**QUERY**

Fin’ora abbiamo visto la confidentiality per i dati locati esternamente quindi content confidentiality, in questo capitolo ci importa mantenere una certa confidenzialità anche a livello di query quindi a livello di accesso dei dati, access confidentiality.

Non voglio far capire al server che un certo utente sta cercando una certa entri specifica di una enciclopedia ad esempio. Non voglio che il server conosca i dati a cui accedo.

Access confidentiality = un accesso punta a dati specifici.

Pattern confidentiality: due richieste differenti mirano allo stesso dato.

**Path ORAM**

È in sostanza una ricerca per chiave.

Il server è un albero binario, NON di ricerca, possiede un’altezza L = log(N) -1 dove N è il numero dei blocchi, ogni blocco è un contenitore che possiede Z posti, in questi Z posti ci possono essere blocchi veri o del padding.

Una foglia x definisce in modo univoco il path P(x) da essa alla root.

Il client ha uno stash nel quale sono contenute le posizioni mappate di alcuni blocchi, sono elementi che devo ancora mettere a posto nell’albero.

x = position[a] means that a block identified by a is currently mapped to the x-th leaf node.

Il blocco a si trova da qualche parte tra la foglia x e il path alla root oppure nello stash(devo ancora metterlo a posto).

L’idea è garantire access control cambiando posizione degli elementi che leggo ogni volta che accedo.

La funzione di mapping del client mi permette di andare a piazzare gli elementi che non ho ancora messo a posto e che sono nello stash, quindi quando mi leggo un elemento dal server lo rimappo tramite questa funzione per sapere dove andare a metterlo e poi lo tengo nello stash (è come se decidessi random una delle foglie e poi saprò che dovrò mettere l’elemento in uno dei path relativi alle foglie dell’albero).

Il cammino su cui leggo, lo riscrivo anche, andrò a mettere a posto tutto quello che andava messo a posto nello stash, tutti quelli che qualcuno cercherà in quel cammino lì, in base alla map function sui nodi che percorro. Rimetto a posto lungo i nodi che percorro quando leggo e se c’è spazio.

Mi conviene mettere gli elementi sempre più verso le foglie perché è dove c’è meno incidenza, e quindi più sicurezza nell’accesso.

L’idea è mettere a posto quando leggo così il server non capisce dove ho rimesso le cose lette. Quindi il path lo faccio solo 1 volta. Ogni volta che leggo vado fino in fondo alla foglia, per confidentiality e in più posso rimettere a posto più cose.

PER ORA tutti passano dallo stesso trusted client e quindi non modello la concorrenza e il multiutente.

Conclusione: a quanto capito leggo e rimappo subito, non mi interessa di andare li fisicamente a prendere il dato, ce l’ho appena leggo e lo rimetto nell’albero. A meno che parte dalla foglia alla root.

**Ring ORAM**

Variante del path oram, spostare i blocchi non lo fa a tutti gli accessi ma fa la mescolazione delle carte in modo periodico.

ORAM e basta = oblivious random access memory, cioè completamente random.

Ring e path sono entrambi migliori rispetto all’oram.

**Shuffle index data structure:**

Utilizza la struttura ad albero B+.

Foglie in basso + figli indexing ecc...

L'albero è B+ ma è unchained cioè non ha i puntatori tra foglie, viene fatto per non mantenere l’ordine dei valori nelle foglie altrimenti non sarebbe così sicuro. Se no saprei l’ordine di grandezza

Fan out = numero di puntatori nei nodi.

La visione logica che vedo io, quindi B+ tree è diversa da ciò che memorizzo fuori, cioè ciò che vede il server. L’ordine logico può non essere rispettato dall’ordine fisico.

Ogni volta che cripto uso un sale diverso per randomizzare i risultati. I nodi sono criptati in questo modo, così se l’encryption del nodo cambia o non cambia questo non lo saprà mai il server.

Il server non sa se il contenuto è cambiato oppure no, non può accorgersene.

Il server però non mi può più indirizzare al contenuto che mi serve, perché esso non conosce più nulla di quello che sto cercando, questo ragionamento lo devo fare io client.

Quindi dico al server che nodi darmi da portare a casa in modo da decriptarli e utilizzarli.

L’accesso quindi è un processo iterativo, ad ogni iterazione ho un livello di decriptazione in più.

Il server conosce la radice e in generale i livelli dell’albero, sa osservare a che livello appartengono i diversi nodi.

Il server non dovrebbe sapere il contenuto di ciò che mi serve: content conf., non dovrebbe sapere a che dati è indirizzata la mia richiesta: access conf., e non dovrebbe sapere che due richieste puntano allo stesso dato quindi pattern conf.

L’encryption da sola non mi garantisce l’access confidentiality perché i meccanismi dell’albero possono essere ricostruiti grazie allo studio delle frequenze delle queries.

Una sola query non mi dice niente ma un’insieme di query mi dicono tutt’altro.

Ancora una volta l’idea è fare in modo che la corrispondenza tra ciò che osservo e ciò che vado a prendere con le query deve essere rotta per ottenere una access confidentiality.

Per romperla devo avere dinamicità.

Per farlo lo shuffle index utilizza 3 tecniche insieme:

1. Cover search: non cerco solo 1 cosa
2. Cached searches: cache è diverso da quello che utilizza path oram, la cache è esattamente una copia non indica ciò che devo rimettere ancora a posto.
3. Shuffling: ad ogni accesso cambio la locazione dei blocchi.

Cover searchers:

Nascondo la richiesta insieme con altre, non cerco solo 1 cosa ma ne cerco di più, così chi mi osserva è confuso. Più coperture uso e più sono protetto, però più ho overhead.

Le cover devono confondersi con gli accessi reali, questo è il problema principale delle covers, devono essere fatte bene. Però la relazione padre-figlio può essere esposta ad attacchi di intersezione perché se ad esempio faccio due volte la stessa richiesta e ottengo 3 path, 1 è lo stesso di prima e gli altri due cambiano, quindi diventa ovvio chi è il padre e chi il figlio della prima richiesta. (cammino comune che più ricerche possono avere).

Le covers non bastano, aggiungo cache.

Cache:

La cache è qualcosa che ho acceduto da poco e che ho salvato in cache con polcy LRU (least recently used va fuori).

Se un nodo è nella cache allora anche suo padre è nella cache non val il viceversa.

Se non faccio la richiesta al server perché ho già ciò che cerco in cache in modo implicito sto dicendo che quella cosa ce l’ho già e che ci ho acceduto di recente, però se invece faccio la richiesta posso fare due path che non sono più in comune e il server non può sapere quello che gli chiedo dove sta, non può inferire la relazione padre e figlio perché i paths che farò per la stessa richiesta saranno tutte e due le volte differenti. Se devo cercare F, faccio richieste farlocche perché F ce l’ho già in cache e svio il ragionamento del server.

La cache comunque non protegge ad attacchi di intersezione che vanno oltre la dimensione della cache es: cache dim 2 mi protegge da FAF ma non da FABF F sarà di nuovo esposto.

Perciò utilizzo la terza tecnologia:

Shuffling:

Tutte le volte che accedi qualcosa cambiagli di posto (cuore della protezione)

Ogni volta cripto con sale diverso altrimenti i blocchi sono riconoscibili, ad ogni livello, cambio l’indexing e quindi scambio di posizione i blocchi.

L’unico problema è che quando mescolo devo dirlo ai padri, devo cambiare i puntatori in modo che i padri sappiano a che figli puntano. In sostanza guardo come cambiano i figli i+1 intanto che sono al livello i così cambio subito i puntatori.

Quello che mi costa è che scrivo i blocchi in base a num cover e num cache, per fuorviare il server, anche se in realtà mi serve solo 1 blocca da leggere e da portare a casa.

La confidenzialità in accesso me la da la cover search + cache, mentre la confidenzialità del pattern me la garantisce l’unione con lo shuffling.

Costo = num\_cov + num\_cache + writeback server, costa un sacco ma non si può fare di meglio per garantire questa protezione.

Lo shuffle index + più bello rispetto a path oram perché se perdo la cache non perdo molto, perdo magari garanzia nel breve termine ma si rifà, mentre nel path oram perdo i dati. Questo è molto importante perché mi impatta la corretteza della struttura in path e ring oram sarei fritto, in shuffle indexing no.

Se dovessi farlo in concorrenza comunque non potrei, dovrebbero diventare tutti acessi sequenziali.

Il server posso giocarmelo anche tra più server, non tenendone solo 1.(con lo shuffling)

Inoltre può essere esteso ad avere più accessi.