFAULT
// determine which of 256 slots in probe_array is cached
```

Intel OOO Execution

Avoid prefetching & different lines

15/11/203 (lez.20)
Attacco alla BPU:
La BPU impara i "behavioural paths" di un'applicazione per inserire nella pipeline l'istruzione corretta.

un’applicazione può “avvelenare” la BPU nel senso che può eseguire ripetutamente lo stesso percorso così che lo apprenda.
La BPU non cambierà immediatamente comportamento, quindi se l’applicazione cambia istantaneamente comportamento la BPU farà una predizione sbagliata, continuando a predire lo stesso outcome imparato.
Grazie alla OOO pipeline possiamo far processare alla gerarchia di cache informazioni sbagliate.
Ad un certo punto l'errore verrà rilevato, ma resteranno dei side effects micro-architetturali.

Spectre v1:

Si esegue più volte lo stesso codice facendo sì che l’”actual value” con cui confrontare la condizione non sia disponibile in cache, in questo modo si dà luogo ad un’esecuzione speculativa; il check viene ritardato.
Quando si accede un elemento di un array quell’array viene reso disponibile in cache; insegnamo alla BPU ad eseguire speculativamente questo pezzo di codice; fino ad un certo punto x risulterà minore di array_size, ma all’ultima iterazione non sarà più così.
Nell’attacco si può saltare a qualsiasi punto della memoria, come effetto collaterale della speculazione.
Se si accede un valore fuori dal limite si avrà un segmentation fault e lo si potrà intercettare.

Spectre v2:
Anziché utilizzare il proprio codice qui l’attaccante sceglie del codice dall’AS della vittima (e.g. usando shared libraries).
Gadget: pattern di bytes che rappresenta un’istruzione macchina e di cui si può cambiare il comportamento.
Utilizzando il costrutto dello switch si possono raggiungere diverse porzioni di codice.
Si sta utilizzando il codice dell’applicazione per fare qualcosa di diverso rispetto a ciò per cui è stato pensato.
Si attiva una porzione di codice per eseguire in modo diverso rispetto a quello per cui era stata progettata.
Si avvelena il branch target buffer affinché salti ad una locazione dove c’è un gadget. Non c’è una vulnerabilità dell'applicazione, si sfrutta una vulnerabilità presente a livello hw.
Il gadget solitamente è costituito da 1 o 2 istruzioni macchina, quindi non si deve iniettare molto codice per ottenere il comportamento desiderato, ciò rende l’attacco realizzabile.
Solitamente il gadget si prende da librerie condivise.

Esempio:

Si somma un registro con una variabile in memoria.
il probe array è una qualche regione di memoria in una libreria condivisa.
Tramite l'addizione si può raggiungere qualsiasi locazione di memoria.
Facendo ebx = m - 0x13BE13BD − edx si fa ricalcolare alla CPU la locazione di memoria che si vuole accedere.
Se questo codice viene eseguito speculativamente avremo in pipeline un accesso ad una locazione di memoria illecita; poi ci sarà lo squash della pipeline, ma si potranno sfruttare comunque i side effects nella gerarchia di cache.

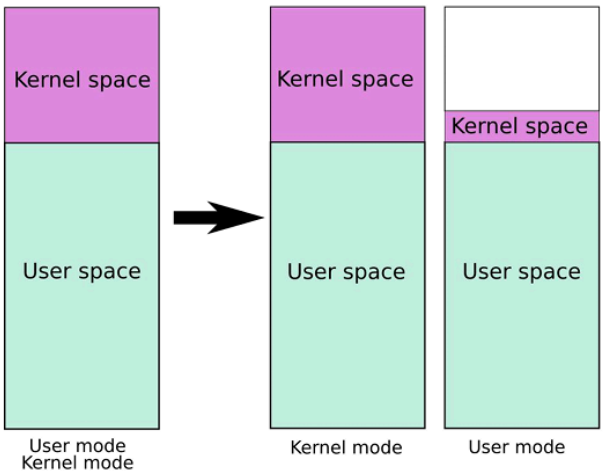
Mitigate side channel attacks:
Come mitigare i side-channel attacks?
Si potrebbe pensare di rendere la gerarchia di memoria time invariant, ma si perderebbe ogni vantaggio della gerarchia avendo il tempo di miss pari a quello di hit.
Quindi non c'è un modo semplice di risolvere il problema a livello di gerarchia di cache.
Possiamo attuare qualcosa di più complesso:

1. Restringere l'accesso agli high resolution timers (come rdtsc). Quindi a livello utente non si può misurare il tempo di nulla; in questo modo a livello utente non si può notare la differenza che intercorre tra un hit ed un miss.
   Alcuni vendors hanno proposto di introdurre questa restrizione, ma il benchmark è necessario nell'ingegneria informatica, perchè bisogna capire quanto bene funzioni qualcosa, quindi questa capability non può essere rimossa
2. Marcare alcune regioni della memoria uncachable, questa è una challenge a livello hw, bisogna mantenere dei metadati, quindi ci sono problemi nelle performance, perché ciò che vogliamo rendere uncachable sono librerie condivise in particolare, quindi accedute molto di frequente; c'è bisogno di un firmware che controlli se questa memoria viene messa in cache
3. Dopo che AES è stata rotta a livello sw Intel ha implementato AES a livello hw, nel chip; ma qui risolviamo un problema specifico, non vale per tutti gli altri algoritmi crittografici
4. Distribuire i segreti, suddividere i dati sensibili tra più linee di cache, così da aumentare il tempo di ricostruzione, ma questo è un esempio di security by obscurity, che non funziona mai
5. Disabilitare l'hardware transactional memory (tsx), di default lo è, ma il problema è fondamentale è nella O.O.O pipeline, quindi disabilitare la OOO pipeline, ma sarebbe critico dal punto di vista delle performances; quindi si può fare questo soltanto nei sistemi che mantengono segreti particolarmente rilevanti, quindi particolarmente critici.

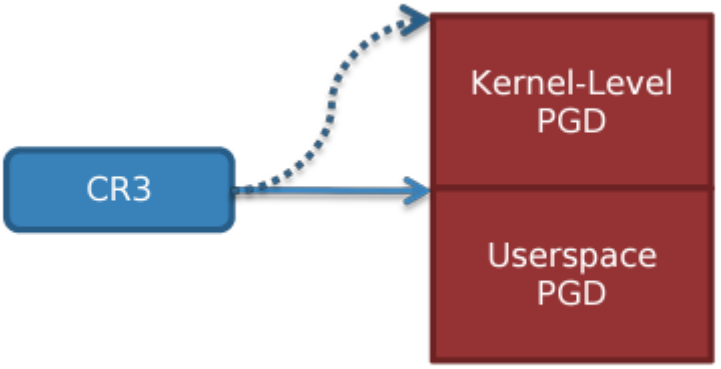
Ora ci focalizziamo su particolari livelli della gerarchia di cache.
Mitigazioni agli attacchi L1:
La cache L1 è per-core, quindi l'unico modo in cui può essere usata per effetti collaterali è nel caso del "multithreading simultaneo".
Un CPU core è condiviso tra più threads.
Quindi si dovrebbe disabilitare il multithreading, ma questo sarebbe critico dal punto di vista delle performance.
Un'altra mitigazione sarebbe mettere in atto un sistema di rilevamento, promettente ma difficile da mettere in atto.

Gli attacchi sono tutti costituiti da due fasi, la preparazione (pre-attack) e l'attacco vero e proprio. Nella fase di pre-attack si seguono dei pattern, si porta la cache in un qualche stato noto, si flusha il probe array (grande mole di dati); si crea del rumore nella gerarchia di cache.
Le moderne CPU hanno "hardware performance counters", sono registri che si possono programmare per contare alcuni elementi specifici dell'architettura microkernel, in questo modo si possono estrarre dei profili comportamentali delle applicazioni.
Se si matchano dei pattern di attacco si può sospettare che un'applicazione sia malevola.
Una volta identificata un'applicazione con queste caratteristiche la si esegue da sola su un CPU core; se il sistema è particolarmente critico si può anche uccidere.
Dato che nei sistemi si hanno computazioni condivise ci vuole molto tempo per eseguirle, quindi non si può uccidere l'applicazione, la si isola.
Il problema è che questi contatori cambiano a seconda del vendor.

Mitigazioni per Meltdown:
Le applicazioni regolari possono trafugare dati dal SO.
Nell'AS virtuale di un'applicazione si ha il kernel space e lo user space.



Meccanismo di protezione segmentation fault quando si accede il kernel space, ma c'è il problema dei side channels, come abbiamo visto.
Come evitare i side channels?
Dato che noi accediamo la memoria logica, quando si accede quella fisica vengono eseguite delle traduzioni da parte del SO.
Il SO può cancellare parte della struttura, così da impedire che l'applicazione veda la memoria fisica per la maggior parte della memoria.
Ogni volta che si passa da user a kernel mode il SO viene eseguito per ricreare il mapping originario, sono 4GB di mapping, quindi può essere costoso rimappare tutto, per questo motivo si utilizza un'altra modalità: si legge soltanto il valore di un registro: CR3.
CR3 è radice di due alberi, una è la PGD user mode e l'altra la PGD kernel mode.
Pezzi di memoria diversi sono allocati a diverse applicazioni, quindi si hanno diverse page tables a livello user; quindi un nodo radice per ogni applicazione.
Per spostarsi da user a kernel mode basta il flip di un bit, dato che le 4Gb sono una potenza di 2.



CR3 is updated when transitioning to and from kernel mode

Mitigazioni per Spectre:
E' più complicata.
La soluzione è la retpoline, sta per "return trampoline".
Si utilizza per far eseguire alla CPU dei percorsi di codice noti anche in caso di cattiva speculazione. Ci focalizziamo sui salti indiretti, li vogliamo rimpiazzare con chiamate a dei "thunk", ovvero delle subroutine che servono a rimandare la computazione finché il risultato non serve davvero.
Il compilatore genera il thunk.

```
jmp *%r11     jmp retpoline_r11_trampoline;
call *%r11    call retpoline_r11_trampoline;

              retpoline_r11_trampoline:
                call set_up_target
              capture_spec:
                pause
                jmp capture_spec
              set_up_target:
                mov %r11, (%rsp)
                ret
```

Target known at compile time: will not trigger speculative target resolution

Speculation is captured here, which is a loop which will be eventually flushed

Modify the target of the return instruction on stack, leaving the Return Stack in the CPU unaffected

Il salto incondizionato viene sostituito con un salto ad un target noto a compile time, quindi non c'è speculazione.
saltiamo a "retpoline_r11_trampoline", che chiama "set_up_target", ora qui:

a. Se avviene la predizione corretta effettivamente si salta a %r11, in quanto in "set_up_target" modifichiamo lo stack pointer, ponendolo uguale all'indirizzo finale che vogliamo raggiungere (%r11), così quando si fa la return si torna a questo indirizzo
b. Se avviene una speculazione sbagliata ed il salto non viene effettuato si va in "capture_spec", che rappresenta un loop infinito, che verrà eventualmente poi flushato; in questo modo non abbiamo alcun side effect in cache.

Abbiamo bisogno di un thunk per ciascun registro di CPU.

Un'altra soluzione possibile per prevenire l'avvelenamento dei salti è utilizzare delle istruzioni ISA aggiuntive per controllare come avviene una cattiva speculazione.
Questa soluzione è significativa dal punto di vista delle performance.

Indirect Branch Restricted Speculation (IBRS):

- Il processore entra in una modalità IBRS speciale.
- La BPU non è influenzata dalle previsioni al di fuori di IBRS.

Single Thread Indirect Branch Prediction (STIBP):

- Limita la condivisione della previsione dei branch tra il software in esecuzione su iperthread dello stesso core; i dati vengono disposti su più copie della BPU, una per ogni thread

Indirect Branch Predictor Barrier (IBPB):

- Impedisce al software in esecuzione prima di impostare la barriera di influenzare la previsione dei branch da parte del software in esecuzione dopo la barriera.
- Questo viene fatto svuotando lo stato del Branch Target Buffer, viene flushata la BPU ogni volta che si invoca la barriera

Setting up a writing primitive:
Dato che i condensatori sono molto vicini tra loro anche le linee per leggere e scrivere il contenuto di una cella sono molto vicine; ogni volta che leggiamo o scriviamo la memoria i flussi di corrente sono molto vicini e generano dei campi elettromagnetici, che andandosi a sommare vanno a caricare o scaricare i condensatori in maniera più rapida rispetto all'andamento naturale, quindi si può arbitrariamente cambiare il contenuto di una cella di memoria, perché se ad esempio si scarica un condensatore e lo si porta sotto una certa soglia da 1 diventa 0.
Questo non avviene randomicamente e ciò avviene perché il chip è realizzato in questo modo.
Un'altra cosa che si può fare è sfruttare la duplicazione: se un attaccante sa che qualcosa è vulnerabile e vuole flippare un bit di una variabile/segreto di un'applicazione, può aspettare che una pagina sia duplicata, leggere ripetutamente per flippare i bit dell'altra applicazione e nessuno se ne accorge. Si usa la memoria condivisa.

RowHammer POC

```
code1a:
  mov (X), %eax  // read from address X
  mov (Y), %ebx  // read from address Y
  clflush (X)    // flush cache for address X
  clflush (Y)    // flush cache for address Y
  mfence
  jmp code1a
```

Si accede X, poi si accede Y, si creano campi elettromagnetici su due linee, se sono vicine si generano dei campi elettromagnetici; poi si flushano e si riaccedono, quindi stiamo "martellando" queste locazioni di memoria finché non siamo sicuri di aver flippato un bit.

Possibili mitigazioni:

1. Codici a correzione di errore: le memorie più recenti lo implementano, se un bit flippa il controller se ne accorge. Si può, però flippare anche il parity check.
2. Ridurre l'intervallo di refresh, ma gi' ci si spende troppo tempo, quindi sarebbe un problema dal punto di vista delle performances
3. Detection dei pattern, se si accede e flusha sempre la memoria il pattern [facilmente identificabile
4. pTRR: traccia l'attivazione delle linee e fa il refresh in caso di molteplici attivazioni, perché potrebbe essere un attacco. È un approccio di "security by obscurity", quindi fatto internamente dal memory controller.

Memory performance attacks:
Implementare denial of service memory al livello della RAM.
Ogni volta che si legge qualche riga della memoria si scarica qualche condensatore, quindi bisogna trovare un modo per non perdere i dati, per questo c'è un raw buffer, che è essenzialmente un registro, quindi non fatto con i condensatori.
Il raw buffer inizialmente è vuoto, poi quando viene richiesto l'accesso ad una colonna di una certa riga la riga viene portata nel raw buffer, tutti gli accessi a colonne diverse della stessa riga risulteranno in un hit. Ad un certo punto si chiede l'accesso ad una nuova riga, si ha un miss nel raw buffer, quindi si esegue il "write back" in memoria.
La memoria serve cache multiple.
Abbiamo due diversi attori che ad un certo punto iniziano a sperimentare conflitti.
C'è una latenza ridotta ogni volta che si ha un hit nel raw buffer, ma si vuole anche ottimizzare il modo in cui si accedono i vari banchi di memoria, perché non si vuole mettere molto sforzo nell'arbitraggio per accedere ai diversi banchi.
Dato che le richieste possono essere ritardate introduciamo una coda associata a richieste diverse che vengono da thread diversi.
Se si mandano continuamente richieste da uno stesso thread si continua a servire lui e gli altri threads non riescono ad accedere la memoria.
Non ci sono fix per questo problema al momento.

20/22/2023 (lez.21)
OS security principles:
Saliamo di un livello, dall'hw al so.

Principi fondamentali di sicurezza:
1. I sistemi devono essere utilizzabili sono dagli utenti legittimi
2. L'accesso deve essere basato su autorizzazione, che viene data dall'amministratore di sistema

Una tipologia di attacco è DoS, che fa sì che il sistema risulti inutilizzabile per gli utenti legittimi.
Bisogna configurare il sistema in modo sicuro.

Identificare gli utenti:
User authentication:
L'utente fa il login tramite una password, il sistema operativo mantiene le password in due pseudofile: /etc/passwd (accessibile dall'utente) e /etc/shadow (accessibile solo da root).

Com'è fatto il file /etc/passwd?
Ogni riga ha il seguente formato: username:passwd:UID:GID:full_name:directory:shell
username→associazione tra utente e username
passwd→ il SO la mantiene al sicuro
UID→ user ID usato dal SO dopo il login
GID→ group ID, i privilegi di un utente dipendono dal suo GID.
full_name→ stringa
directory→ home directory

Se in questo file si potesse effettivamente accedere la password di ogni utente avremmo un problema nella sicurezza, anche se si usasse il "salt"; per questa ragione le password sono mantenute nell'altro pseudofile: /etc/shadow.
In /etc/passwd al posto della password c'è un placeholder "x".

Com'è fatto /ets/shadow?
Ogni riga ha il seguente formato: username:passwd:ult:can:must:note:exp:disab:reserved
passwd→è la password cifrata
ult→ rappresenta i giorni passati da quando la pw è stata cambiata l'ultima volta; necessario per implementare politiche di sicurezza
can→ intervallo di tempo in termini di giorni, dopo cui è possibile cambiare nuovamente la password
must→ numero di giorno dopo cui la password va cambiata di nuovo
note→ numero di giorni dopo cui l'utente è sollecitato a cambiare la pw
exp→ numero di giorni dopo cui l'account viene disabilitato dopo la scadenza della password
disab→ numero di giorni a partire dal 1/1/1970 dopo cui l'account sarà disabilitato
reserved→ non viene usato, è un campo riservato

Come funzionano gli IDs in Unix?
L'utente che esegue un programma è identificato tramite UID. Quando si lancia un processo vengono mantenuti 3 UID:
-real UID: UID dell'utente che lancia l'applicazione
-effective UID: dice che cosa si può fare, ad un certo punto si può diventare un altro utente. In generale è "0", ovvero si possono eseguire delle capabilities di root. Ad un certo punto si chiede al SO di rimpiazzare il real UID con l'effective UID, seppure si lancia l'applicazione con privilegi amministrativi non si possono eseguire azioni come root finché l'applicazione non lo richiede internamente
-saved ID: è il vecchio ID, che viene salvato quando si cambiano privilegi, serve per poter tornare indietro.

Come funziona lo switch?
Si basa su due systemcall: setuid() e seteuid().
setuid() e non reversibile, si possono cambiare tutti e tre gli UID.
seteuid() è reversibile, e sta per "effective".
Si può cambiare lo UID anche ad un intero che non corrisponde ad alcun utente, dal punto di vista del SO un utente è solo un intero.
Si possono usare queste systemcall solo se si esegue come root.
Per essere root bisogna lasciare l'applicazione con "sudo" e "su".

C’è un bit nei metadati associati ad un file che dice se si può o meno cambiare il “real UID”.
Le capabilities associate ad un file sono “rwsm”, dove c’è uno sticky bit che indica che l’applicazione può essere lanciata solo dal proprietario del SO.
Il SO interagisce con il filesystem e lancia un’applicazione per conto di un utente; ogni utente può lanciare l’applicazione sudo con privilegi di root, il processo viene forkato così da poter essere lanciato con privilegi da amministratore.
→*Esempio montaggio di un modulo:*
Per Unix tutti è un file, quindi se rimpiazzi le systemcall intercettandole con un modulo si può fare tutto per cambiare i privilegi utente: questo è possibile solo se si monta il modulo come super user.

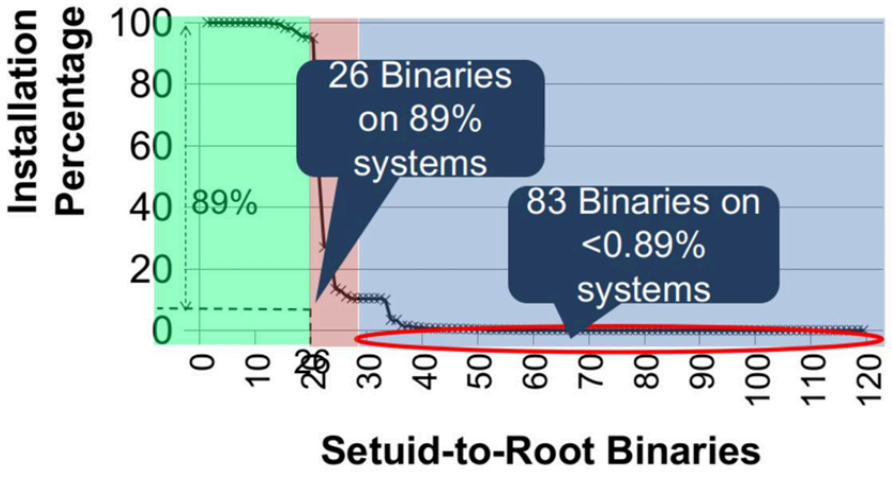
Quindi tutta la sicurezza è associata ai privilegi di super user, è questo il punto in cui un sistema sicuro si trasforma in un sistema non sicuro.
Se si è amministratori del sistema si può fare tutto.

Recap principle of least privileges:

- In a particular abstraction layer of a computing environment, entities must be able to access only the information and resources that are necessary for its legitimate purpose
  - Applies to processes, users, programs, virtual instances, ...

- *Better system stability*: it is easier to test possible actions of the applications/users and interactions with other applications.
- *Better system security*: vulnerabilities in one application cannot be used to exploit the rest of the machine.
- *Ease of deployment*: the fewer privileges an application requires the easier it is to deploy within a larger environment.

L’account root rompe il principio dei privilegi minimi.
nel file “/etc/sudoers” si può specificare cosa un utente può fare nel sistema, si possono dare privilegi a grana fine.
Un “God administrator” può causare problemi perchè se viene compromesso tutta la macchina viene compromessa.
Anche lo sticky flag può essere problematico: se lo si concede a molte applicazioni un’applicazione con un bug può lanciare un shell e tutti possono fare tutto eseguendo come root.
I binary con sticky bit sono vettori per attacchi.



Bisognerebbe eseguire come amministratore solo quando necessario, questo ci porta al concetto dei privilegi a grana fine, che controllano l'abilità di un soggetto di eseguire un'azione su un oggetto.

Component	Description
Subject	The entity making the request (a *process* or *user*)
Action	The requested operation (read, write, execute)
Object	The target of the action (a file, network connection, hardware)

La proposta di controllo degli accessi limita i danni, infatti un sistema sicuro non esiste, bisogna sempre tenere in conto che può essere attaccato, quindi bisogna limitare i danni di un attacco.
Inoltre i danni potrebbe essere anche causati da un utente che utilizza male il sistema.

Politiche di sicurezza:
Principali classi di politiche di sicurezza:

1. DAC→ la definizione di attributi di sicurezza avviene da parte degli utenti regolari del sistema
2. MAC→ c'è una distinzione tra utenti e "policy makers". Le politiche sono sotto il controllo degli amministratori di sistema, diverso dagli utenti del sistema.
   Chi usa il sistema è diverso da chi lo amministra.

In DAC gli utenti con privilegi possono cambiare le politiche, mentre con MAC no; si possono anche rafforzare le "time-based policies".
le politiche MAC sono particolarmente importanti in sistemi critici dal punto di vista della sicurezza. Ci sono prodotti commerciali (e.g. MACAfee) per gestire le politiche da remoto, utile in ambienti grandi.

Quando si configura una politica di sicurezza bisogna tener conto della distinzione tra:

1. External security→ nessun utente esterno al sistema deve poter accedere

2. Internal security→ nessuna persona deve poter attaccare il sistema dall'interno, neppure gli utenti temporanei (e.g. un utente attaccato tramite cavo ethernet alla rete di un azienda). Non va MAI trascurata.

Politiche a grana fine:
Possono essere DAC o MAC.
Si riferisce a qual è la granularità del permesso dato ad un soggetto su un oggetto?
A seconda della granularità si possono avere politiche più semplici o più difficili da gestire.
Essa si può basare su:

a. grandezza del sistema→ se si è root si può fare tutto ciò che si vuole, è la più semplice di tutte
b. directory→ se si vuole proteggere file o dispositivi
c. file
d. azioni su un file→ il classico rwx, per discriminare le azioni che qualcuno può fare su un file
e. porzioni di file→si può voler proteggere anche solo parte di un file

Il costo di non configurare accuratamente le politiche di sicurezza è comunque minore di quello che si pagherà se si verrà attaccati.

Posix ACLs:
Vediamo alcune implementazioni di politiche a grana fine su Posix (Unix e Mac).
Essenzialmente schema discretionary (DAC).
Le tre azioni classiche sono read, write ed execute: per ogni file abbiamo il proprietario ed una serie di attributi aggiuntivi per specificare chi può fare cosa su quel file.
Il proprietario del file può cambiare i permessi.

```
# file: audio
# owner: gwurster
# group: audio
user::rwx
group::r-x
group:powerdev:r-x
mask::r-x
other::---
```

Come funziona l'algoritmo di garanzia di permessi con ACL?

- if the user ID of the process is the owner, the owner entry determines access
- else if the user ID of the process matches the qualifier in one of the named user entries, this entry determines access
- else if one of the group IDs of the process matches the owning group and the owning group entry contains the requested permissions, this entry determines access
- else if one of the group IDs of the process matches the qualifier of one of the named group entries and this entry contains the requested permissions, this entry determines access
- else if one of the group IDs of the process matches the owning group or any of the named group entries, but neither the owning group entry nor any of the matching

named group entries contains the requested permissions, this determines that access is denied
- else the other entry determines access

Il group ID si può usare per bloccare l'accesso ad una risorsa.

Dopo il controllo su chi sei si fa il controllo sulle azioni che si possono fare:
- if the matching entry resulting from this selection is the owner or other entry and it contains the requested permissions, access is granted
- else if the matching entry is a named user, owning group, or named group entry and this entry contains the requested permissions and the mask entry also contains the requested permissions (or there is no mask entry), access is granted
- else access is denied

mask→ serve per mascherare l'accesso a chi non deve avere capabilities, è un modo veloce per configurare gli accessi. Per esempio se nessuno deve poter eseguire un file si "maschera" la capability di esecuzione.

Le ACL sono una serie di passi che il SO compie per determinare chi si è e dare o meno accesso in base all'oggetto che si vuole accedere e alle azioni che si vogliono eseguire su di esso.

Questo ha delle limitazioni: se l'owner è root ci si ferma al primo controllo e si può fare qualunque cosa si vuole.
Abbiamo visto che avere un god user rompe il sistema.

Capabilities:
Sono state progettate per evitare il problema del god administrator, dobbiamo definire cosa un amministratore di sistema può fare.
Lavorano a grana ancora più fine, perché implementano privilegi a grana fine anche per il root account.
In Unix ci sono due categorie di processi:
a. Unprivileged process→ si possono gestire con le ACL
b. Privileged process→ possono bypassare qualsiasi check dei permessi, che è esattamente il problema che abbiamo osservato prima

Le capabilities nascono per buttare tutto ciò che ha a che fare con privilegi di root.
E' un modo per ridurre i danni che possono essere fatti dagli utenti legittimi.
In generale se si lavora in applicazioni o a livello kernel ci sono diversi tipi di capabilities.
1. CAP_CHOWN→ "change owner" per cambiare l'owner di un file, se il tuo unico obiettivo è cambiare l'owner devi accedere solo quel privilegio, per il principio del privilegio minimo.
2. CAP_KILL→ per mandare un segnale di kill a qualche processo. Ispirato dal KISS principle, se devi mandare un segnale devi fare soltanto questo.
3. CAP_NET_RAW→ per creare un raw socket, perchè crearlo può essere critico, significa poter fare qualcosa sulla rete
4. CAP_NET_BIND_SERVICE→ per associare porte (<1024) a un socket
5. CAP_SYS_NICE→ per cambiare la priorità di un processo, in questo modo si potrebbe realizzare un attacco DoS

6. CAP_SYS_TIME→ cambiare il clock potrebbe causare problemi con i certificati
7. CAP_SYS_ADMIN→ introdotto per obiettivi di sviluppo e debugging. Il problema è che questa capabilities può essere usata come un nuovo “root”, infatti è una capability che permette di fare troppe cose scorrelate fra loro insieme.

Il kernel controlla se un thread ha una certa capability nel suo “effective set”.
Per assegnare e ritirare capabilities abbiamo due systemcalls: capget() e capset().
Per invocarle bisogna essere in un qualche livello di esecuzione privilegiata.
Sono systemcalls a basso livello.
Sono necessarie modifiche al filesystem, perché esso deve permettere di associare capabilities ad un file in esecuzione.

Ogni thread ha qualche set di capabilities associato:

1. Permitted set→ sono le capabilities che un thread può acquisire. Se si cancella una capability da questo set non la si potrà più riacquisire
2. Effective set→ se si vuole utilizzare una certa capability tra quelle permesse deve essere qui dentro, il kernel controlla qui i permessi di un thread
3. Inheritable set→ sono le capabilities date da un thread parent ad un qualche processo spawnato
4. Ambient set→ essenziali per l’esecuzione, esse sopravvivono dopo un execv, possono essere settate nuovamente solo se il thread è privilegiato.

La fork() non ha effetti sul set delle capabilities, lo spawning si.

Recap sulla differenza tra spawn e fork: [ChatGPT]
La differenza principale tra fork e spawn thread riguarda il modo in cui vengono creati e gestiti i processi e i thread in un sistema operativo.
La **fork** è un sistema di creazione di processi, tipico dei sistemi operativi Unix e Unix-like.
Quando viene chiamato fork, il processo genitore crea una copia di se stesso chiamata processo figlio.
Il processo figlio ha un proprio spazio di indirizzamento della memoria, una copia del contesto del genitore e un nuovo ID processo (PID).
Dopo la chiamata a fork, i due processi (genitore e figlio) continuano l'esecuzione indipendentemente l'uno dall'altro.
I processi figlio e genitore condividono file aperti, variabili globali e altre risorse del sistema operativo, ma ognuno ha una copia separata della memoria.
fork è utile per la creazione di processi concorrenti, ma può richiedere un carico significativo sul sistema a causa della duplicazione di risorse.
Lo **spawn** di un thread è un meccanismo per la creazione di thread all'interno di un processo.
I thread condividono lo stesso spazio di indirizzamento della memoria, le variabili globali e altre risorse del processo padre.
La creazione di un thread è generalmente più leggera rispetto a fork poiché i thread condividono la maggior parte delle risorse del processo padre e richiedono meno overhead.
I thread possono essere utilizzati per eseguire operazioni concorrenti all'interno dello stesso processo, consentendo una maggiore parallelismo.
La sincronizzazione tra thread deve essere gestita attentamente per evitare problemi come le race condition.

In sintesi, fork è utilizzato per creare processi separati con spazi di indirizzamento della memoria separati, mentre spawn è utilizzato per creare thread all'interno dello stesso processo condividendo lo stesso spazio di indirizzamento della memoria.

File capabilities:
Le capabilities sono mantenute nei file, negli “extended attributes”, che sono metadati attaccati al file.
Sul filesystem ci sono due categorie di capabilities associate ad un file binario.
Sono in gioco due attori: le capabilities di chi lancia un file e quelle associate al file lanciato.
Abbiamo due set essenziali di capabilities per i file:

1. Permitted set
2. Inheritable set

C’è anche uno sticky flag “effective” che dice se una capability è anche nell’effective set o solo nel permitted set.

27/11/2023 (lez.24)
Capability bounding set:
cap-bset→ è un meccanismo di sicurezza per-thread
Si utilizza per:

a. ridurre le capabilities di un thread dopo una execve. Il capability bounding set durante una execve viene messo in AND con il permitted set di un file ed il risultato di questa operazione è assegnato al permitted set del thread
b. Pone un limite al permitted set di capabilities che si possono concedere da parte di un file eseguibile
c. Si possono anche limitare le capabilities che un thread può mettere nel suo inheritable set usando capset(); ciò significa che un thread non può inserire una capability nell’inheritable set anche se sta nel suo permitted set.

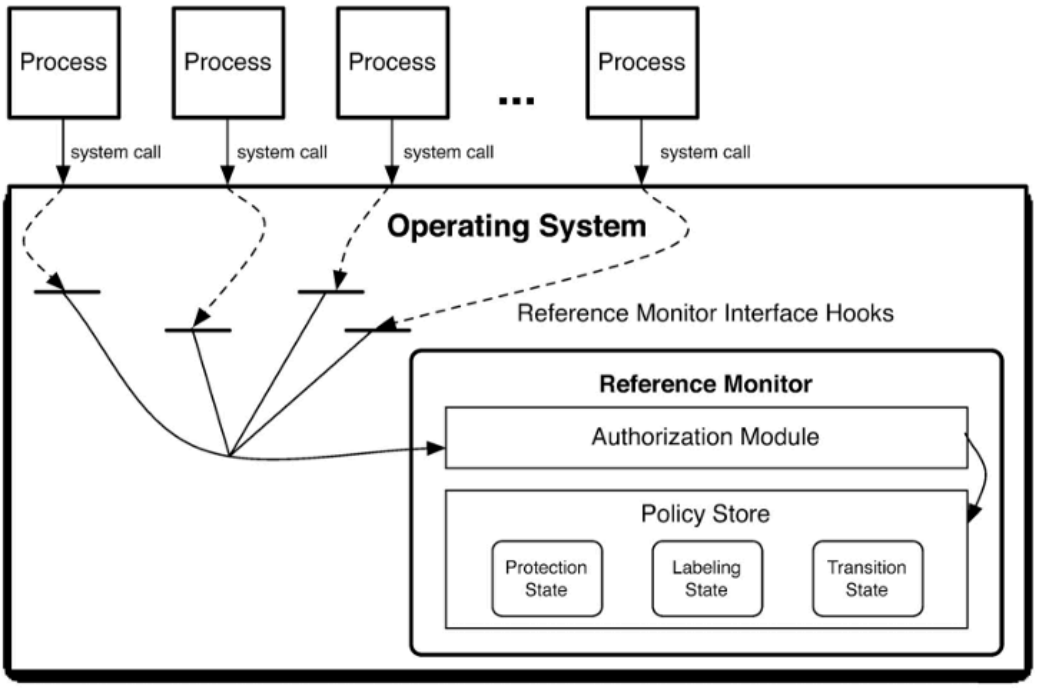
Tramite le capabilities stiamo quindi limitando la superficie di attacco.

Security modules:
Ciò che abbiamo visto finora segue dei principi di natura generica, tutto viene maneggiato a livello utente.
Sviluppare nuove capabilities richiede di essere in un account super user.
Molto è demandato ad una progettazione appropriata del sistema.
I “security modules” non sono altri che dei frameworks di sicurezza a livello kernel.
Ci possono essere richieste di sicurezza diverse a seconda del sistema in questione.
Non c’è un’unica singola soluzione, quindi per risolvere il problema di una mancanza di “accordi” è necessario uno strumento di sicurezza.
A seconda del sistema che si amministra si vuole essere liberi di avere la garanzia di un certo livello di sicurezza.

Il framework Linux Security Modules offre un'implementazione generica a basso overhead per supportare il Mandatory Access Control.
Un modello di sicurezza diversa può essere messo in atto semplicemente caricando un altro security module; in questo modo abbiamo un supporto non invasivo.

→Esempio di accesso ad un file:

Reference Monitor



Abbiamo dei processi generici (privileged o unprivileged), che si interfacciano con il SO ad esempio per aprire un file (systemcall open()).
Abbiamo un execution path, che è quello tradizionale finchè non troviamo un “interface hook”. L’esecuzione tradizionale in una systemcall open ha a che fare con il fatto che il SO guarda nella propria cache per trovare informazioni sulla risorsa che sta cercando di aprire, per trovare l’inode (astrazione sw di un file nel sistema), poi ci sono check sulla validità dell’inode ed infine si fanno i check DAC, che vengono prima dei check MAC. Noi non stiamo sovrascrivendo ciò che fa tradizionalmente il kernel, perché vogliamo causare il minimo overhead. Quindi i security modules coesistono con i check tradizionali DAC implementati dal SO.
Ad un certo punto incontriamo qualche “hook”; dopo i checks DAC abbiamo gli hooks, se un security module è configurato tramite una systemcall chiamiamo il security module, a cui viene chiesto se è d’accordo con l’attuale catena di esecuzione, ovvero se l’utente può accedere la risorsa. A seconda dell’implementazione il security module può fare qualsiasi cosa. Quindi infine il security module garantirà o no l’esecuzione, se non la permette la open ritornerà un errore.
Abbiamo minima intrusività, nessun overhead addizionale, come quello che conseguirebbe dall’introduzione di nuove systemcalls.
l’organizzazione e la progettazione di un security module è critico, ma ancora di più lo è l’organizzazione delle policies da controllare.
Gli hooks sono ristretti, non possono impedire al kernel di eseguire qualcosa che farebbe se non ci fossero security modules, ma possono interrompere l’esecuzione ad un certo punto.
Gli hooks possono fermare l’esecuzione di specifici code paths nel kernel.

security_operations→ struttura dati che contiene una serie di function pointers per registrare i diversi hooks.
A boot time stiamo essenzialmente registrando le operazioni di default, quando viene caricato un security module chiama la capability di “register” a livello kernel per rimpiazzare alcuni hooks di interesse.
Possiamo montare un singolo security module per volta, questo ha senso perchè security modules diversi possono avere strutture differenti.
Se un security module viene montato allo startup non se ne può montare un altro che lo sovrascrive.
register_security() → per montare un altro security module c’è bisogno di riavviare il sistema.
Neppure l’amministratore può modificare le attuali policies di sicurezza.

Security fields:
Dal momento che i security modules sono di carattere generale, hanno bisogno di etichettare le risorse del sistema. Devono mantenere informazioni su soggetti/oggetti del sistema, quindi hanno bisogno di strutture dati addizionali.
Ad esempio per dire che si può accedere ad una porzione di un file si devono allegare al file dei metadati.
L’implementazione dei MAC è demandata al security module per limitare l’intrusività.
La struttura dati “security_operations” contiene dei function pointers per gestire i security fields.
Ad esempio un f.p. per dire se uno specifico soggetto può accedere un altro programma.
Questo ha senso poiché ciò che può essere fatto da un debugger può essere fatto da un attaccante.

Data Structure	Object
task_struct	threads
linux_binprm	Program being loaded
super_block	Filesystem
inode	Pipe, file, or socket
file	Open file
sk_buff	Network buffer (packet)
net_device	Network device
kern_ipc_perm	Semaphore, shared memory segment, or message queue
msg_msg	Individual message

I DAC vengono controllati prima dei MAC perchè si vuole evitare che un utente regolare possa cambiare gli attributi di uno specifico file.
I security modules fanno dei controlli in base a chi sei e cosa stai tentando di fare.
Si prepara l’insieme di hooks e si consegnano al kernel tramite la systemcall “register_operations”.

Ci sono cinque categorie di security hooks:

1. Task hooks→ per controllare se un'azione può essere eseguita
2. Program loading hooks→ forniscono un'interfaccia per decidere quale programma può essere lanciato
3. Inter Process Communication hooks→ per gestire le politiche di accesso alle funzioni IPC
4. Filesystem hooks→ forniscono controlli a grana più fine sugli oggetti del filesystem
5. Network hooks→ forniscono un'interfaccia per gestire dispositivi e oggetti di rete.

MAC su Linux:
S.M.A.C.K.
Tutto è basato su etichette associate a oggetti e soggetti.
ogni volta che un soggetto cerca di accedere un oggetto l'accesso è garantito in base alla corrispondenza delle etichette.
La configurazione è basata su regole esplicite.
Viene fornita una lista di etichette: soggetti, oggetti e capability di accesso (rwx).
Queste etichette sono mantenute all'interno degli extended attributes dei file.
per cambiare questi attributi c'è bisogno di avere la capability: CAP_MAC_ADMIN; ma per fare ciò c'è bisogno di eseguire il login, mentre noi non vogliamo che l'amministratore sia un utente del sistema, quindi questo è un problema.
L'amministratore delle policies è l'utente "problematico", perché può cambiare le proprie policies.
Con questo approccio abbiamo solo spostato il problema, non lo abbiamo risolto.
Le etichette di default sono: floor (_), star (*) e hat (^).
Funzionamento:

a. Ogni accesso da un soggetto identificato da * è vietato
b. Ogni accesso da un soggetto identificato da ^ è garantito in r,x
c. Ogni accesso su un oggetto identificato da _ è garantito in r,x
d. Ogni accesso su un oggetto identificato da * è garantito in r,w,x

S.M.A.C.K implementa uno pseudo filesystem: SMACKFS
Ci sono diversi file che possono essere usati per configurare il sistema:

- access2: si usa per chiedere l'attuale configurazione. Per fare la query la scrivi in questo file e poi rileggendo si trova la risposta
- change-rule: si scrive questo file per cambiare i permessi di accesso associati con le labels
- load2: per specificare nuovi permessi di accesso
- onlycap: specifica etichette aggiuntive richieste dai processi per ottenere la capability CAP_MAC_ADMIN
- revoke-subject: per rimuovere tutti i permessi associati ad un certo soggetto; è fondamentale perché si possono revocare tutti permessi senza dover navigare l'intero filesystem.

Per risolvere il problema precedente si utilizza un demone sul sistema che ascolta connessioni remote dell'amministratore.
La capability di gestione del sistema può essere gestita solo da remoto.
L'amministratore delle policies non esegue il login, interagisce con il sistema solo tramite un proxy (il demone).

il problema è se il demone soffre di bug.
S.M.A.C.K. offre facilities per implementare ciò, ma non lo implementa lui stesso.

Tomoyo:
Non usa esplicitamente le labels per garantire gli accessi.
I percorsi dei file sono utilizzati implicitamente come etichette.
Dal momento che quando spawniamo threads in Linux abbiamo una domain transition, si può risalire a quale applicazione ne ha lanciata un'altra, quindi Tomoyo tiene traccia della catena di attivazione dei processi.

Domain ACL example

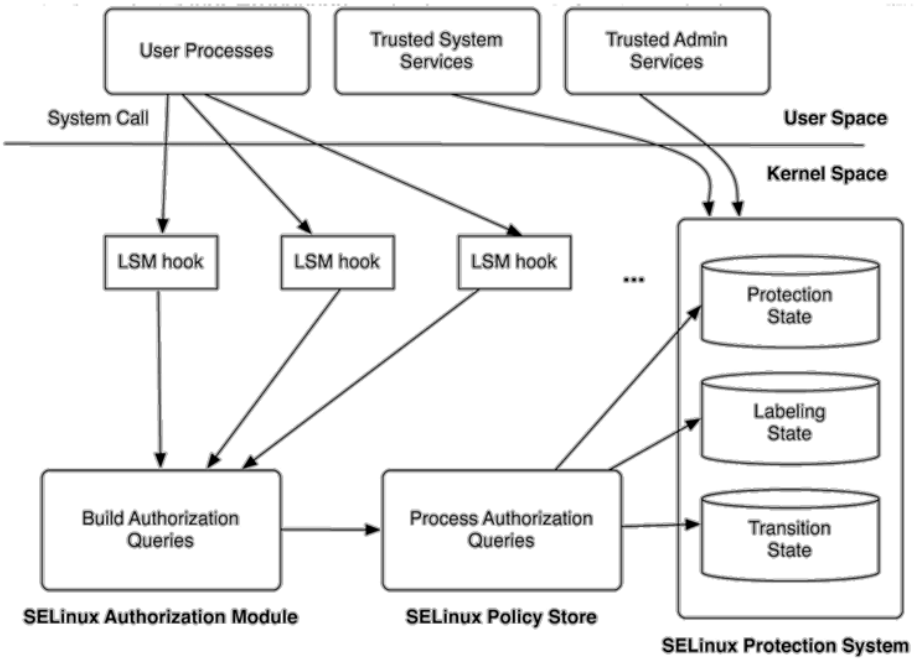

```
<kernel> /usr/sbin/sshd /bin/bash
allow_execute /bin/ls
allow_read /home/pellegrini/.bashrc
allow_read/write /home/pellegrini/.bash_history
...
<kernel> /usr/sbin/sshd /bin/bash /bin/ls
allow_read /etc/group
allow_read /etc/nsswitch.conf
allow_read /etc/passwd
```

E' molto prolisso nel dire che cosa si può fare.
Quando si esegue un'azione il contenuto cambia.
La configurazione può risultare complessa.
Si può anche discriminare se si sta accedendo il sistema localmente o da remoto.
Qui non stiamo specificando i soggetti, quindi non possiamo dare capabilities diverse in base ai soggetti; ma se si usano bene i DAC si può evitare, dato che vengono eseguiti prima dei MAC, il problema sta nel fatto che i DAC possono essere modificati dall'utente.
L'altro problema è che le etichette possono essere molto grandi, quindi la configurazione può essere complessa.

04/12/2023 (lez.27)
I due approcci MAC visti finora utilizzano le etichette.
AppArmor:
L'idea è che nei sistemi Unix gli utenti interagiscono lanciando processi, quindi l'idea di base sono i tasks.
Ogni processo (o gruppo di processi) può avere un proprio profilo, che è un "confinamento" del programma.
I "tasks profiles" sono realizzati e caricati a partire dallo user space.
Se un programma non ha profili non c'è alcun confinamento, si usano solo le ACL (DAC tradizionale Linux). Proprio per questo motivo si parla di "selective confinement".
L'approccio principale è la semplicità di utilizzo.

Si configurano profili per ogni programma e si impedisce all'amministratore di installare un nuovo programma.
E' una sorta di MAC.
Si fornisce un file sorgente "/etc/apparmor.d" in cui vengono inserite le policies scritte nel linguaggio "AppArmor Profile Language", questo file viene poi compilato in formato binario.
Per ogni processo si deve specificare cosa si può fare.
Se si vive in un profilo "bash" e si lancia un processo si può fornire un "child profile" (ad esempio ls); nel child si possono specificare permessi di accesso ai files e capabilities.

SELinux:



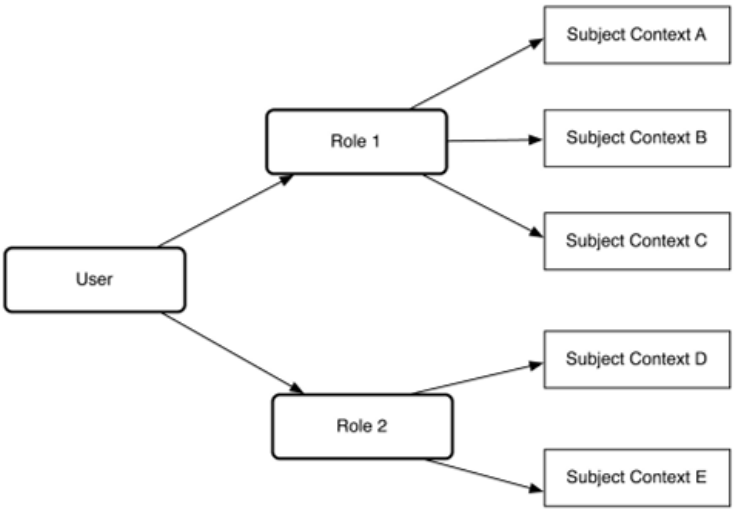
E' un security module che permette di definire a grana super fine cosa può fare una specifica azione.
Le applicazioni user space devono essere differenziare tra i servizi di sistema "trusted", che sono i demoni nello user space, che possono interagire con i security modules tramite degli entry point, abbiamo trusted applications che gli amministratori possono usare per interagire con SELinux, e poi abbiamo tutto il resto delle applicazioni-
I processi a livello utente sono soggetti ai security hooks.
Nel kernel space abbiamo una serie di moduli e componenti:

1. **SeLinux Protection System**, che garantisce e revoca accessi a risorse specifiche, che implementa una serie di "data stores"
    - Protection state→ per determinare quali risorse possono essere o meno accedute
    - Labeling state→set di etichette attualmente allegate alle varie entità del sistema
    - Transition state→ cambia le etichette in base a ciò che succede nel sistema
2. Gli hooks mirano il **SELinux Authorization Module**, che funziona come un DB in kernel mode; costruisce le "authorization queries" a partire dagli hooks, che sono essenzialmente function pointers. Le invocazioni agli hooks vengono tradotte in queries

3. Le authorization queries che ne escono passano attraverso il **SELinux policy store**, dove vengono processate e si decide se le queries devono essere permesse o negate.

Come funzionano questi moduli e come interagiscono tra loro?

1. *Convertire le chiamate negli hooks in authorization queries:*
   Si estraggono dai f.p. il soggetto, l'oggetto e l'operazione richiesta.
   Prendiamo come esempio l'accesso ad una risorsa.
   il soggetto sarà l'attuale processo che esegue la systemcall, l'oggetto che si vuole accedere è l'inode e l'operazione che si vuole eseguire dipende dal f.p. attuale, quindi dall'hook attivato dal kernel.
   Il soggetto e l'oggetto vengono convertiti dall'authorization module nel "contesto".
   Il concetto di contesto è essenzialmente una label, queste labels sono nel labeling o nel protection state e sono accessibili dal kernel space.
   Ogni volta che un utente esegue un'applicazione in base al soggetto l'utente può avere un ruolo diverso, che può essere soggetto a diversi contesti.
   Un utente può acquisire diversi ruoli, ogni ruolo può essere soggetto a diversi contesti.
   Abbiamo grana più fine, si danno etichette in maniera più versatile.



2. *Collegare le etichette alle diverse azioni:*
   Un policy statement è una stringa compilata nel kernel.
   Permettiamo esplicitamente ad un soggetto, a seconda dell'oggetto e della classe cui appartiene l'oggetto, di eseguire una serie di operazioni.
   Classe dell'oggetto legata al fatto che ogni oggetto ha un tipo di dato.
   - Example policy statement:

```
allow <subject_type> <object_type>:<object_class> <operation_set>

allow      user_t    passwd_exec_t:file          execute
allow    passwd_t    shadow_t:file               {read write}
```

In questo esempio al programma “passwd_t” viene dato l’accesso al file “shadow_t”, che può essere acceduto in read write mode.

- Labeling state:

Mappa un oggetto ed un utente ad una label.
Definisce come le nuove risorse debbano essere etichettate inizialmente: ogni volta che si crea un oggetto lo si vuole etichettare.
Nel contesto del labeling state si usa il filepath per descrivere soggetti ed oggetti in termini di contesto (similmente all’id in Tomoyo).
La struttura è:

```
<file path expr>        <context>
/etc/shadow.*           system_u:object_r:shadow_t:s0
/etc/*.*                system_u:object_r:etc_t:s0
```

Il contesto è essenzialmente un albero di etichette.

- Transition state:

Si può cambiare una label nel tempo, in base all’esecuzione di una qualche azione.
Si può specificare un tipo di transizione, si può fornire un creator_type ed un default_type per le etichette attaccate ad un oggetto.
A seconda del soggetto che esegue un’azione si possono realizzare diverse transizioni.

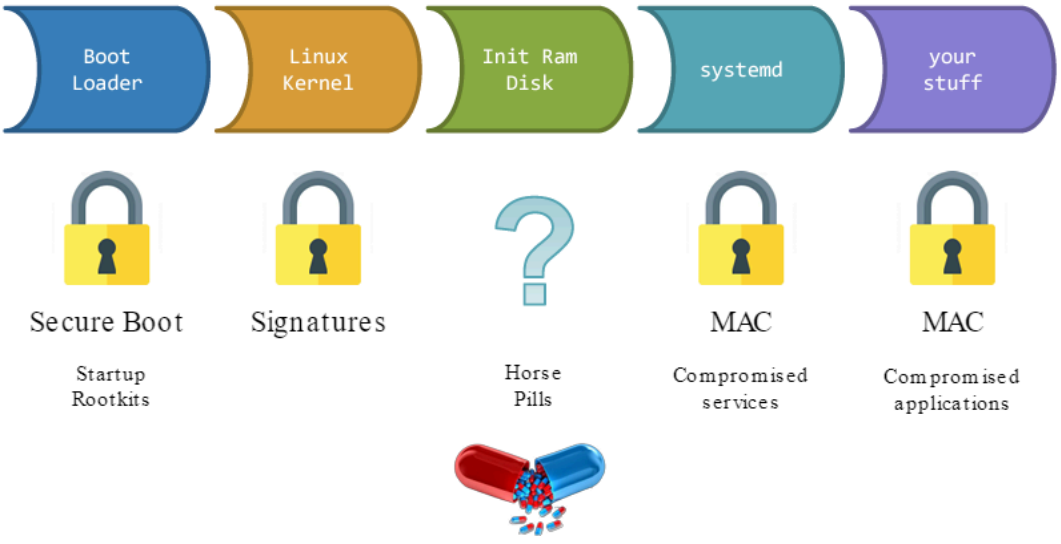
```
type_transition <current_type> <executable_file_type>:process <resultant_type>

type_transition user_t          passwd_exec_t:process          passwd_t
```

Di default quando un utente crea un certo file avrebbe il tipo “executable_file_type”, ma dato che c’è una transizione di tipo “type_transition” deve essere di tipo “resultant_type”.
Il current_type è “user_t”, il soggetto è un processo, se il processo ha label “user_t” può diventare “passwd_t” quando esegue un programma con label “passwd_exec_t:process”.

Ciò che abbiamo visto sono 4 diverse implementazioni di ciò che dovrebbe fare MAC.

Boot time security:



Cerchiamo di capire qual è il lifetime della macchina da quando la accendiamo a quando effettivamente si arriva allo user space finale, essenzialmente cosa succede quando accendiamo il nostro computer?
Si accende il computer, poi il bios fa dei controlli sull'hw (bios=firmware), essenzialmente il **boot loader** inizia la sua esecuzione, il suo incarico è caricare il kernel Linux.
Il **kernel Linux** è la componente essenziale del SO, essenzialmente è un file sul disco, al suo interno ha del codice che scopre l'HW, inizia i servizi e dà il controllo allo user space.
La diversità dell'hw che abbiamo sulle nostre macchine è così grande che è impossibile che il kernel abbia al suo interno tutti i driver possibili. I driver sono pezzi di codice nel SO che dicono al SO come interagire con i device hw.
Il SO deve caricare il prima possibile i driver necessari, quindi tipicamente quando si installa qualche driver possono avere due obiettivi diversi, alcuni possono essere relativi a devices di importanza secondaria, caricati dopo che l'avvio del SO viene completato, altri invece sono fondamentali per la configurazione del SO stesso, ad esempio quelli per leggere e scrivere file su disco; il boot loader caricherà una parte molto ridotta del SO, che capirà quali driver sono necessari, verranno caricati e poi il SO finalizzerà il proprio startup (stage 1, 1.5 and stage 2 bootloading, drivers diversi necessari a diversi stages).
Qui entra in gioco l'initial RAM disk, è essenzialmente un file che contiene tutto il filesystem, tutti i file salvati sul disco vengono copiati in memoria principale, questo contiene i drivers fondamentali per interagire con l'hw; il kernel monta questo filesystem per accedere altri file e portare a termine la configurazione. Data la diversità dell'hw che abbiamo se un driver è necessario per una certa macchina il kernel dovrà fare la build dell'initial RAM disk localmente per avere i driver necessari per il prossimo startup.
La prima applicazione con cui si interagisce è "systemd" in Unix, il login screen in caso di Windows.
Infine si caricano le nostre applicazioni.
Questa è la timeline dell'avvio del computer.

Dove sta la sicurezza in tutti questi pezzi?
Il bootloader si può considerare sicuro; è un'immagine firmata crittograficamente che viene verificata dal firmware.
Il kernel Linux è sicuro perchè anch'esso certificato da una signature, ogni volta che si scarica una nuova immagine di SO esso è correlato con una firma che si può verificare localmente.
Per quanto riguarda la sicurezza in systemd essa si può garantire per mezzo di security modules, possiamo configurare MAC; lo stesso vale per quanto riguarda le nostre applicazioni. MAC evita l'esecuzione di applicazioni malevole.
Per quanto riguarda l'initial RAM disk la situazione è più complessa.
Esso serve per:

1. Caricare i moduli
2. Rispondere agli hotplug events, mandati al SO per trovare il driver corretto
3. Fare il cryptosetup in caso di filesystem cifrato, deve fare il setup dei tools per accedere al filesystem cifrato, se è soggetto ad una password deve chiedere al software di richiedere la password
4. Trovare e montare il filesystem vero e proprio che verrà usato come root filesystem
5. Ripulire le azioni iniziali
6. Attivare il primo processo, che nel mondo Linux è "init"

Ciò che un init RAM disk può fare è vasto.

Horse pills attack:
E' un boot time attack che vuole modificare il contenuto del RAM disk.
Ogni volta che si assembla il RAM disk non ci sono signatures, perché viene creato localmente.
In un RAM disk abbiamo utilities minimali, alcuni moduli kernel.
Un RAM disk infetto può prendere il controllo della macchina molto prima che MAC e DACL siano messe a punto e può completamente compromettere la sicurezza dello userspace.
Questo attacco si serve di tecnologia basata su containers: tutto ciò che viene eseguito sul container non può guardare al di fuori, applicazioni ed utenti pensano di usare l'intera macchina.
Un RAM disk infetto può caricare i security modules, fare il cryptosetup, montare il filesystem attuale e guardare tutto ciò che avviene nel sistema, ad esempio vedere quali demoni sono in esecuzione, abbiamo già caricato i moduli quindi possiamo interagire con i dispositivi. Poi avvia un container prima che venga avviato lo userspace, a questo punto abbiamo due ambienti: il SO vero e proprio ed un container nel SO.
Dato che il RAM disk è fuori dal container può controllare entrambi gli ambienti.
Cosa può fare il RAM disk nel container?
Può simulare ciò che farebbe un SO legittimo:

1. Rimontare il filesystem proc per nascondere ciò che avviene al di fuori del container
2. Creare dei finti threads di livello kernel cosicché la vittima pensi di interagire con un kernel legittimo, anche se non è così perché il kernel vive fuori dal kernel. Threads con stesso nome dei demoni veri
3. Cleanup
4. Iniziare init process: l'utente vede solo ciò che sta nel container.

Fuori dal container può montare qualsiasi tipo di filesystem, può fare una fork e lanciare ad esempio una backdoor shell, ad esempio un'outdoor shell, in tal modo la macchina può essere esposta sulla rete e dal container non ci si rende conto di nulla perché tutto ciò che vede è l'interno del container.
Poi dice al container di spegnersi, essenzialmente facciamo kill dell'applicazione, il container si spegne, possiamo intercettarla e quando si riavvia la macchina l'utente pensa che non stia avvenendo nulla di male.
Si può osservare tutto ciò che accade nel container dall'esterno, quindi ad esempio si possono rubare i file scritti dall'utente.

Mitigazioni:
Ci sono una serie di cose che si possono notare all'interno del container.
Nel container possiamo vedere quali sono i namespaces associati alle applicazioni che lanciamo.
I thread kernel non sono generati dal kernel iniziale con pid pari a 0.
In generale bisogna fare un esame e revisione estensiva del sistema anche un esame esterno realizzato staccando l'hard drive.

Come prevenire questo attacco?
Non è semplice, bisognerebbe evitare di creare localmente il RAM disk, ma non è possibile far fronte alla complessità hardware in questo modo, i RAM disk sono l'unico modo in cui possiamo configurare il nostro sistema basato su diversi driver.

Ogni volta che il RAM disk viene ricreato sulla macchina l'attaccante deve re-installare il malware.

OS level network security:
Ci sono alcune capabilities del SO interessanti dal punto di vista della rete, anche se oggi non sono ancora molto utilizzate, ma che potrebbero migliorare di gran lunga la sicurezza del nostro SO.
Si possono usare le ACL per i servizi di rete, i soggetti di alcune azioni sulla nostra macchina possono essere ad esempio gli indirizzi da cui provengono le richieste alla nostra macchina.
Se vogliamo implementare MAC in maniera propria dobbiamo garantire l'accesso anche da remoto per evitare che l'amministratore sia un utente del sistema.
Essenzialmente i SO Posix forniscono una serie di capabilities per implementare CAL basate su indirizzi di rete.
Usano super server, che sono applicazioni per spawnare server addizionali o containers.
inetd (internet daemon) e xinetd (extended internet daemon) sono esempi di super server.

UNIX inetd:
Super daemon che controlla quali applicazioni dovrebbero ascoltare su una specifica porta, ogni volta che qualche richiesta arriva su una specifica porta non viene immediatamente inviata all'applicazione in ascolto su quella porta; inetd è in ascolto su tutte le porte, esso usa dei file di configurazione in cui ha porte, protocolli e applicazioni che deve essere invocata relativamente a una porta e un protocollo; ogni volta che trova associazione con porta e protocollo lui lancia la relativa applicazione.
La configurazione di inetd specifica: service name, socket type (e.g. stream), socket protocol (e.g. TCP), service flag (wait/nowait) (concurrent or not) , l'id utente da associare all'istanza di servizio in esecuzione, il path del file eseguibile ed i suoi argomenti.

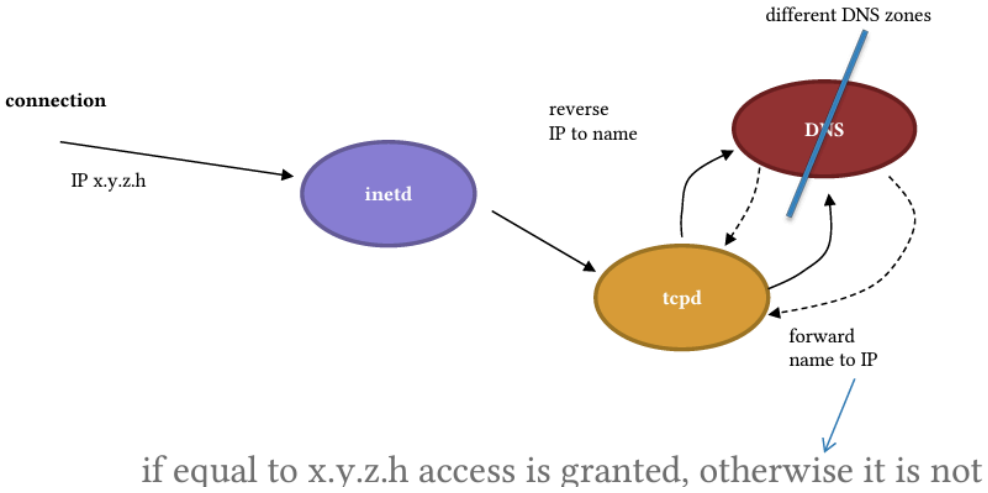
UNIX xinetd:
Ha capabilities in più:

- AC basato su indirizzi (e.g. IP che possono contattare specifici servizi)
- AC basato sul tempo, per quanto tempo una connessione è abilitata
- log completo di eventi a run-time, uno dei pilastri della sicurezza è avere log per capire cosa è andato male nel caso in cui qualcosa vada male
- prevenzione a DoS mettendo limitazioni su:
    - numero di istanze per servizio
    - numero totale di istanze di server
    - taglia del log file
    - connessioni ad una risorsa da parte di una singola macchina.

La configurazione è in un file diverso, /etc/xinetd.conf; ci sono utilities per generarlo automaticamente in maniera corretta, perchè può risultare complesso.

Ogni volta che una richiesta arriva a inetd e xinetd, xinetd farà lo user space AC, poi invoca il tcp daemon per processare la richiesta.
tcpd determina se una richiesta specifica deve andare avanti.
La configurazione originale di tcp è basata su /etc/conf/host.deny e /etc/conf/host.allow, dove dice quali sorgenti possono accedere uno specifico modulo daemon.

tcpd fa anche un altro controllo di sicurezza, che permette di accorgersi se c'è DNS tampering.
Il record associato ad un nome DNS può essere stato alterato; quindi controlla rDNS, ovvero fa la query inversa per ottenere il nome simbolico a partire dall'indirizzo IP.
Questo funziona perché la query diretta ed inversa sono eseguite da zone diverse; la zona che si occupa delle reverse queries è il proprietario stesso del record DNS in questione.



Se si accorge del DNS tampering tcpd nega l'accesso.
Oggi non si usa più per il principio KISS (keep it simple stupid): inetd passa i dati a tcpd tramite file, quindi essenzialmente i dati che vengono dalla rete vengono rediretti a stdout e letti da stdin, ma dato che usiamo frameworks "stupidi" che non possono leggere da stdin, ma usano direttamente socket non si usa tcpd.

11/12/2023 (lez.30)
Antivirus:
Parti di software in esecuzione in background.
Tipicamente hanno una serie di obiettivi ad alto livello.
Cercano di prevenire le infezioni, rilevano il sw malevolo e se il mw è già nel sistema fanno uno scan del sistema e lo rimuovono a posteriori.
Ripulire dal MW è più complesso che prevenirlo, infatti il MW tenta di diventare un thread permanente.

Problemi:
Il MW cerca di nascondersi dagli antivirus tramite tecniche di evasione, in generale gli antivirus devono tenerne conto.
La superficie di attacco è molto vasta: applicazioni, servizi di sistema, sistemi operativi, email, rete,...
Il numero di MW in circolazione è enorme, se ne vedono migliaia di nuovi esempi al giorno, inoltre lo scopo del MW è cambiato rispetto al passato, passando dalla soddisfazione intellettuale al denaro, c'è stato un grosso incremento di ransomware.
Gli antivirus sono scritti in linguaggi compilati perché devono essere molto veloci, il problema è che lo sviluppo tramite linguaggi compilati è più complesso, quindi anche negli antivirus possiamo osservare dei bug: il malware può sfruttare questi bug per rendere l'antivirus inutilizzabile.

Le tecniche che un AV utilizza per riconoscere un MW sono cambiate negli anni. All'inizio gli AV erano essenzialmente degli scanner che leggevano i bytes cercando di identificare dei pattern riconducibili ai MW.
I moderni AV hanno a che fare con una superficie di attacco molto ampia e con molti più MW, quindi devono continuamente eseguire scan in background.
Ogni volta che viene creato un nuovo file viene mandata una notifica ad un componente persistente.
Gli AV installano anche dei firewalls che guardano ciò che accade nel browser.
Un AV riceve migliaia di nuovi MW ogni giorno, quindi rintracciare i pattern non è fattibile.
Quando usiamo un AV gli mandiamo esempi di MW.
Il MW essendo a conoscenza della presenza di AV cerca di nascondersi con meccanismi di offuscamento.
Oggi anziché focalizzarsi sui pattern si usano le euristiche: gli AV danno un punteggio alle applicazioni.

Concetti principali:

1. Kernel→ è il core di un AV è sempre attivo in memoria
2. Scanner manuale (CLI o GUI) → possono anche essere residenti in memoria e real time
3. Drivers che filtrano il filesystem→ dato che i SO sono una superficie di attacco i driver sono critici, perché possono far andare in crash il SO. Ci sono dei servizi di sistema per supportare le notifiche real time e attività di detection relative a ciò che avviene nel filesystem
4. Driver che filtrano la rete→ possono accorgersi di attacchi da parte dei botnet. Costruiscono localmente grafi delle connessioni sul nostro computer
5. Signature DB→ dato che ci sono moltissimi threads e che cambiano nel tempo in questo database si tiene traccia delle signatures dei file malevoli (pattern).
   Dato che oggi non si usano più i patterns ci sono i plugins: carichiamo il programma anziché il pattern. Il programma fa una serie di controlli per identificare un comportamento malevolo del sw
6. Unpacker→ dato che il MW utilizza il packing come meccanismo di offuscamento gli AV implementano una serie di unpacker. Ci sono plugins che eseguono ciò che il MW fa all'inizio per scoprire cosa fa
7. Interpreti di formati di file→ gli AV devono saper interpretare una serie di formati di file; può essere complicato, dato che molti formati sono "open", per avere a che fare con i quali bisogna fare reverse engineering; inoltre ci possono essere diversi livelli di nesting, ad esempio in caso di pdf possiamo fare nesting di pdf in altri pdf. Per osservare questo alto livello di annidamento c'è bisogno di molto tempo, inoltre se il MW scopre fino a che livello gli AV processano l'annidamento ne aggiungono qualche altro livello
8. Packet filters
9. Meccanismi di auto-protezione→ il MW potrebbe anche uccidere l'AV, quindi servono meccanismi di protezione dai segnali di kill. Il MW inoltre potrebbe attaccarsi all'AV così da avere sempre una componente in esecuzione sul sistema, per evitare questo si usa la randomizzazione, l'ASLR è una capability di tutti i SO oggi
10. Emulatori→ dato che un obiettivo degli AV è quello di fermare la diffusione dei virus, gli emulatori servono per emulare altre architetture su cui far girare il MW.

Unpackers:
Il MW vuole nascondersi dagli AV.
In caso di ricerca di pattern gli AV aprono un file, leggono i bytes e cercano determinate sequenze. Il problema è che il MW potrebbe comprimere il file e fare la decompressione nel momento in cui esegue sul sistema; in tal caso l'AV può fare la decompressione allo stesso modo.
Approccio più complesso è quello dell'encryption: serve la chiave giusta per decifrare.
Il codice originale viene trasformato dal formato originale, quindi l'AV vede solo una sequenza di bytes randomici.
Il MW per eseguire il codice usa gli unpackers, pezzi di codice che spacchettano il codice quando si accorgono che il MW sta eseguendo.
Dopo l'unpacking l'esecuzione passa dall'unpacker al codice originale spacchettato.
Cosa fa l'AV in questo caso?
Indizi che il codice è impacchettato sono:

1. Mancanza di IAT ed import→ la IAT dice al sistema quali librerie saranno usate. L'unpacker non usa sw di terze parti, quindi solitamente "embedda" l'algoritmo per fare l'unpacking nel codice.
2. Nomi delle sezioni non standard: i compilatori usano nomi standard, quindi se non sono standard significa che non sono stati usati compilatori standard
3. Le sezioni hanno dimensioni ridotte sul disco, ma occupano molta memoria virtuale; significa che ci sono sezioni vuote, necessarie a spacchettare qualcos'altro
4. Se non ci sono parole comuni nella sezione "data" non c'è interazione con l'utente
5. Sezioni rwx; non tipico nel codice comune, perchè si adotta il principio di protezione della memoria, quindi rx o rw tipicamente; se ci sono queste sezioni significa che c'è codice modificabile a runtime
6. Se si osserva il codice dell'unpacker ad un certo punto passerà il controllo al codice del mw vero e proprio, quindi ci sarà un salto ad una qualche strana locazione di memoria: questo è il trampolino per dare il controllo al mw originale.

Se accade più di qualcosa in questo elenco indica che il mw è in esecuzione.

Signatures:
Sono pattern che cercano di matchare dei file con dei pezzi di malware noti.
L'unpacker nasconde nel codice la chiave per eseguire lìunpacking, fare lo scan dell'intera sezione data per trovare la chiave avrebbe un impatto sulle performance, diverso è se si conosce il punto in cui sta la chiave, in tal caso si fa solamente lo scan di alcune porzioni di codice.

Tecniche di evasione:

1. Ineffective code sequences→ il mw usa codice polimorfo, che si modifica ogni volta che infetta una macchina, lo stato architetturale resta lo stesso, ma si inseriscono azioni che non fanno nulla
2. Code transposition→ si suddivide il codice in porzioni e si rimescolano; per eseguire il codice nell'ordine corretto si utilizzano una serie di "jump"; questo è il cosiddetto "spaghetti code". Per identificare l'utilizzo di questa tecnica si deve ricostruire il flusso di esecuzione, ma è costoso.

In generale il concetto di virus che modifica se stesso è interessante.
Polimorphic virus→ cerca di nascondersi con packing e cifratura, ogni volta che infetta una nuova macchina utilizza un "packing" differente.
Si possono avere diversi cifratori e decifratori e chiavi, combinabili tra loro in molteplici modi.

Metamorphic virus→ il MW stesso ha della logica che permette di generare una versione totalmente differente di alcuni algoritmi nella prossima generazione: algoritmi totalmente diversi, ma con lo stesso comportamento: la semantica resta la stessa.
Solo una parte è il payload del virus, il resto è ciò che serve per modificarsi e replicarsi.

Firme comportamentali→ dobbiamo focalizzarci su cosa fa il mw nel payload, quindi farne un'analisi comportamentale; essa ha due nature:

1. Ispezionare la sequenza di systemcall
2. Ispezione a livello di istruzioni

Se osserviamo la sequenza di istruzioni guardandole una per volta non capiamo molto, dobbiamo correlarle per trovare pattern comportamentali.
Detection di codice malevolo tramite istogramma:

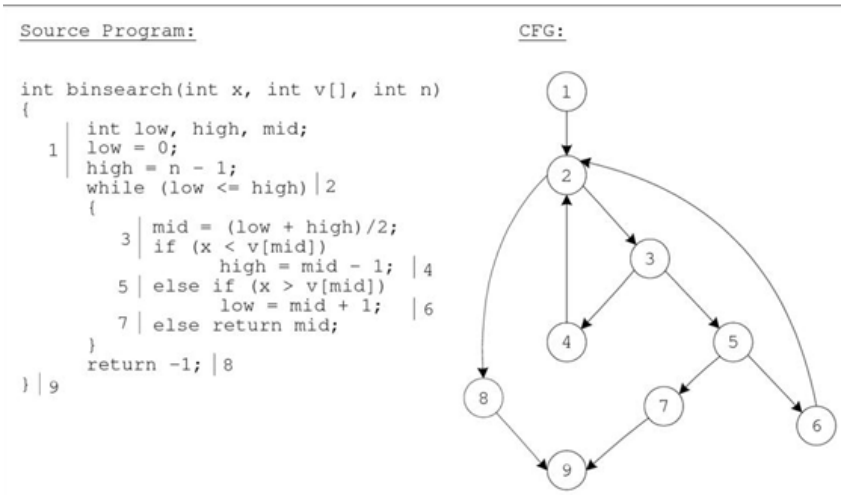
1. Tracciare l'occorrenza di istruzioni
2. Tracciare l'occorrenza di coppie di istruzioni
3. Usare l'istogramma per matchare le signatures

Anche se il codice è mescolato implementa una certa logica.
Questo approccio si usa in ambienti simulati: eseguo il mw in una sandbox affinché si spacchetti da solo.

18/12/2023 (lez.33)
Heuristic engines:
Si basano su modelli di machine learning.
L'applicazione viene eseguita in una sandbox e l'AV osserva come il MW interagisce con l'ambiente esterno.

1. Sequenze di chiamate API → l'interazione di un malware con il sistema operativo può essere un'indicazione delle azioni che l'applicazione sta compiendo.
   Tipicamente basato sulla classificazione: milioni di malware e benignware utilizzati come training set
2. Sequenze di opcode→ gruppi di istruzioni macchina possono essere un'indicazione delle operazioni che l'applicazione sta eseguendo. Le applicazioni vengono analizzate per determinare la frequenza degli opcode nei binari. La frequenza dei termini ponderata (weighted term frequency) è utilizzata per trovare le sequenze più rilevanti di opcodes
3. Call function graph→ cerca di estrarre blocchi di base dall'applicazione, se troviamo connessioni tra questi blocchi possiamo capire la semantica del codice. I blocchi di base delle applicazioni sono quelle che, qualora invocati, indicano che è stato invocato qualsiasi altro blocco.
   Due versioni della stessa applicazioni avranno lo stesso CFG, anche se pensiamo all'ineffective code perchè i basic blocks saranno gli stessi.
   Quando costruiamo un CFG consideriamo solo 4 tipi di istruzioni, quelle che cambiano il flusso di esecuzione: non-conditional jumps (jmp), conditional jumps (jcc), function calls (call) e function returns (ret).
   Ci focalizziamo solo su queste perché vogliamo essere veloci.

Se pensiamo allo spaghetti code due blocchi in cui da uno si salta all'altro diventano un solo blocco.
Vogliamo ridurre la complessità del codice; rimuoviamo i nodi relativi a salti non condizionali (jmp) e tutti i loro predecessori vengono collegati al suo successore.
Stiamo facendo una trasformazione che potrebbe non riflettere totalmente l'esecuzione originale.
Il grafo risultante funge da signature.
Visitando il grafo lo si confronta con la signature che si ha, questo è un processo molto rapido.
Possiamo avere falsi positivi e falsi negativi, ma comunque risulta essere una tecnica molto efficace.



4. n-grams→ sequenze continue di parole che si possono trovare in un testo. Si confrontano diverse sottosequenze del testo e si cercano quelle che occorrono molteplici volte. Si osservano le parole che si ripetono: si cerca di rendere più prolisso il testo originario.
   Si usa una finestra scorrevole parola per parola e si raggruppano così n parole per volta.
   Quando applichiamo questa tecnica ad un file binario la applichiamo ai bytes.
   Applichiamo poi dei modelli di machine learning per classificare il codice malevolo: si estraggono gli n-grams dal codice e li si inseriscono nel classificatore.
   Per migliorare le performances si usano tecniche di boosting.
   Ne risultano dei falsi negativi, ma pochi falsi positivi.
   Le DNN si comportano bene in questo contesto; dato che lavorano bene con immagini una tecnica usata è quella di convertire il binario in un'immagine (byteplot).
   Tipicamente l'entropia del mw è elevata, soprattutto se impacchettati, ciò è dovuto ai bytes randomici relativi alle tecniche di cifratura.

Emulatori:
Tipico approccio degli AV è eseguire il MW in delle sandbox, emulatori di un sistema.
Mentre l'applicazione è in esecuzione di osserva il suo comportamento.
Le sandbox hanno al loro interno l'implementazione dell'ISA e di APi di SO.
Lavora facendo ciò che fa un debugger.

L'AV esegue l'app per un periodo di tempo limitato (solitamente non più di 11s), questo serve per avere un compromesso con l'utente, ma ha uno svantaggio: se un'applicazione inizialmente ne emula una legittima l'AV non si accorge che è MW.

Fingerprints→se l'emulatore reimplementa tutte le systemcall diventa esso stesso un SO, quindi per evitare ciò gli AV ritornano i valori di ritorno di default delle systemcall, quindi un valore "hard-coded". Il MW può controllare questo valore di ritorno per capire se è in esecuzione in una sandbox ed in tal caso modificare il proprio comportamento, fingendo di non essere sw malevolo.

Encrypted traffic analysis:
Molti attacchi vengono dalla rete, trovano bug nel SO per penetrare il sistema. Alcuni MW usano codice cifrato per nascondersi dagli AVs.
Gli AV fanno analisi del traffico cifrato.
Che cosa fanno?
Lanciano un attacco MITM e installano un certificato firmato da una CA affidabile, oppure creano nuovi certificati per ogni nuovo sito visitato dagli utenti, firmandoli con una CA valida, questo viene fatto in Windows installando un nuovo root certificate.
Problema: questo approccio potrebbe ridurre la sicurezza:

- 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

Ci si deve fidare dell'AV?
Gli AVs sono pur sempre pezzi di codice, quindi possono essere soggetti a bug, inoltre come abbiamo visto alcune tecniche addirittura riduco la sicurezza del nostro sistema, quindi affidare totalmente la sicurezza del nostro sistema ad un AV non ha senso.

Database security:
E' l'insieme di meccanismi e migliori pratiche che proteggono i dati memorizzati nel database da minacce intenzionali o accidentali.
Security policy→ descrive una misura di sicurezza applicata
Bisogna utilizzare i meccanismi di sicurezza del DBMS sottostante per rafforzare le policy, ma anche usare policies di sistema e gestione esterne al DBMS per rafforzare la sicurezza dei dati.

Bisogna comprendere le questioni di sicurezza in:

- un ambiente di sistema di database generale e un'implementazione specifica del DBMS
- considerando le problematiche di sicurezza del database nel contesto dei principi generali di sicurezza
- considerare le problematiche relative sia all'archiviazione del database che alla comunicazione del sistema di database con altre applicazioni, la comunicazione è importante in quanto sottintende il coinvolgimento di terze parti.

SQL injection→ attacco che deriva dal fatto che l'utente interagisce con un DB tramite stringhe. L'input utente viene inserito all'interno di queries SQL, ciò è totalmente insicuro. Anche se una stringa è corretta dal punto di vista dell'utente può non esserlo dalla prospettiva del db (piccolo Bobby Table docet).

Soluzione a questo problema: anziché concatenare stringhe usare i "prepared statements", ovvero delle queries incomplete che utilizzano dei placeholders al posto dei parametri attuali.
e.g. INSERT INTO score (event_id,student_id,score) VALUES(?,?,?)
Non bisogna permettere all'utente di cambiare la semantica di una query.
Il DBMS analizza, verifica e compila la query incompleta; poi quando i parametri effettivi sono disponibili, possono essere associati alla query pre-compilata.
Quando il DBMS riceve i parametri effettivi, viene eseguita la query completata effettiva e vengono prodotti i risultati.
L'altra soluzione per la SQL injection sono le "stored procedures" rendono il codice non manutenibile, perchè dobbiamo avere a che fare con tantissime procedure, che rendono il codice meno leggibile; per questo motivo sono meglio i "prepared statements".

Autenticazione e autorizzazione nei DB:
Il principio dei privilegi minimi va applicato anche ai DBMS.
Se tutti eseguono sul DB con privilegi di root si può fare di tutto, ciò causa problemi dal punto di vista della sicurezza, perché se l'applicazione che si connette al DB è buggata può fare qualsiasi cosa.
Bisogna dare i privilegi in base a ciò che un utente ha necessità di fare.

Autenticazione→ Un meccanismo che determina se un utente è chi dice di essere. Un amministratore di sistema è responsabile di concedere agli utenti l'accesso al sistema creando account utente individuali. La maggior parte dei moderni sistemi DBMS permette solo agli utenti autenticati di stabilire una connessione con il DBMS

Autorizzazione→ L'assegnazione di un privilegio che consente a un utente autenticato di avere un accesso legittimo a un sistema. Sono al pari degli access control. Il processo di autorizzazione coinvolge l'autenticazione dell'utente che richiede l'accesso agli oggetti. Questa è una caratteristica importante dei moderni sistemi DBMS per implementare il PoLP.

SQL access control:
Lo standard SQL permette di specificare chi può accedere una porzione del DB ed in che modo può farlo.
L'oggetto dei privilegi sono le tabelle.
Di default:

1. Chi crea una risorsa ha privilegi completi su di essa
2. L'amministratore ha privilegi completi su tutto

Il modo appropriato per progettare un database è creare diversi utenti, ognuno con dei permessi diversi su porzioni diverse del DB.

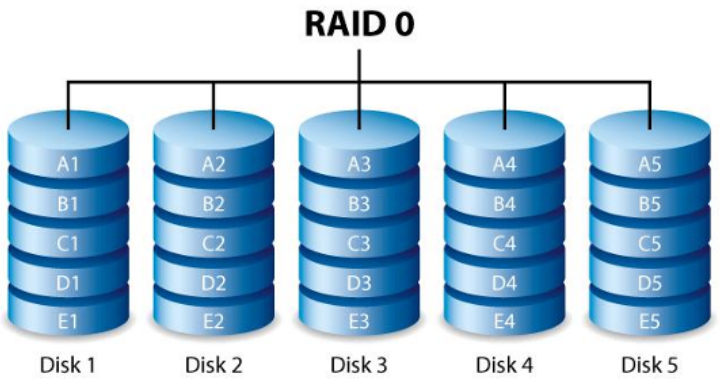
- Granting privileges:
  grant < *Privileges* | all privileges > on *Resource*
  to *Users* [with grant option]
  - grant option determines whether a certain privilege can be transferred to other users
- Revoking privileges:
  revoke *Privileges* on *Resource* from *Users*
  [restrict | cascade]

- SQL-99 introduces the concept of *role*:
  - a set of privileges
  - a role can be associated with a specific user

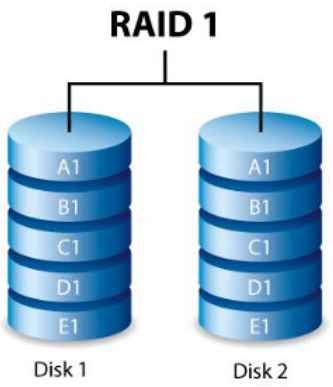
Viste→ Esse sono delle particolari relazioni nel database richieste da uno specifico utente, non sono realmente esistenti in un DB. Sono anche un modo per garantire sicurezza, in quanto permettono di rendere visibile a ciascun utente solo ciò che deve poter vedere.
Una query su una vista è identica ad una query su una qualsiasi tabella.

RAID:
La ridondanza è necessaria per soddisfare la “dependability”.
COsa fare se una macchina si rompe?
La sicurezza ha a che fare con il fatto di essere “operativi”.
Un approccio alla ridondanza è il RAID, che può essere implementato a livello sw o hw.
Si tratta di raggruppare diversi dischi poco costosi in un unico array: la probabilità di fallimento non si somma, in quanto sono indipendenti tra loro.
Dobbiamo coordinare il comportamento di molteplici componenti poco efficienti.
Le tecniche che si usano sono associate al “data striping”.
Sui dischi si possono fare due operazioni di base: read e write.
Dobbiamo gestire la concorrenza.
In cosa consiste il data striping? Distribuire su dischi diversi blocchi di file.
Solitamente la stripe size coincide con la taglia del blocco.
Ciò aumenta la disponibilità ed il throughput della concorrenza del sistema.

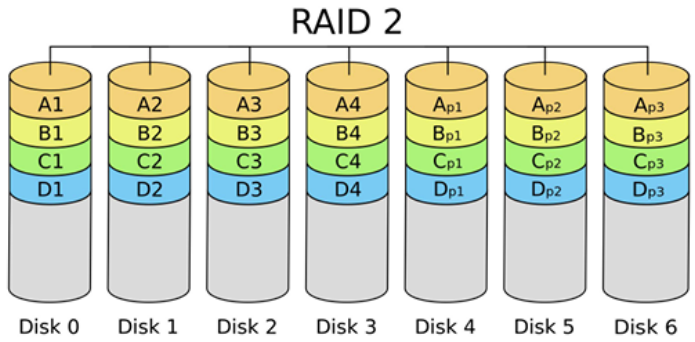
RAID-0:
Modo più semplice di organizzare dischi: nessuna ridondanza. Non è rilevante dal punto di vista della sicurezza.
Si suddivide un singolo file su molteplici dischi, quindi le operazioni richieste sono suddivise tra più dischi.



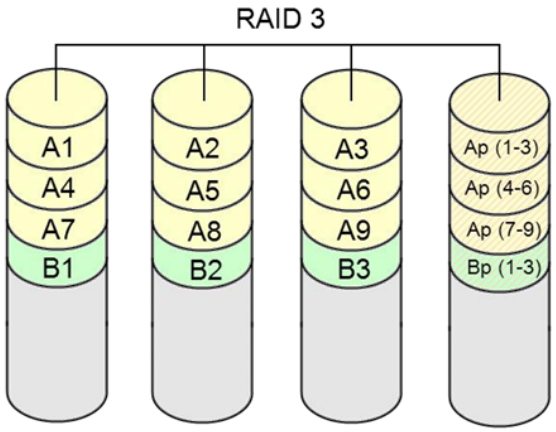
RAID-1:
“Disk mirroring”.
Ci sono molteplici dischi, ognuno ha una copia di alcuni file. Si possono servire più richieste in parallelo. E’ buono da un punto di vista di tolleranza ai guasti. Se un dato è particolarmente importante di può aumentare il numero di dischi.
Prima di fare il commit di una write bisogna scrivere su tutti i dischi, quindi le scritture rallentano il sistema.



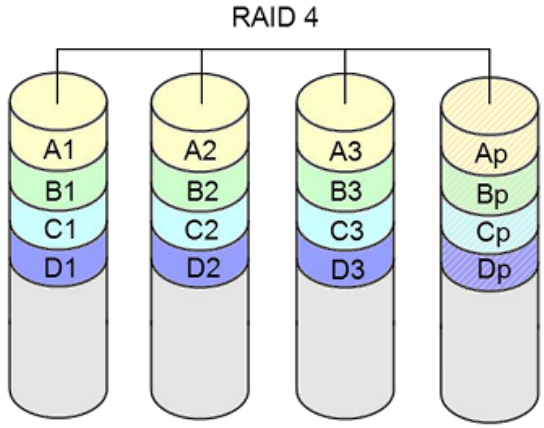
RAID-2:
Striping a livello di bit: scriviamo bit diversi su dischi diversi.
Le letture rallentano, per quanto riguarda le scritture le performance sono le stesse.
Si usano codici a correzione di errore (hamming code) per ricostruire il valore di un bit perso (se un disco fallisce).
C’è bisogno di un disk controller dedicato a livello hw, non c’è concorrenza su accessi diversi, ma allo stesso tempo tutte le testine di tutti i dischi si muovono insieme.
Non viene usato oggi.



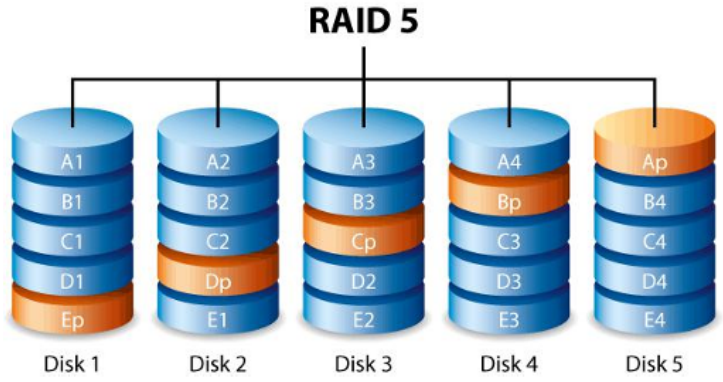
RAID-3:
Striping a livello di byte, usa la parità.
Si ottiene il miglior throughput, ma non garantisce concorrenza. C’è un disco aggiuntivo che contiene tutti i dati, quindi abbiamo due livelli di tolleranza ai guasti.
Le testine si sincronizzano per avere buone performances.



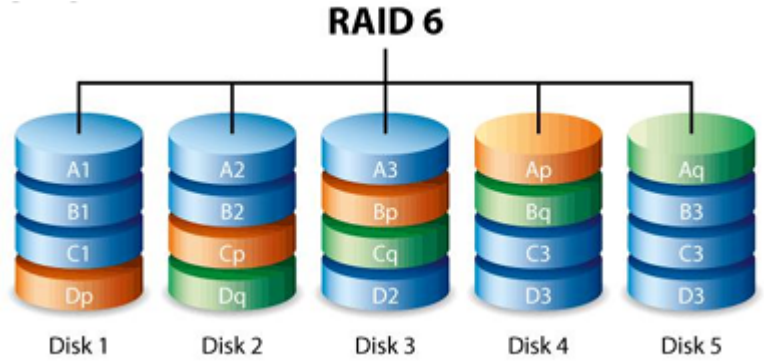
RAID-4:
Striping a livello di blocchi.
Si aggiunge un disco con codice a correzione di errore per ricostruire i bytes persi. Questo costituisce un collo di bottiglia per il sistema poiché l'informazione di parità deve essere aggiornata dopo ciascuna operazione di scrittura.
Si può aggiungere un disco ogni volta che ce n'è necessità.



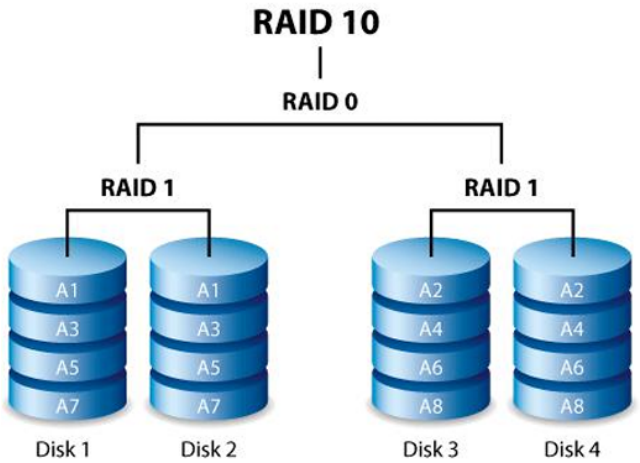
RAID-5:
Usato dai data center ad elevata disponibilità.
E' una soluzione costosa in termini di numero di dischi.
Non solo un disco ha informazioni di parità.
In questo modo non c'è bisogno di aggiornare le informazioni di parità solo su un disco, ma sono suddivise tra più dischi. Il throughput aumenta, le scritture non soffrono.
La tolleranza è di un solo disco guasto.



RAID-6:
C'è un'altra informazione di parità suddivisa tra i dischi.
Tolleranza: fallimento di 2 dischi.
Ci vuole un gran numero di dischi.



RAID-10:
Unisce RAID-0 e RAID-1, ovvero i dischi del RAID-0 sono array di tipo RAID-1.



08/01/2024 (lez.36)
Backup:
Per aumentare l'affidabilità di un sistema c'è bisogno di eseguire dei backup.
Anche avere più dischi può portare a perdita di dati.

I backup vengono visti come un costo aggiuntivo, ma solo fondamentali ogni volta che dei dati vengono distrutti, si pensi ad esempio ai ransomware, se si avessero backups dei dati sarebbero totalmente inefficaci.
Avere a che fare con i backup significa capire quanto spesso vanno fatti e dove mantenere i dati.
La copia va gestita appropriatamente perché mantiene dati significativi.
E' importante tener conto anche della sicurezza interna, ovvero se qualcuno dall'interno riesce ad accedere a tutti i dati, questo costituisce un problema.
Le grandi aziende hanno delle facilities dedicate in cui mantenere i dati, anche interi edifici dedicati, per accedere ai quali è richiesti autenticazione a più fattori.
Non ci sono strade facili, bisogna avere backups e mantenerli in maniera appropriata.

Disaster recovery:
Un concetto aggiuntivo è quello di "disaster recovery". Se si perdono tutti i dati essenzialmente perdiamo tutto, se abbiamo il backup nello stesso edificio del server e l'edificio va a fuoco perdiamo tutto.
Bisogna pianificare appropriatamente il "disaster recovery"; bisogna tenere in conto due aspetti: ammontare di tempo per rendere il sistema nuovamente operativo e ammontare di dati che si può accettare di perdere quando accade un disastro.
Bisogna trovare un trade off tra questi due aspetti.
Questo è il motivo per cui i cloud providers dividono in zone le VMs.
Molteplici copie di VMs sono replicate in zone differenti, in edifici diversi. Tutti i dati periodicamente vengono copiati in una località remota.
I dati sono salvati in nastri magnetici: molto spazio, poco costo.
Se le regioni di backup vanno giù bisogna reinstallare tutto da questi dischi.
RPO→ Quanti dati posso permettermi di perdere?
RTO→ quanto a lungo il mio sistema può restare offline se il backup è fatto su nastri magnetici? La tecnologia che si usa ha a che fare con il costo che si paga.
Il trade off tra costo e benefici è fondamentale quando si pianifica disaster recovery.
SLA ci dice quanto un sistema è "dependable" da un punto di vista di sicurezza.

Encryption:
Ad un certo punto il nostro sistema deve avere a che fare con i dati: i clienti devono poter fare operazioni sui dati.
Non ci si può fidare di terze parti. Se condivido informazioni con i clienti, ma non voglio che leggano il contenuto dei dati, uso la cifratura: i client possono eseguire operazioni sui dati senza leggere tutti i dati.
Le terze parti non devono avere la possibilità di decifrare.

Homomorphic encryption:
Entra in gioco la cifratura omomorfa.
Il client fa operazioni sui dati cifrati senza decifrarli.
Non c'è bisogno di decifrare per calcolare il risultato dell'operazione.
Algoritmo aggiuntivo: valutazione omomorfa.
Abbiamo la funzione f che è un'implementazione matematica di una query.
Il risultato è il CT.

Cifratura omomorfica corretta:

DECsk(ENCpk(f,c1,...ct))=f(m1,...,mt)
Utile per ottenere insiemi, infatti possiamo scoprire se qualcosa è in un insieme o meno senza scoprire l'intero insieme.
Però questa condizione non ci dice che possiamo valutare qualcosa, perchè se f è la funzione identità non possiamo calcolare nulla. Quindi bisogna aggiungere qualche bit di informazione.
Qualsiasi schema di cifratura omomorfa:
η=lunghezza in bit della SK
γ=lunghezza in bit di PK
ρ=lunghezza in bit del "rumore"

Ci sono limitazioni sul rumore che si può accumulare.
SK=intero pari con η bit
PK=p*q con q$\in$[0,$2^{\gamma}/p$]

Gli algoritmi di cifratura si basano su alcune equazioni lineari: ogni volta che si cifra qualcosa si aggiunge del rumore.
Questo schema è omomorfo rispetto ad addizione e moltiplicazione:
$c1 + c2 = m1 + m1 + p * (q1 + q2) + 2 * (r1 + r2)$
Ci ottiene il CT dei due PT ma con il doppio per rumore.
Se si somma più volte il rumore diventa troppo grande.
Lo stesso succede con la moltiplicazione.

Leveled Fully Homomorphic Cryptography
Facciamo qualcosa di meglio: una serie di ENC e DEC in modo che andando da un livello all'altro il rumore si elimini.
Quando si esegue un calcolo si sta decifrando e cifrando qualcosa in modo da rimuovere e aggiungere rumore.
Sequenza di livelli, ad ogni livello si decifra e ricifra il CT.
C'è bisogno di una serie di chiavi PK e SK.

PSI
E' un'applicazione rilevante.
Abbiamo più parti che vogliono conoscere l'intersezione tra insiemi ma non vogliono che gli altri conoscano il proprio insieme.
Le diverse parti devono accordarsi su uno schema crittografico. gli elementi non presenti nell'intersezione sono mappati a valori random.
Prendiamo l'esempio con due partecipanti.
Abbiamo un sender ed un receiver: il sender ha un dataset S di taglia Ns, il receiver ha S' di taglia Ns' e supponiamo che siano fatti di stringhe di bit di taglia fissa, σ; i valori NS,NS0 e σ sono pubblici.
L'algoritmo è diviso in fasi:

- **setup**: mittente e destinatario concordano su uno schema crittografico completamente omomorfico; il destinatario genera una coppia di chiavi, mantenendo segreta quella privata.
- **encryption**: il ricevente cripta tutti gli elementi x appartenenti a X e invia i ciphertexts criptati (c1, ···, cn) al mittente

- **calcolo dell'intersezione**: per ogni c_i il mittente genera un valore random r_i e calcola omomorficamente:

$$di = ri * \Pi s'(ci - s')$$

  Poi i cyphertext (d1,...,$d_{s'}$) sono inviati al ricevitore
- **estrazione della risposta**: il ricevitore decifra (d1,...,$d_{s'}$) e calcola:

$$S \cap S' = \{sk : decrypt(Sk, di) = 0\}$$

Il ciphertext ricevuto è sottratto omomorficamente 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 del rumore 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
Non si può risalire a nulla riguardo l'altra parte.