Protezione e Integrità degli Accessi

Parte VI

Indice

Garantire la privacy dei dati nel cloud significa garantire anche la confidenzialità dell'**accesso ai dati**:

- ***Access confidentiality:*** garantire confidenziale il fatto che un accesso mira a un determinato dato
- ***Pattern confidentiality:*** confidenzialità del fatto che due accessi mirano allo stesso dato; si vuole proteggere il fatto che un utente abbia fatto la stessa query in due momenti diversi, o che due utenti abbiano fatto la stessa query

Capitolo 1

Path ORAM e Ring ORAM

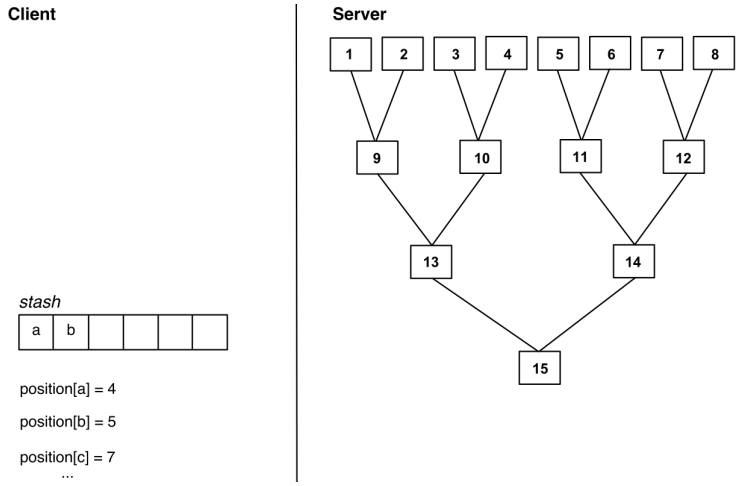
1.1 Path ORAM

- **Server side**
  - Si organizzano i dati in un albero binario
  - Ogni nodo contiene è un *bucket* che contiene sia dei dati che dello spazio disponibile
  - Ogni nodo foglia x rappresenta un cammino unico $P(x)$ da x alla *root*
- **Client side**
  - Si tiene una piccola cache chiamata *stash*
  - Tiene una *position map:* $x = position[a]$
    $\rightarrow$ il blocco a è in qualche *bucket* nel path $P(x)$ per raggiungere x
  - Ogni volta che viene letto qualcosa i blocchi vengono spostati

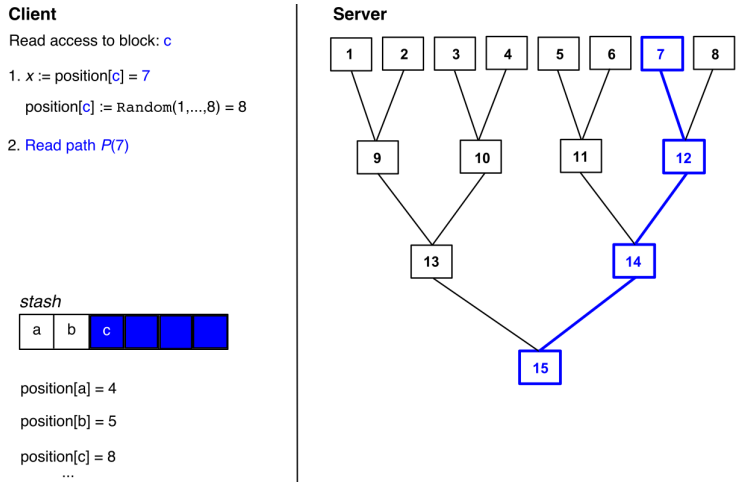
In ogni momento (*main invariants*):

- ciascun blocco è mappato ad un cammino radice-foglia
- quando devo riposizionare un blocco dopo averlo letto, non lo faccio subito (altrimenti vedi dove l'ho messo); vengono tenuti nella memoria locale e poi vengono riallocati durante una nuova lettura

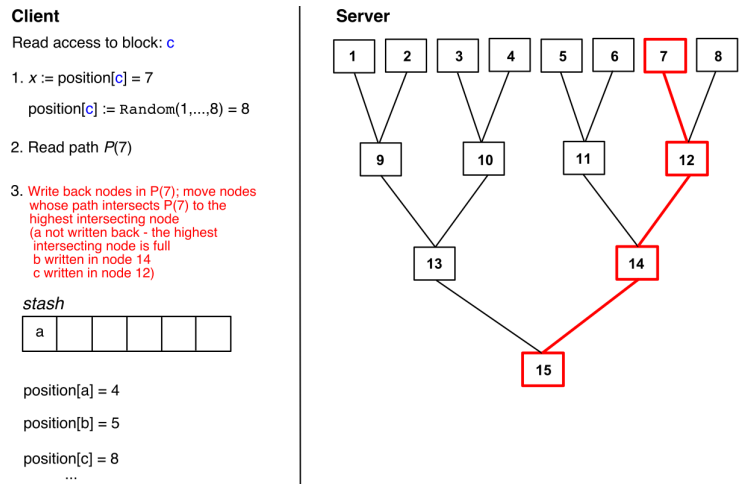
Esempio



- in *stash* abbiamo gli elementi che ancora dobbiamo mettere a posto (a e b)
- sotto abbiamo la *position map*:
  - dobbiamo mettere a in un cammino verso 4
  - dobbiamo mettere b in un cammino verso 5
  - sappiamo che c si trova in un cammino verso 7



- Leggiamo c; dato che sappiamo che dovremo cambiare posto, calcoliamo anche la sua nuova posizione
- aggiorniamo la mappa



- intanto che percorriamo il cammino $P(7)$ per leggere c, ci portiamo tutto quello che possiamo riposizionare dallo *stash*

  $\rightarrow$ io faccio sempre tutto il cammino fino alla foglia, anche perché se mi fermassi prima ti starei spifferando dove ho trovato il blocco
- b viene scritto nel nodo 14; c viene scritto nel nodo 12

1.2 Ring ORAM

È una variante del Path ORAM; ha la stessa struttura server-side del Path ORAM con qualche piccola modifica per la gestione.
La differenza principale sta nel fatto che non si va a rimappare e riscrevere le cose ad ogni accesso, ma viene fatto *ogni tanto*.

1.3 Vantaggi e Svantaggi

- + performance migliorate
- - query di range non supportate
- - accessi da client multipli non supportati
- - vulnerabile a *failures* del client

Capitolo 2

Shuffle Index

2.1 Struttura Dati

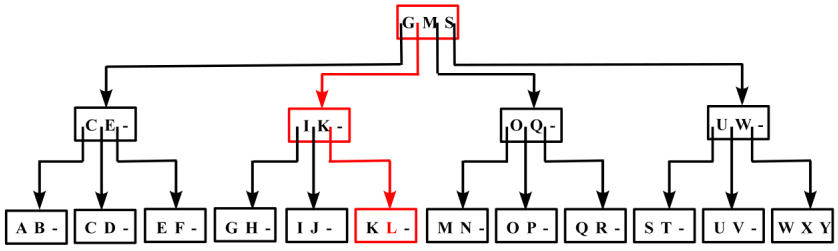
Si utilizza un B_+-tree (usati per rispondere anche a query di range):

- Segue il ragionamento dell'albero di ricerca
- Ogni nodo contiene delle chiavi; deve essere pieno almeno per metà
- Tra le chiavi ci sono dei cammini che vanno ai nodi figli; ogni cammino mi dice il percorso da seguire per arrivare al valore che mi interessa

  $\rightarrow$ se ho le chiavi $2, 5, 9$:

  - per un valore $x < 2$ prendo il cammino a sinistra
  - per un valore $2 \leq x < 5$ prendo il cammino tra 2 e 5
  - e così via ...
- i valori stanno solo nelle foglie; le foglie sono tutte allo stesso livello

Rappresentazione Astratta



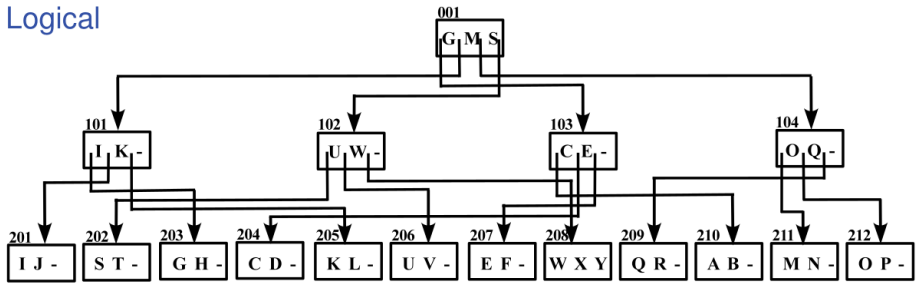
Search: L

Rappresentazione Logica

I puntatori tra nodi corrispondono, a livello logico, a degli identificatori.
Avrò un insieme di coppie $\langle id, n \rangle$ dove:

- id identificatore del nodo
- n contenuto del nodo

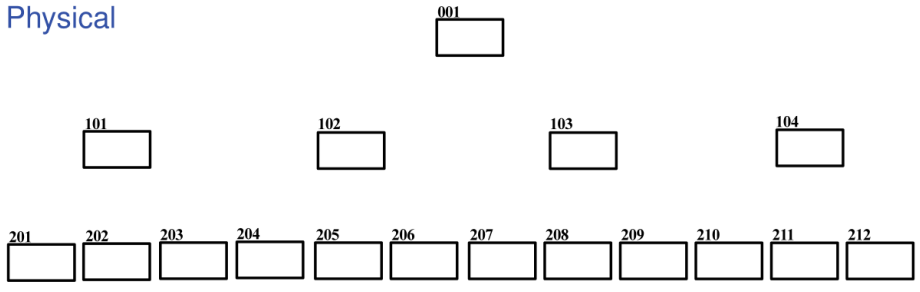
$\rightarrow$ l'ordine degli id non dovrebbe dirmi nulla sull'ordine dei nodi nella rappresentazione astratta



Rappresentazione Fisica

A livello fisico avrò dei blocchi criptati; ciascuna coppia $\langle id, n \rangle$ corrisponde a una coppia $\langle id, b \rangle$, dove:

- id identificatore
- b è il blocco criptato, costruito come la concatenazione tra:
  - cifratura di n con *salt* (il sale serve perché altrimenti anche cambiando posto alle cose, se la criptazione è sempre la stessa tu sei in grado di riconoscerlo)
  - id del blocco

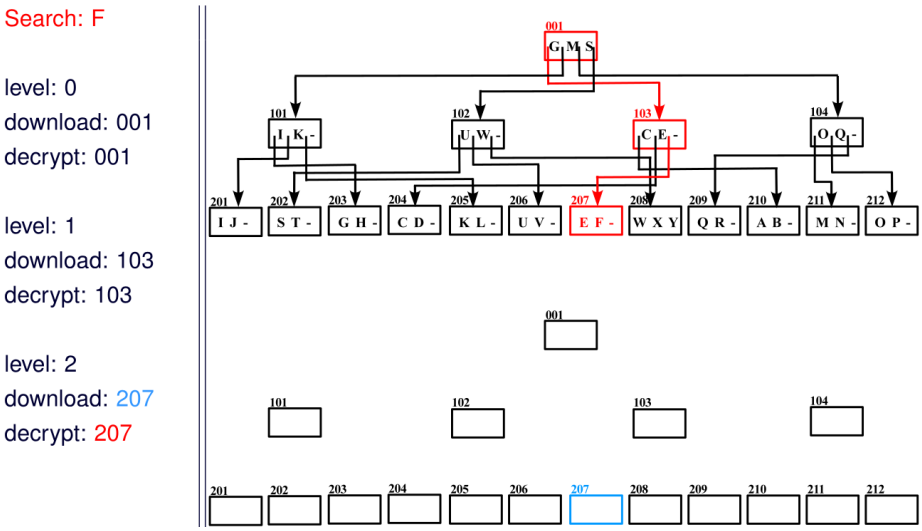


2.2 Accesso ai dati

L'accesso ai dati richiede un processo iterativo tra il client e il server; ad ogni iterazione, il client:

- decripta il blocco appena recuperato
- determina il blocco che deve recuperare il server al livello inferiore (verso le foglie)

→ il processo termina quando si raggiunge una foglia



2.3 Conoscenza del server

Il server ottiene una sequenza di blocchi che compongono delle osservazioni:

- $o_i = \{b_{i1}, b_{i2}, \ldots, b_{ih}\}$

Il server può inferire facilmente:

- il numero m di blocchi e i loro identificatori
- il livello associato ad ogni blocco
- altezza dell'albero

La cifratura basta?

- **+ protegge**
  - contenuto dei dati
  - contenuto del singolo accesso
- **- non protegge**
  - *access confidentiality*
  - *pattern confidentiality*
    → quando cerco un dato lo trovo sempre nella stessa posizione

→ **attacchi basati su frequenze possono permettere al server di ricostruire i valori in chiaro**

2.4 Tecniche di protezione

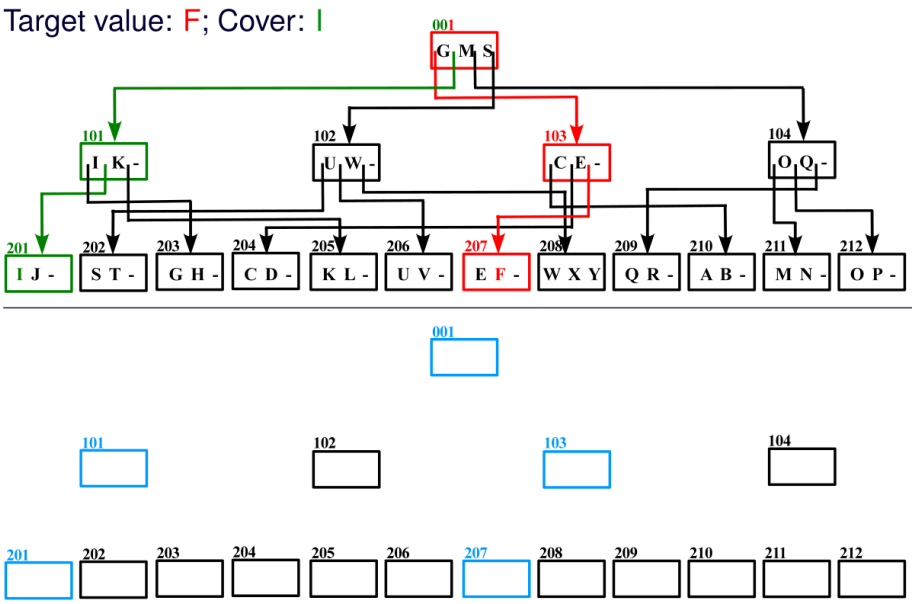
Per proteggermi l'obiettivo è **rompere la corrispondenza tra contenuto e luogo in cui è memorizzato**
Si combinano 3 strategie.

2.4.1 *Cover searches*

- Introduce confusione sul target di un accesso, nascondendolo tra un gruppo di altre richieste che fanno da *cover*
- Il numero di richieste di cover *num_cover* è il parametro di protezione
- Le *cover searches* devono:
  - fornire **diversità** tra blocchi
  - essere **indistinguibili** dalla ricerca reale

Protezione offerta

- + le foglie hanno la stessa probabilità di avere il vero target
- + le relazioni tra nodi padre-figlio vengono confuse (*il blocco 201 potrebbe essere figlio di 101 o 103*)
- - le relazioni padre-figlio possono essere inferite con attacchi di intersezione



2.4.2 *Cached searches*

Il client tiene una cache di nodi:

- di dimensione num_cache, che stabilisce la dimensione per ciascun livello
- se un nodo è presente, allora è presente anche il suo genitore
- ogni volta che leggo qualcosa lo metto nella cache
- gestisco la cache con politica *Last Recent Used*
- se il target è nella cache, vengono fatte solo delle *cover searches*

Protezione offerta

- \+ protegge da attacchi di intersezione
- \- protegge solo finché sto nella dimensione della cache

2.4.3 *Shuffling*

- Rompe la corrispondenza tra blocchi e nodi, mescolando ogni volta il contenuto in maniera casuale
- Ogni volta devo decriptare e re-criptare per poter mescolare

2.5 Analisi della protezione

- con lo *shuffling* rompo la corrispondenza nodo-blocco, riuscendomi a proteggere da attacchi di intersezione
- ***Access confidentiality:*** ogni accesso deve essere diviso tra $num\_cover + 1$ richieste; in più lo *shuffling* rompe la corrispondenza nodo-blocco
- ***Pattern confidentiality:*** ci si riesce a proteggere con:
  - breve termine con cover e cache
  - lungo termine con cover e shuffling

2.6 Protezione vs Performance

- - un accesso di lettura implica $num\_cover + num\_cache + 1$ scritture al server
- - gli accessi concorrenti diventano accessi esclusivi
- \+ migliori performance di Path ORAM
- \+ non ci sono soluzioni che offrono la stessa protezione a performance migliori