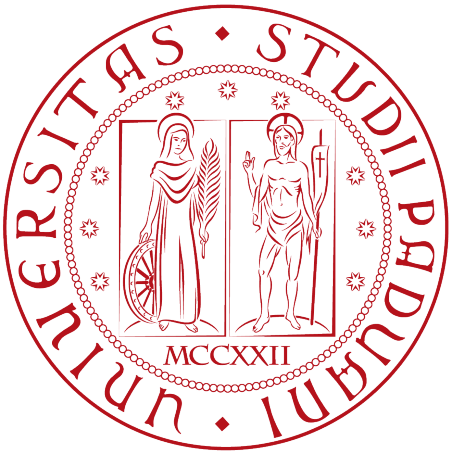
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Crittografia applicata per la protezione dei dati at-rest

Tesi di laurea

Relatore

Prof.Davide Bresolin

Correlatore

Dott.Mattia Zago

Laureando

Leonardo Speranzon

Sommario

La sicurezza delle moderne infrastrutture aziendali rende necessario lo sviluppo di logiche di protezione che possano coprire tutto il perimetro digitale, particolarmente nel contesto di servizi ed infrastrutture as-a-service.
Uno dei vettori principali d'attacco risulta essere l'outsourcing di componenti di sviluppo a fornitori e terze parti in genere, dove i flussi dei dati sono di difficile mappatura ed i controlli d'accesso bypassati, favorendo un accesso indiscriminato ai dati sorgente.
Questa tesi si pone l'obbiettivo di analizzare la protezione dei dati sorgente quando questi non sono attivamente utilizzati dalle applicazioni che devono legittimamente utilizzarli, ovvero lo studio delle soluzioni crittografiche di cifratura dei dati at-rest. Attraverso un'analisi qualitativa dello stato dell'arte in termini di applicazioni crittografiche innovative come la crittografia omomorfica e crittografia simmetrica ricercabile (dall'inglese Symetric Searchable Encryption, SSE) verrà costruito uno studio di fattibilità ed implementato un prototipo di middleware crittografico.

Indice

1 Introduzione **1**
- 1.1 L'azienda . . . 1
  - 1.1.1 La startup . . . 1
- 1.2 Lo stage . . . 2

2 Descrizione dello stage **3**
- 2.1 Organizzazione dello stage . . . 3
- 2.2 Tecnologie utilizzate . . . 4
  - 2.2.1 MongoDB . . . 4
  - 2.2.2 Typescript . . . 4
  - 2.2.3 Libreria crypto-js . . . 4

3 Nuove tecnologie crittografiche **5**
- 3.1 Homomorphic Encryption . . . 5
- 3.2 Searchable Encryption . . . 6
- 3.3 Order-preserving Encryption e Order-revealing Encryption . . . 7
- 3.4 Attribute Based Encryption . . . 8
- 3.5 Proxy re-encryption . . . 8

4 Scelta delle tecnologie **11**
- 4.1 Valutazioni dei diversi tipi di tecnologie crittografiche . . . 11
- 4.2 Valutazioni dei diversi tipi di Searchable Encryption . . . 12

5 Analisi e metodologia della soluzione **15**
- 5.1 Modelli analizzati . . . 16
  - 5.1.1 Aggiunta di un campo contenente l'hash . . . 16
  - 5.1.2 Indice invertito criptato . . . 18
  - 5.1.3 Indice invertito criptato e concatenato . . . 20
  - 5.1.4 Schema da Zheng et al. (2020) . . . 22
- 5.2 Comparazioni tra i vari modelli . . . 27
  - 5.2.1 Caratteristiche dei modelli . . . 27
  - 5.2.2 Complessità . . . 27
  - 5.2.3 Tempi risultanti dal PoC . . . 27

6 Architettura del PoC **29**
- 6.1 Obiettivo del PoC . . . 29
- 6.2 Decisioni generali . . . 29
- 6.3 FieldIndex . . . 31
- 6.4 Repository . . . 32

6.5 Aspetti migliorabili 33
6.5.1 Accoppiamento eccessivo tra FieldIndex e la libreria MongoDB 33
6.5.2 Utilizzo di un Builder per la creazione delle MongoMovieRepository 33

7 Conclusioni **35**
7.1 Risultati raggiunti 35
7.2 Conoscenze acquisite 35
7.3 Valutazione personale 35

Acronimi e abbreviazioni **37**

Glossario **39**

Elenco delle figure

1.1 Logo Athesys 1
1.2 Logo Monokee 1

4.1 Metodi per ottenere una crittografia limitata a dei gruppi 11
4.2 Rappresentazione semplificata dell'applicazione 12

5.1 Rappresentazione in forma tabellare del modello dell'indice formato dal campo hash 16
5.2 Diagramma di sequenza della ricerca con l'indice formato dal campo hash 17
5.3 Rappresentazione in forma tabellare del modello indice invertito criptato 18
5.4 Diagramma di sequenza della ricerca con l'indice invertito criptato . . 19
5.5 Rappresentazione in forma tabellare del modello indice invertito criptato e concatenato 20
5.6 Diagramma di sequenza della ricerca con l'indice formato dal campo hash e concatenato 21
5.7 Rappresentazione in forma tabellare del modello di Zheng et al. 22
5.8 Diagramma di sequenza della ricerca utilizzando il modello di Zheng et al. 23

6.1 Diagramma delle classi del PoC 30
6.2 Dettaglio dello schema delle classi relativo a FieldIndex 31
6.3 Pattern Repository 32
6.4 Dettaglio dello schema delle classi relativo a MovieRepository 32

Elenco delle tabelle

5.1 Complessità del modello dell'indice formato dal campo hash 17

5.2 Complessità del modello dell'indice formato dal campo hash su campi singoli 18
5.3 Complessità del modello dell'indice invertito criptato 19
5.4 Complessità del modello dell'indice invertito criptato e concatenato . . 22
5.5 Complessità del modello proposto da Zheng et al. 24
5.6 Caratteristiche dei diversi schemi 27
5.7 Complessità dei diversi schemi 27
5.8 Tempi ottenuti dal PoC 28

Capitolo 1

Introduzione

In questo capitolo descrivo l'azienda in cui ho svolto il tirocinio e lo scopo del tirocinio stesso.

1.1 L'azienda

Figura 1.1: Logo Athesys

Athesys nasce nel 2010 dalla sinergia di affermati professionisti del settore IT, con lo scopo di offrire consulenza altamente specializzata in ambito System Integration, Database Management, Sicurezza applicativa, Governance Cloud Platform, Hyperconvergenza e Sviluppo Software in modalità Agile.

1.1.1 La startup

Figura 1.2: Logo Monokee

Durante il periodo del mio tirocinio ho lavorato per l'espansione del prodotto di Monokee, la startup di Athesys.

Monokee è una startup nata nel 2017 per la creazione di una soluzione IAM (Identity Acess Management) basata sul cloud. Nel 2019 con l'avvento della SSI (Self-Sovereign Identity) ha deciso di sviluppare la prima soluzione hybrid-SSI ovvero un sistema di *Access Management* sia centralizzato che decentralizzato. Attualmente Monokee è impegnata in molteplici progetti sia per quanto riguarda la tecnologia SSI,

in collaborazione anche con atenei come quello di Napoli o la Cattolica di Milano, che nell'ambito blockchain in generale.

1.2 Lo stage

I crescenti attacchi alle infrastrutture aziendali hanno dimostrato come le logiche di protezione applicative siano insufficienti e non possano coprire tutto il perimetro digitale, particolarmente nel contesto di servizi ed infrastrutture as-a-service. Uno dei vettori principali d'attacco risulta essere l'outsourcing di componenti di sviluppo a fornitori e terze parti in genere, i flussi dei dati sono di difficile mappatura ed i controlli d'accesso bypassati, favorendo un accesso indiscriminato ai dati sorgente.

Dato che lo sviluppo del prodotto Monokee da parte della startup non è ancora terminato ho avuto la possibilità di proporre di aggiungere allo stack di sicurezza dei dati delle nuove feature riguardo la sicurezza dei dati salvati nel database. Per questo lo scopo dello stage è stata la creazione di un PoC di un middleware crittografico che permetta di implementare la nuova soluzione all'interno di Monokee in modo da migliorare la sicurezza dei dati at-rest.

Capitolo 2

Descrizione dello stage

2.1 Organizzazione dello stage

Lo stage è stato suddiviso in 4 periodi di 2 settimane ciascuno.

Primo periodo (09/05/2022 - 20/05/2022) Ho ricercato e studiato, attraverso paper ed estratti di conferenze, quali fossero le tecnologie crittografiche allo stato dell'arte in modo da poterne discutere con il mio tutor aziendale e decidere quale fra queste sarebbe stata la tecnologia crittografica più adatta per lo sviluppo del tirocinio.

Secondo periodo (23/05/2022 - 03/06/2022) Ho studiato approfonditamente le diverse ricerche rispetto alla tecnologia scelta, la *Searchable Encryption*, per capire il funzionamento delle diverse soluzioni e le definizioni di sicurezza ad essa collegate.

Al termine di questo secondo periodo insieme al tutor aziendale abbiamo valutato come questa tecnologia potesse essere utilizzabile all'interno del prodotto, il tutor mi ha spiegato come sarebbe dovuta avvenire la progettazione del PoC e abbiamo concordato le possibili soluzioni in modo da poterle compararle.

Terzo periodo (06/06/2022 - 17/06/2022) Ho prodotto la documentazione di queste soluzioni, descrivendo gli schemi di funzionamento, le valutazioni sulla sicurezza e le performance teoriche.

Tale documentazione è stata verificato con il tutor aziendale prima di iniziare con la progettazione e codifica del PoC.

Quarto periodo (20/06/2022 - 01/07/2022) Ho scritto il PoC che implementava le soluzioni e le testava comparandone l'efficienza in termini temporali e di spazio aggiuntivo occupato all'interno del database. In questo modo è possibile combinare le informazioni relative alla sicurezza di ogni soluzione con il loro costo in modo da poter scegliere la più adatta ad ogni esigenza.

Al termine del tirocinio ho avuto un meeting col gruppo completo di sviluppo per discutere la possibile implementazione di questa tecnologia nello stack finale.

2.2 Tecnologie utilizzate

Per facilitare una futura integrazione del PoC con il prodotto Monokee ho deciso di utilizzare le stesse tecnologie, ovvero *Typescript* come linguaggio di programmazione e *MongoDB* come database.

2.2.1 MongoDB

MongoDB è un DBMS non relazionale orientato ai documenti, questo gli permette una maggiore velocità rispetto ai tradizionali DBMS relazionali. Essendo un DBMS noSQL non segue la sintassi standard SQL ma ne ha una propria, composta da operatori a cascata. Di seguito una query che stampa tutti i nominativi dalla tabella studenti scritta in SQL (A) e nel linguaggio di MongoDB (B):

```
(A) SELECT nome, cognome FROM Studenti WHERE sesso='M'
(B) db.collection(Studenti).find({sesso:'m'},{nome:1, cognome:1})
```

2.2.2 Typescript

Il linguaggio *Typescript* è costruito su Javascript e ne estende la sintassi per ottenere un linguaggio fortemente tipizzato, rendendo possibile la scrittura di un codice più chiaro e comprensibile, caratteristica molto utile in scenari di collaborazione. Inoltre, data la necessità della compilazione dei file Typescript in Javascript viene anche eseguito un attento *type-checking* in modo da ridurre gli errori durante l'esecuzione del codice.

2.2.3 Libreria crypto-js

Come consigliato dal mio tutor ho utilizzato questa specifico modulo node per utilizzare le primitive crittografiche all'interno di *Typescript*. È importante utilizzare una libreria apposita per queste funzioni perchè quelle implementate direttamente dai linguaggi di programmazione non sono create a scopo crittografico e possono quindi risultare vulnerabili a possibili attacchi.

Capitolo 3

Nuove tecnologie crittografiche

3.1 Homomorphic Encryption

Con il termine Homomorphic Encryption (HE) si intende la tecnologia crittografica tramite cui sia possibile eseguire operazioni, solitamente aritmetiche, su uno o più ciphertext senza la necessità di doverli decifrare.

Più in particolare si dice Fully Homomorphic Encryption (FHE) un modello che permetta sia operazioni di addizione che moltiplicazione.

Dato che i modelli di HE non sempre riescono a raggiungere i requisiti della FHE vengono suddivisi in:

- *Partially homomorphic encryption*: i modelli che permettono solo un tipo di operazione;
- *Somewhat homomorphic encryption*: i modelli che permettono sia operazioni di addizione che di moltiplicazione ma solo su un dominio ristretto;
- *Leveled fully homomorphic encryption*: i modelli che permettono sia operazioni di addizione che di moltiplicazione ma hanno un limite predeterminato sul numero di operazioni utilizzabili;
- *Fully homomorphic encryption*: i modelli completamente omomorfo come da definizione di Rivest [1].

Storia Il problema di uno schema crittografico omomorfo è stato presentato per la prima volta nel 1978 [1] in cui viene descritta la FHE.

Questo problema è però rimasto insoluto fino a quando Gentry [2] nel 2009 ne ideo il primo modello.

Fino ad allora vi furono diverse proposte ma nessuna di queste soddisfaceva completamente la definizione di FHE, proprio per questo motivo vennero create diverse classificazioni intermedie.

Problemi attuali Come descritto da Peng [3] tutti i modelli omomorfi finora proposti presentano problemi relativi alla sicurezza, questi problemi sono ben conosciuti a livello teorico ma non è molto trattato il loro impatto su applicazioni reali.

Il principale problema evidenziato da Peng è che tutti i modelli presentati fino ad ora non sono ind-CCA e questa vulnerabilità potrebbe portare insieme a dei minimi

leak da parte del *decryption oracle* alla possibilità che un attaccante possa calcolare la chiave segreta.

3.2 Searchable Encryption

Questo particolare tipo di crittografia permette di poter eseguire query di ricerca anche sui ciphertext senza la necessità di doverli decifrare. Questo rende possibile la completa cifratura di un database mantenendone però la facilità di utilizzo. Inoltre, tramite questa tecnologia è verosimile pensare a scenari in cui la chiave segreta non è in possesso del proprietario del servizio ma dell'utente finale, assicurando così la reale segretezza dei dati presenti sul database. Un classico esempio di questo caso d'uso sono i servizi di *cloud storage* in cui la riservatezza del contenuto dei file è primaria.

I principali modelli di Searchable Encryption (SE) [4][5][6] si concentrano su una ricerca keyword-based, ovvero basato su match esatti delle keyword, è quindi importante tenere a mente questo particolare, anche se recentemente sono cresciuti anche i modelli che permettono la ricerca per sottostringhe.

La crittografia di tipo Searchable Encryption può essere suddivisa in due macro-gruppi in base al tipo di crittografia che viene usata, se simmetrica o asimmetrica. La principale differenza, oltre all'ovvia diversità negli algoritmi utilizzati, sta nella finalità per cui un certo modello è pensato. Uno schema basato su crittografia simmetrica avrà principalmente come scopo quello di avere un'unica entità che possiederà "a tutto tondo" dei dati. Con la crittografia asimmetrica invece è possibile dividere i produttori del dato, potenzialmente chiunque, e i consumatori, coloro che sono in possesso della chiave privata. Inoltre, utilizzando un modello a chiave pubblica si potrebbe creare un "ibrido" tra Searchable Encryption ed Attribute Based Encryption (descritto nella sezione 3.4), come dimostrato da Zhang et al. [7], riuscendo in questo modo a gestire al meglio i permessi di accesso ad ogni documento.

Storia La prima definizione formale della Searchable Encryption è attribuibile all'articolo Secure Indexes di Goh et al.[4] scritto nel 2003 in cui viene definito il modello di Searchable Encryption basato su *trapdoor* e nel quale vengono anche definiti gli standard di sicurezza ind-CKA e ind2-CKA che rappresentano l'indistinguibilità delle keyword di un documento una volta cifrate.

L'anno dopo, nel 2004, Boneh et al. [8] hanno creato questa tipologia di schema utilizzando la crittografia asimmetrica, l'uso di una crittografia a chiave pubblica permette di poter distinguere il produttore di un dato dal suo consumatore.

La definizione data da Goh et al. della Searchable Encryption è quella di un modello composto da 4 algoritmi:

Keygen(s): Dato un parametro di sicurezza s, ritorna la chiave privata K_{priv}.

Trapdoor(K_{priv}, w:) Data K_{priv} ed una parola w, ritorna la trapdoor T_w.

BuildIndex(D, K_{priv}): Dato un documento D e la chiave K_{priv}, ritorna l'indice $\mathcal{I}_D$.

SearchIndex(T_w, $\mathcal{I}_D$): Data la trapdoor T_w per la parola w e l'indice $\mathcal{I}_D$ relativo al documento D, ritorna 1 se $w \in D$ e 0 altrimenti.

Questa prima definizione di Searchable Encryption porta però con sé il problema di una bassa scalabilità, infatti tutti i modelli costruiti secondo la definizione data da Goh et al. hanno una complessità per la ricerca pari a $O(n)$, dove n rappresenta il numero

di documenti presenti nel database, questo perché il server deve applicare la trapdoor all'indice di ogni documento. Per quanto una complessità lineare sia considerata ottima in altri scopi nella ricerca, specialmente su grandi volumi di dati, è da considerarsi insufficiente.

Solo nel 2006 Curtmola et al. [5] presentarono il primo modello che, grazie all'utilizzo di una struttura di supporto, riusciva ad estrarre i documenti ricercati in tempo lineare rispetto al numero di documenti da ritornare e costante rispetto al numero di documenti presenti nel database. In particolare, in questo caso la struttura di supporto era composta da un array che al suo interno conteneva multiple linked list, una per keyword, in cui ogni nodo conteneva un documento associato alla keyword. Dato che questo array è sia cifrato che rimescolato in modo randomico per trovare la testa della linked list relativa alla keyword da ricercare viene utilizzata una lookup table apposita.

Negli anni successivi si creò sempre maggiore attenzione alla segretezza rispetto al search pattern dell'utente e quindi anche ai concetti di *Forward Privacy* (o *Forward Security*) e *Backward Privacy* (o *Backward Security*). I primi a descrivere formalmente entrambe le proprietà furono nel 2017 Bost et al. [9] con le definizioni:

Forward Security : Un record aggiunto o modificato successivamente ad una query di ricerca non deve rispondere correttamente alla query.

Backward Security : Un record rimosso precedentemente ad una query di ricerca non deve rispondere correttamente alla query.

Problemi attuali Al giorno d'oggi gli schemi che hanno come scopo quello di ricerca testuale senza limitarsi all'utilizzo di keyword sono ancora poco efficienti e per questo difficilmente utilizzabili in contesti reali.

3.3 Order-preserving Encryption e Order-revealing Encryption

Sia la Order Preserving Encryption (OPE) che la Order Revealing Encryption (ORE) sono tipi di modelli crittografici che hanno come obbiettivo quello di rendere possibile il confronto rispetto all'ordine (numerico o lessicografico) di due ciphertext senza la necessità di doverli decifrare. La differenza tra i due è che un modello per essere OPE deve far coincidere l'operazione di confronto sul ciphertext a quella sul plaintext, nel ORE invece è sufficiente che esista una funzione capace di calcolare se un ciphertext è minore, uguale o maggiore ad un altro ciphertext. Come è evidente la OPE è un sotto-caso della ORE, questo perché dato che ne l2012 è stato provato da Xiao et al.[10] che qualsiasi modello OPE statico presenta dei leak consistenti, grazie a cui sarebbe possibile calcolare la metà dei bit più significativi del valore originario partendo da un singolo ciphertext, l'unico modo per rafforzare le garanzie di sicurezza che il modello poteva garantire era quello di rendere più "flessibile" la sua definizione.

Storia Il primo modello di OPE è descritto da Agrawal et al. [11] nel 2004, mentre la prima costruzione di un modello del tipo ORE avviene solo nel 2015 [12] anche se schemi di questo tipo erano indicati già prima come *committed efficiently-orderable encryption*.

Problemi attuali La crittografia di tipo OPE, come già detto, è impossibile da creare senza avere dei cospicui leak dei dati.

La crittografia di tipo ORE anche se nettamente più sicura della sua predecessora perde la completa trasparenza rispetto al DBMS usato. Ovvero utilizzando OPE il server avrebbe utilizzato un normale operatore di confronto come se il valore non fosse cifrato, invece con ORE il server deve essere cosciente che il campo sia cifrato e deve utilizzare la funzione apposita.

3.4 Attribute Based Encryption

La Attribute-based Encryption (ABE) è una tipologia di crittografia a chiave pubblica che permette di cifrare un messaggio non appositamente per un destinatario ma secondo le caratteristiche (attributi) che i destinatari dovrebbero avere.

Storia La prima descrizione di questa crittografia risale a Sahai et al. nel 2005 [13] come evoluzione della Identity-based Encryption (IBE). Più precisamente definisce l' identità come un insieme di attributi, proprio per questo inizialmente venne chiamata Fuzzy Identity-based Encryption in quanto la differenziazione tra le identità non è più chiara ma "sfocata".

Originariamente questo tipo di crittografia richiedeva una autorità centrale incaricata di distribuire le chiavi con i permessi adeguati e i token necessari alla cifratura, ma ormai molte ricerche hanno mostrato metodi per avere autorità multiple, così da poter dividere la responsabilità per ogni attributo diverso.

Problemi attuali La più grande sfida attuale per questo tipo di crittografia è la revoca degli attributi, questo perché nei sistemi attuali significherebbe un cambio del token per tutto il sistema.

Proprio per questo motivo le principali soluzioni proposte finora sono di tipo *lazy* in cui gli attributi hanno una scadenza o i client devono controllare, attraverso un'autorità, ad intervalli di tempo regolari se ci sono modifiche da effettuare.

3.5 Proxy re-encryption

La *Proxy re-encryption* è un crittosistema che permette, in un sistema con crittografia asimmetrica, di poter trasformare un ciphertext cifrato con chiave pubblica di Alice in un ciphertext decifrabile da Bob, con la sua chiave privata, senza che il proxy sia a conoscenza della chiave privata di Alice. Questo grazie alla creazione, da parte di Alice, di una chiave apposita che permette al proxy la trasformazione.

Un classico caso d'uso per questa tecnologia è quello di un inoltro automatico delle e-mail, nel caso queste fossero criptate. L'utilizzo della proxy re-encryption permetterebbe di attribuire al server mail il compito dell'inoltro minimizzando il più possibile la quantità di informazioni cedute.

Storia Il concetto di Proxy re-encryption esiste dalla fine degli anni novanta con il nome di *Proxy Cryptography* [14] ed ha trovato sviluppo negli ultimi anni specialmente in relazione alla crittografia IBE con il nome di *Identity-based conditional proxy re-encryption* (IBCPRE)

Problemi La proxy re-encryption non soffre di veri e prori problemi l'unico aspetto su cui va posto attenzione è che qualora la *re-encryption key* dovesse essere in possesso

di un terzo non ci sarebbe modo di revocarla. Ad esempio mettiamo caso che Alice invii dei messaggi a Bob e Bob per un periodo incarichi un server proxy di inoltrare re-cifrando i messaggi a Charlie. Nel momento in cui Bob notifica il server proxy di terminare l'inoltro Charlie non avrebbe più accesso a questi messaggi, ma se Charlie dovesse ottenere la *re-encryption key* dal proxy non ci sarebbe modo di fermare Charlie dal re-cifrare i messaggi diretti a Bob se non quello di cambiare le chiavi crittografiche di Bob.

Capitolo 4

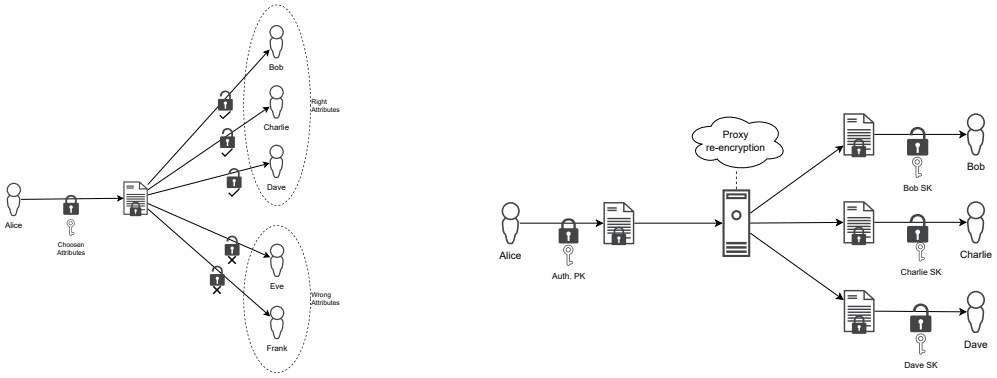
Scelta delle tecnologie

4.1 Valutazioni dei diversi tipi di tecnologie crittografiche

A fronte di un colloquio con il mio tutor aziendale siamo giunti alla conclusione che la tecnologia più utile e di facile inserimento fosse la *Searchable Encryption* questo perché per il sistema sarebbe per lo più trasparente delegandone la responsabilità alla classe repository che si occuperebbe dell'interfacciarsi col database.

L'altra tecnologia per cui c'era stato un cospicuo interesse era la *Homomorphic Encryption* che però è ancora in uno stato troppo precoce per il suo utilizzo, data la grande inefficienza e le preoccupazioni per la mancanza della proprietà ind-CCA.

Inoltre, vi era anche la curiosità di implementare insieme alla Searchable Encryption una funzionalità per distribuire il permesso di decifrare le informazioni al meglio, lavorando direttamente a livello di crittografia senza bisogno di un'entità intermediaria. A questo scopo sarebbero state utili la *ABE* 3.4 o la *Proxy re-encryption* 3.5 in quanto con la prima sarebbe stato possibile definire degli attributi per identificare nel modo migliore i permessi dei singoli individui e con la seconda si sarebbe potuto avere un'autorità (centrale o non), che non avrebbe accesso ai plaintext, che si occupa di trasformare il ciphertext originale in quello specifico per ogni destinatario.



(a) Usando Proxy re-encryption **(b)** Usando ABE

Figura 4.1: Metodi per ottenere una crittografia limitata a dei gruppi

In conclusione, però, abbiamo deciso che sarebbe bastato limitarsi alla sola Searchable Encryption dato che il tempo era limitato.

4.2 Valutazioni dei diversi tipi di Searchable Encryption

Successivamente alla decisione della tecnologia su cui incentrare il mio tirocinio mi sono concentrato sull'analisi di tutte le diverse possibilità che essa offriva.

Durante la scelta delle soluzioni ho incontrato un grande numero di decisioni da prendere e dato che il numero di soluzioni da me implementabili era molto limitato ho scelto, ove possibile, la alternativa più adatta.

Innanzitutto, è importante sapere il tipo di applicazione in cui questa tecnologia andrà inserita

- Il fine della SE in questa applicazione non sarà quello di rendere gli utenti possessori dei propri dati ma quello di rendere il database sicuro da un potenziale attacco passivo, per questo il database (o almeno i campi necessari) saranno sempre cifrati e solo piccole porzioni di dati verranno richieste dall'application server e poi decifrate (in-use). In questo modo si eviterebbe il problema, di cui soffrono molti database criptati su disco, che al momento dell'avvio vengono caricati in memoria e completamente decriptati.

- All'interno dell'applicazione non tutti i dati avranno la stessa necessità di essere cifrati, quindi come è logico il compromesso prestazionale che si ritiene accettabile varia da dato a dato. Per questo è preferibile non avere un unico modello ma una "scala" di modelli tra cui scegliere.

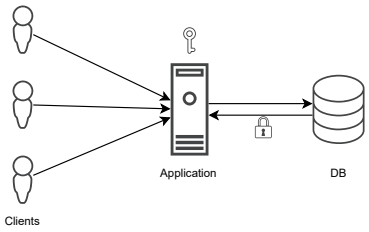


Figura 4.2: Rappresentazione semplificata dell'applicazione

Le varie decisioni che ho dovuto prendere sono state:

Crittografia simmetrica o asimmetrica Ho preferito scegliere di utilizzare la crittografia simmetrica per tutte le soluzione dato che come tecnologia è notoriamente più veloce della sua alternativa ma soprattutto perché nel nostro caso d'uso il database verrà gestito da un'unica entità, l'application server.

Keyword o Substring Come accennato anche nella sezione relativa alla descrizione della SE 3.2 i modelli che eseguono un ricerca anche nelle sottostringhe sono ancora acerbi e poco ottimizzati, quindi ho preferito limitarmi a dei modelli keyword-based

Single-keyword o Multi-keyword Purtroppo anche i modelli keyword based ma capaci di eseguire query contenenti keyword multiple, in unica richiesta e mantenendo una sicurezza elevata, sono ancora piuttosto inefficienti perciò la scelta migliore è quella di utilizzare dei modelli single-keyword. In ogni caso qualora servisse si potrebbero comunque ottenere i risultati di query multi-keyword con lo svantaggio di richiesta di banda maggiore, tra client e DB, ed uno sforzo di computazione superiore da parte del client.

Supporto per più client La possibilità che un indice crittografico possa essere utilizzato da più di un client purtroppo non è così scontato, infatti soprattutto le soluzioni più all'avanguardia richiedono che il client tenga in memoria una struttura di piccole dimensioni che permetta di interpretare l'indice nel modo corretto e a volte questa struttura viene cambiata molto frequentemente rendendo impraticabile la sincronizzazione tra più client. Data la topologia del prodotto finale la limitazione ad un singolo client non risulterebbe problematica, quindi ho considerato questa limitazione trascurabile nei casi in cui lo specifico modello fosse promettente in altri aspetti.

Dinamico o non dinamico Dovendo utilizzare questa tecnologia in un contesto operativo non è pensabile la ricostruzione dell'indice ad ogni modifica del database quindi è d'obbligo la scelta di un modello che sia dinamico.

Quindi, riassumendo, per la scelta dei modelli mi sono limitato ad algoritmi di *Dynamic Symmetric Searchable Encyption* (DSSE) keyword-based e limitati a query composte da una singola keyword alla volta. Questa tipologia di modelli essendo trattato in maniera esaustiva in letteratura è di facile reperibilità ed inoltre grazie alle continue revisioni è arrivata ad uno stadio di quasi maturità. Nonostante questo però soffre dei problemi riguardante la limitazione della ricerca a match completi che in molti casi non è sufficiente per una ricerca soddisfaciente su un dato.

Capitolo 5

Analisi e metodologia della soluzione

Purtroppo, non è ancora possibile garantire a tutti i dati la massima sicurezza senza alterare notevolmente le prestazioni del servizio, per tali ragioni ho dovuto scegliere il miglior compromesso per ogni tipologia di dato. Questo perché ogni informazione è diversa ed ha una sua riservatezza, per esempio l'ufficio dove lavora un dipendente è completamente diverso dalla lista di servizi a cui esso ha accesso.

Per semplificare questo processo di decisione ho selezionato diversi modelli crittografici che spaziano dal più semplice, con garanzie di sicurezza essenziali, ad un modello presentato all'interno di un paper di ricerca[15],con maggiori garanzie, e li ho messi a confronto.

Tutti i modelli proposti e analizzati sono del tipo Symetric Searchable Encryption e si basano sull'uso di keyword per la ricerca, richiedono quindi un match perfetto.

Ricordo che questa analisi si concentra sul metodo per la ricerca e come esso venga implementato, in tutti i modelli si presuppone che il campo contenente la o le keyword sia cifrato, usando ad esempio AES o altri algoritmi di cifratura considerati sicuri ai giorni d'oggi.

5.1 Modelli analizzati

5.1.1 Aggiunta di un campo contenente l'hash

Questo modello, preso da un articolo di Scott Arciszewski su Sitepoint[16], consiste nell'aggiunta in ogni record di un campo contenente l'hash del valore da ricercare. Anche non offrendo garanzie elevate di sicurezza offre la possibilità di ricercare su dati cifrati con un overhead molto basso e mantenendo la stessa complessità della ricerca su un campo in chiaro.

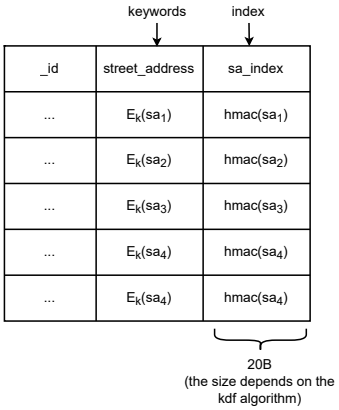


Figura 5.1: Rappresentazione in forma tabellare del modello dell'indice formato dal campo hash

Funzionamento

Il nuovo campo conterrà al suo interno un hash prodotto da un algoritmo di derivazione di una chiave crittografica (o KDF) utilizzando il valore del campo come salt.

È importante adottare questa tipologia di hash in quanto l'utilizzo di hash che non utilizzano chiave come md5 o SHA256 potrebbero portare a vulnerabilità ad attacchi del tipo rainbow table e chiunque abbia accesso al server potrebbe effettuare ricerche sui dati. Inoltre, l'aggiunta di un semplice salt non sarebbe una soluzione valida, data la possibile presenza di domini a bassa entropia.

Per effettuare la ricerca basterà ricalcolare l'hash della parola voluta con la chiave ed eseguire la ricerca sul database.

Questo modello, inoltre, può essere facilmente adattato per poter funzionare anche in casi dove ci sono multipli valori per campo, per esempio all'interno di un array. Per fare questo basta ricreare l'array contenente gli hash prodotti da ogni keyword.

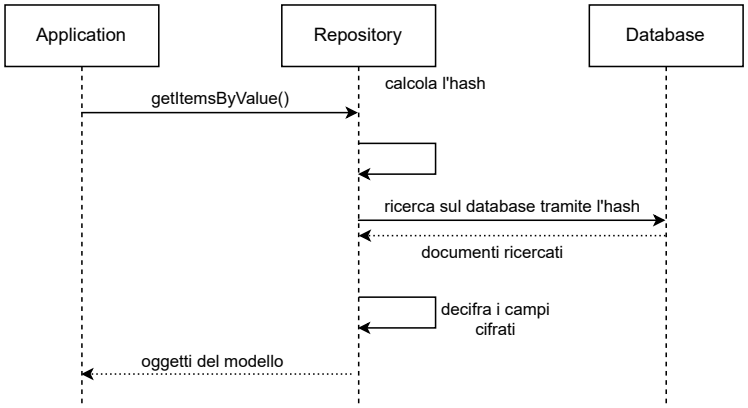


Figura 5.2: Diagramma di sequenza della ricerca con l'indice formato dal campo hash

Pro e contro

Pro:

- Implementazione semplice;
- Mantiene la stessa velocità di una normale ricerca.

Contro:

- Soffre di leaking rispetto alla ripetizione di keyword in più record;
- Soffre di leaking rispetto alla numerosità di keyword per ogni record;
- Non offre né Forward né Backward privacy.

Complessità e dimensioni

La complessità di questo modello dipende principalmente dal numero di hash da creare e su cui poi cercare, che dipende dal numero totale di coppie documento-keyword.

	Build	Search	Update	Size
Index	$O(\|pairs\|)$	$O(\log \|pairs\|)$	$O(\log n)$	$O(\|pairs\|)$
No Index		$O(\|pairs\|)$	$O(n)$	

Tabella 5.1: Complessità del modello dell'indice formato dal campo hash

Se però il campo a cui abbiamo applicato l'indice è singolo e di sicuro non un array possiamo semplificare e dire che dipende dal numero di documenti da indicizzare.

Implementazione all'interno del PoC

L'implementazione non ha presentato problematiche data anche l'intrinseca semplicità del modello. In questo caso l'attenzione si è focalizzata sulla individuazione dell'algoritmo di KDF e sull'adattamento dell'algoritmo per gestire campi contenenti anche array di keyword.

	Build	Search	Update	Size
Index	$O(n)$	$O(\log n)$	$O(\log n)$	$O(n)$
No Index		$O(n)$	$O(n)$	

Tabella 5.2: Complessità del modello dell'indice formato dal campo hash su campi singoli

La mia scelta per l'algoritmo di KDF è stata un HMAC basato su SHA256 e per quanto riguarda l'algoritmo di cifratura son riuscito ad adattarlo con un semplice controllo nella funzione di creazione dell'indice. Inoltre, dato che sul database è salvato come stringa in base64 l'hash ottenuto viene codificato nel formato corretto.

```
if(Array.isArray(plainValue))
  hashedValue = plainValue.map(val =>
    HmacSHA256(val, process.env.SECRET_KEY as string)
      .toString(enc.Base64)
  )
else
  hashedValue = HmacSHA256(plainValue, process.env.SECRET_KEY as string)
    .toString(enc.Base64)
```

Listing 5.1: Controllo della presenza di multiple keyword nella costruzione dell'indice

5.1.2 Indice invertito criptato

Questo schema è una modifica dell'indice invertito dove sia le chiavi che i valori sono criptati. Dei primi (keyword) viene salvato l'hash mentre dei secondi (lista di id) una cifratura randomizzata tramite `nonce` o `iv`. Come per lo schema precedente l'hash è il risultato di un algoritmo di derivazione di una chiave crittografica come HMAC o KDF.

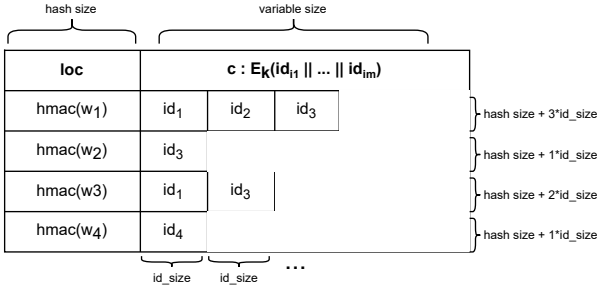


Figura 5.3: Rappresentazione in forma tabellare del modello indice invertito criptato

Funzionamento

Nella tabella dell'indice è presente un record per ogni keyword presente nel campo di cui si sta costruendo l'indice. Il record è composto dalla chiave, la keyword, e da una lista di id corrispondenti ai documenti contenenti la keyword. Per effettuare una ricerca bisogna individuare sull'indice il record che ha come chiave l'hash della keyword da cercare, una volta ritornato il record si potrà decifrare la lista degli id.

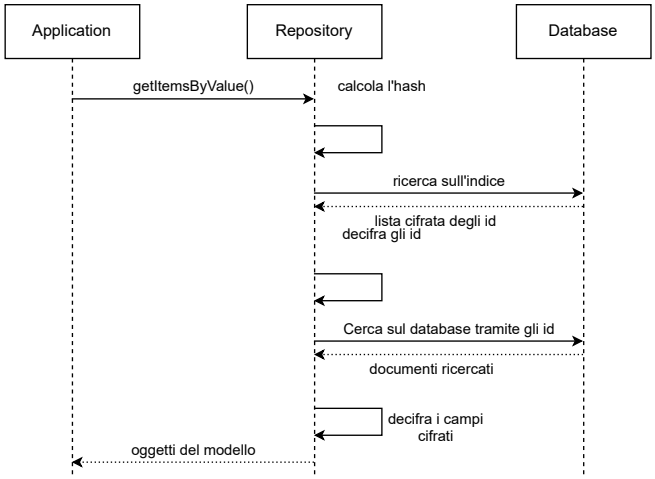


Figura 5.4: Diagramma di sequenza della ricerca con l'indice invertito criptato

Pro e contro

Pro:

- Tempo di ricerca ottimale;
- Non è soggetto a leaking riguardo i record in cui è ripetuta una keyword.

Contro:

- É soggetto leaking riguardo la frequenza delle keyword, senza però rivelarle;
- Non offre né Forward né Backward privacy.

Complessità e dimensioni

La complessità di ricerca ed aggiornamento dell'indice dipende principalmente dal numero di keyword presenti all'interno di esso ma nel caso di una modifica va eseguita l'iterazione (in locale) della lista degli id corrispondenti alla keyword. Per la creazione invece la complessità è da attribuire alle operazioni di raccolta dei record dal database ($|ID|$) ed alla cifratura e all'inserimento dei record dell'indice ($|W|$).

	Build	Search	Update	Size
Index	$O(\|W\| + \|ID\|)$	$O(\log \|W\|)$	$O(\log \|W\| + \|ID_w\|)$	$O(\|W\| + \|pairs\|)$
No Index		$O(\|W\|)$	$O(\|W\| + \|ID_w\|)$	

Tabella 5.3: Complessità del modello dell'indice invertito criptato

Implementazione all'interno del PoC

Per l'implementazione ho preferito utilizzare rispettivamente l'HMAC basato su SHA256 per generazione degli hash e l'AES per la cifratura della lista degli id.

5.1.3 Indice invertito criptato e concatenato

Questo schema non è che una revisione del precedente modello atta a mitigare il leak relativo alla distribuzione dei record rispetto alle keyword.

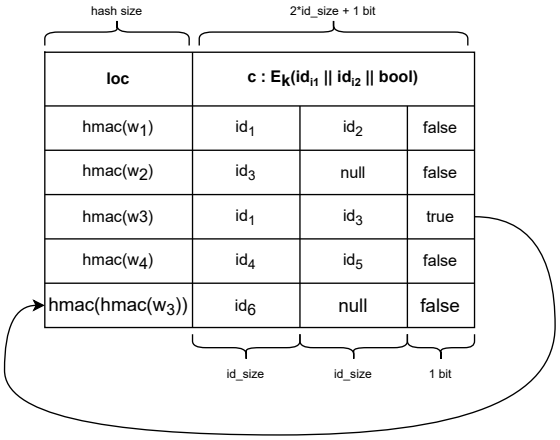


Figura 5.5: Rappresentazione in forma tabellare del modello indice invertito criptato e concatenato

Funzionamento

La mitigazione del leakage della distribuzione dei record rispetto alle keyword avviene tramite la forzatura di una dimensione fissata della lista. Se gli elementi sono inferiori lo spazio vuoto viene riempito attraverso un padding, se invece superano la dimensione massima viene creato un secondo record all'interno in cui vengono inseriti gli id in eccesso e così via. La presenza o no di un altro record è segnalata tramite un bit aggiuntivo che se settato ad 1 ne indica la presenza. La chiave di questo nuovo record sarà calcolata come hash della chiave precedente (quindi un hash dell'hash).

Alternativa: Invece di eseguire un hash ricorsivo si potrebbe eseguire l'hash della keyword concatenata con il numero del record per quella specifica keyword: HMAC(kw$\|i$). Questa variante renderebbe possibile la computazione indipendente dei vari hash, ma oltre questa piccola differenza è indifferente quale metodo venga utilizzato.

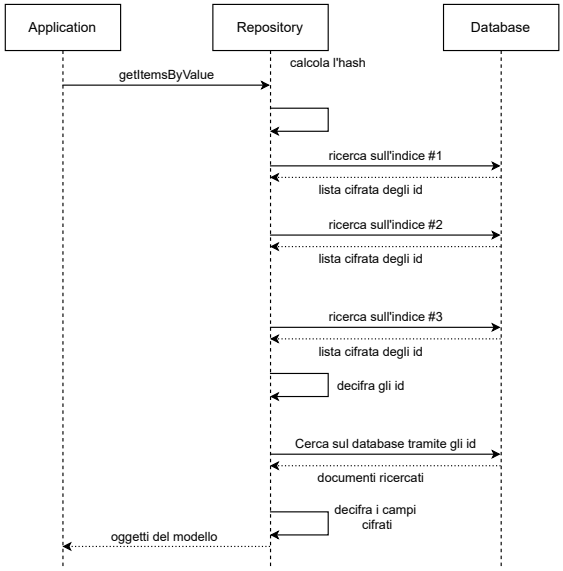


Figura 5.6: Diagramma di sequenza della ricerca con l'indice formato dal campo hash e concatenato

Pro e contro

Pro:

- Non è soggetto a leaking riguardo la distribuzione dei record rispetto alle keyword;

Contro:

- Aggiunge un numero di roundtrip pari a al numero di record dell'indice per la keyword che sto cercando;
- Non offre né Forward né Backward privacy.

Complessità e dimensioni

All'interno di questo modello come spiegato precedentemente c'è una variabile aggiuntiva ovvero la lunghezza del campo contenente gli id dei documenti nell'indice. Dato che la sua dimensione è cruciale per definire la complessità del modello ho deciso di definire con p la capienza massima di id in un record dell'indice.

Le complessità e le dimensioni sono simili a quelle del precedente, che è alla base di questo modello, la differenza è dovuta alla frammentazione dei record causata dalla limitazione della loro grandezza massima per cui le occorrenze di $|W|$ diventano $\frac{|pairs|}{p}$ per quanto riguarda il tempo di creazione e la dimensione dell'indice, per la ricerca o l'aggiornamento dell'indice invece il numero di richieste da eseguire invece che una sono $\frac{|ID_w|}{p}|W|$.

I valori di $\frac{|pairs|}{p}$ e $\frac{|ID_w|}{p}|W|$ non sono precisi ma la media si assesterà sull'ordine di grandezza di questi valori.

	Build	Search	Update	Size
Index	$O(\|W\| + \frac{\|pairs\|}{p})$	$O(\frac{\|ID_w\|}{p} \log \|W\|)$	$O(\frac{\|ID_w\|}{p} \log \|W\| + \|ID_w\|)$	$O(\frac{\|pairs\|}{p} + \|pairs\|)$
No Index		$O(\frac{\|ID_w\|}{p} \|W\|)$	$O(\frac{\|ID_w\|}{p} \|W\| + \|ID_w\|)$	

Tabella 5.4: Complessità del modello dell'indice invertito criptato e concatenato

5.1.4 Schema da Zheng et al. (2020)

Questo schema è stato proposto da Zheng et al.[15] e consiste in un indice invertito dove nel campo dei valori non viene inserita la lista di id ma un vettore di bit in cui ogni bit indica se il documento corrispondente presenta ($b_i = 1$) o no ($b_i = 0$) la keyword. Inoltre, questo schema fa uso di Pseudorandom Function (PRF), Pseudorandom Generator (PRG) e tecniche di offuscamento per soddisfare i requisiti di Forward e Backward security.

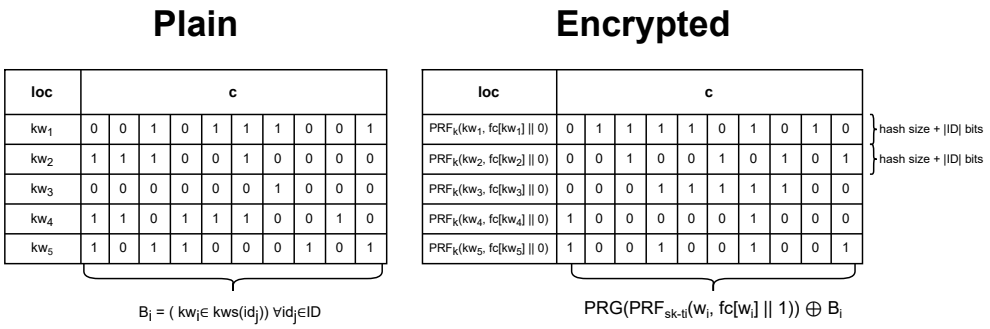


Figura 5.7: Rappresentazione in forma tabellare del modello di Zheng et al.

Funzionamento

Metodi pseudocasuali Questo modello questo fa uso di:

- PRF, una funzione pseudocasuale ($F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$) che è riconducibile ad una funzione di crittografia casuale che utilizza $\mathcal{X}$ come seed;
- PRG, un generatore pseudocasuale ($G : \mathcal{X} \to \mathcal{Y}$) che dato il seed $\mathcal{X}$ produce un numero pseudocasuale di dimensione indicata

Shuffling Lo shuffling o rimescolamento serve per nascondere al server l'access pattern dell'utente spostando in una nuova posizione (loc, la chiave) il record ogni volta che viene acceduto. Questo avviene tramite una apposita mappa chiamata `FileCnt` che tiene conto di quante volte ogni keyword è stata acceduta ed il suo valore viene utilizzato per calcolare la cifratura sia della keyword che del vettore di bit.

$$loc_i = F_K(w_i, FileCnt[w_i] \| 0)$$
$$c_i = G(F_K(w_i, FileCnt[w_i] \| 1)) \oplus B_i$$

k-anonimity La k-anonimity è una proprietà che il modello raggiunge chiedendo ogni volta al server non solo la keyword da ricercare ma anche altre $k - 1$ keyword in

modo che il server non possa risalire al record realmente acceduto. Questa richiesta di più record allo stesso momento fa in modo anche che il server non riesca a tracciare i vari spostamenti eseguiti dallo shuffling.

Processo di creazione dell'indice Per la creazione dell'indice viene creata una mappa indicante la corrispondenza tra keyword e documenti. Poi per ogni keyword viene settato il contatore a 0 e calcolate le cifrature del loc (la chiave dell'indice) e di c (il campo contente le corrispondenze).

Più precisamente loc sarà un hash prodotto dalla formula descritta precedentemente, c invece è il risultato di B_i xorato con un numero casuale generato dal PRG usando l'hash come seed.

Processo di ricerca Quando c'è bisogno di consultare l'indice per rispettare la k-anonimity non va ricercato usando una unica loc ma ne vanno calcolate e richieste altre $k-1$. Successivamente possiamo estrarre da tutte le c ritornate i rispettivi B_i, in particolare per la keyword ricercata dal B_i ricaviamo gli indici dei documenti con cui interrogare la collezione principale. Come ultima cosa dobbiamo eliminare i record dell'indice appena letti e sostituirli con dei nuovi dato che il contatore di fileCnt di ognuno di essi è stato incrementato e quindi anche le loro cifrature sono cambiate.

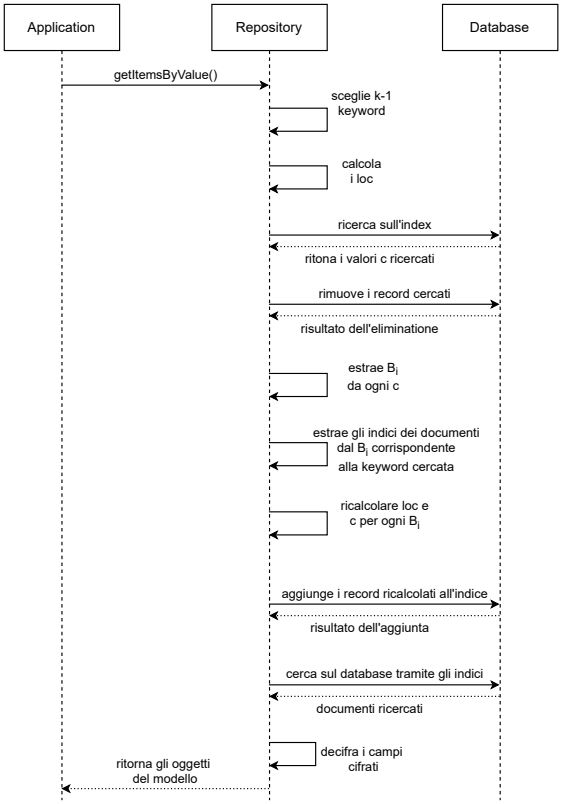


Figura 5.8: Diagramma di sequenza della ricerca utilizzando il modello di Zheng et al.

Pro e contro

Pro:

- Offre sia Forward che Backward security;
- Possibile inserire più keyword per record;
- L'indice rimane molto compatto* in scenari in cui ci sono poche keyword e molto usate.

Contro:

- Aggiunge un overhead consistente per gestire lo shuffle;
- L'indice raggiunge dimensioni notevoli* se l'occorrenza delle keyword è bassa.

*La grandezza dell'indice è intesa rispetto alle altre soluzioni viste

Complessità

Le complessità di questo algoritmi sono dipendenti direttamente dal numero di keyword e documenti e non, al contrario dei precedenti, al numero di coppie documento-keyword, questo perché come spiegato precedentemente per ogni keyword viene salvato un bit per ogni record indipendentemente dalla connessione tra lo specifico documento e la keyword.

Inoltre, la complessità per le operazioni di ricerca o aggiornamento è influenzata anche da k, ovvero il numero di record richiesti ad operazione

	Build	Search	Update	Size
Index	$O(\|W\| \cdot \|ID\|)$	$O(k \cdot \log \|W\| + \|ID\|)$	$O(k \cdot \log \|W\|)$	$O(\|W\| \cdot \|ID\|)$
No Index		$O(\|W\| + \|ID\|)$	$O(\|W\|)$	

Tabella 5.5: Complessità del modello proposto da Zheng et al.

Implementazione nel PoC

Campo auto increment L'indice creato da Zheng et al.[15] permette di rappresentare ogni documento utilizzando un solo bit questo perché probabilmente è stato pensato avendo in mente database SQL in cui gli id `AUTO_INCREMENT` sono la normalità. In ambito noSQL però i valori auto-increment sono sconsigliati in quanto ne potrebbero ostacolare la scalabilità per motivi di concorrenza, detto questo però l'unico modo per poter realizzare questo modello è utilizzare un campo simil-autoincrement.

Per fare ciò c'è bisogno di aggiungere al database una breve funzione trigger che inserisca il valore ad ogni insert e tenga conto dell'ultimo inserito. All'interno del PoC per semplificare ho preferito inserirli con un piccolo script sul client una tantum.

FileCnt in locale Lo schema originale è pensato perché sia il client, da noi l'applicazione, che si occupi di tenere la mappa FileCnt cosa che però nel nostro specifico use case non è sempre possibile. Il problema si pone in quanto il server in cui è contenuta l'applicazione potrebbe essere soggetto a riavvi o wipe. Questo renderebbe impossibile una persistenza continuativa della mappa sul server (applicazione). Per risolvere questo problema ci sono due possibili soluzioni:

- Salvare fileCnt sul database, ma il salvataggio ad ogni operazione oltre ad essere onerosa potrebbe rivelare informazioni che possano aiutare un possibile attaccante. Per questo l'aggiornamento a db andrebbe fatto solamente prima dell'arresto dell'applicazione, per dopo caricarlo all'avvio, se però per qualche problema non si riuscisse ad aggiornare sarebbe impossibile riconoscerlo al nuovo avvio e diventerebbe palese solo quando le query di ricerca non avranno risposta. A questo punto l'unica soluzione sarebbe il re-build dell'indice

- Ricostruire l'indice e FileCnt annesso ogni volta da zero, il tempo di questa operazione anche se consistente non è eccessivo come si può notare dai test [Tabella 5.8] (in cui era anche fortemente limitato dalla piattaforma).

Va comunque osservato come nel modello originale essendo il fileCnt continuativo i suoi valori non si ripetevano e questo contribuiva alla Forward e Backward Security se ad ogni riavvio reinizializziamo FileCnt a 0 andiamo a perdere quella proprietà. La soluzione a questo problema è semplicemente quella di inizializzare i valori di FileCnt ad un numero casuale.

PRG e PRF Come spiegato nel paragrafo del funzionamento questi algoritmi sono fondamentali per questo modello ed una loro implementazione poco attenta potrebbe intaccare la sicurezza. È bene specificare che le primitive utilizzate non derivano dalle librerie di base ma da delle librerie specifiche per la crittografia, in particolare crypto-js.

```
function F(...msgs: any[]): string{
    return HmacSHA256(msgs.join(), process.env.SECRET_KEY!)
        .toString(enc.Base64)
}

function G(seed: string, dbSize: number): bigint{
    return toBigIntBE(randomBytesSeed(Math.ceil(dbSize/8.0), seed))
}
```

Listing 5.2: Implementazione dei due algoritmi

Calcolo di B_i Per la gestione di B_i dato che i normali interi non bastano la strada migliore è l'utilizzo dei bigint che non hanno un limite di dimensione e permettono operazioni come quella di xoring fondamentale per questo algoritmo. Per la creazione di B_i mi è bastato fare la somma di tutti gli indici come esponenziale in base due per ottenere il valore voluto. Questa soluzione è molto efficiente anche dato l'ottimizzazione dei calcolatori per svolgere le potenze di 2 (shift di 1 bit). Per la lettura invece converto il bigint in una stringa binaria e itero controllando il valore di ogni bit.

```
let B_i = 0n

incs.forEach((inc)=>{
    B_i += 2n ** BigInt(inc)
})
let rnd = ...
let c = rnd ^ B_i
```

Listing 5.3: Calcolo e utilizzo di B_i

Per il calcolo di B_i dato `incs`, un array contenente gli id `AUTO_INCREMENT` dei documenti contenenti la keyword w_i, basta tramite un forEach calcolare la somma di tutti gli esponenziali base 2 degli id. Poi per il calcolo di c basta eseguire lo xor con con la parte randomizzata.

```
incs = [...B_i.toString(2)].reverse().map((bit, i) => {
    if (bit === '1')
        return i
    return undefined
}).filter(Number) as number[]
```

Listing 5.4: Lettura di B_i

Per la lettura, invece, B_i viene convertito in un array di caratteri `'0'` e `'1'` in cui gli indici dove è presente `'1'` rapprsentano la presenza della keyword nel documento e quindi ne ritorna l'indice.

5.2 Comparazioni tra i vari modelli

5.2.1 Caratteristiche dei modelli

I modelli sono stati presentati in ordine crescente rispetto alle garanzie di sicurezza come visibile nella tabella 5.6. In particolare, si può notare come il modello proposto da Zheng et al. sia quello più sicuro anche se è l'unico modello per cui è impossibile supportare client multipli, che potrebbe rivelarsi un problema in ottica di scalabilità. Per quanto riguarda gli altri tre modelli si può vedere come le garanzie di sicurezza scalino man mano anche se nessuno dei tre riesce ad essere né forward che backward private.

Il modello privo di indice (Only Enc.) che consiste nei soli campi cifrati potrebbe sembrare strano sia così sicuro ma è proprio l'assenza dell'indice che ne rende impossibile trarre informazioni, anche per l'utente come sarà visibile nelle prossime comparazioni.

	Fw Secure	Bw Secure	Nasconde uguaglianze	Protegge distribuzione	Multiclient	Più keyword per record
Only Enc.	Sì	Sì	Sì	Sì	Sì	Sì
Hash Field	No	No	No	No	Sì	Sì
Enc.Inverted Intex	No	No	Sì	No	Sì	Sì
Chained Enc.Inverted Intex	No*	No	Sì	Sì	Sì	Sì
Zheng et al. 2020 [15]	Sì	Sì	Sì	Sì	No	Sì

*Si può implementare la feature con qualche modifica

Tabella 5.6: Caratteristiche dei diversi schemi

5.2.2 Complessità

Come anticipato nella sezione precedente è evidente come il modello privo di indice (Only Enc.) non sia adatto per un utilizzo reale data la complessità lineare necessaria per qualsiasi tipo di operazione. Per quanto riguarda gli altri schemi hanno tutti una complessità logaritmica rispetto al numero di keyword, tranne il modello "Hash Field" che non avendo un vero e proprio indice ha complessità logaritmica rispetto al numero di documenti.

	Computation		Comunication				
	Search	Update	Search	Update	Search RT	Index size	Client storage
Only Enc.	$O(\|ID\|)$	$O(\|ID\|)$	$O(\|ID\|)$	$O(\|ID\|)$	2 o $\|ID\|$?	—	0(1)
Hash Field	$O(\log \|ID\|)$	$O(\log \|ID\|)$	O(1)	O(1)	1	$O(\|ID\|)$	$O(1)$
Enc.Inverted Intex	$O(\log \|W\|)$	$O(\log \|W\|)$	O(1)	O(1)	2	$O(\|W\| + \|pairs\|)$	$O(1)$
Chained Enc.Inverted Intex	$O(p \cdot \log \|W\|)$	$O(p \cdot \log \|W\|)$	O(p)	O(p)	p + 1	$O(\|W\| + \|pairs\|)$	$O(1)$
Zheng et al. 2020 [15]	$O(k \cdot \log \|W\| + \|ID\|)$	$O(k \cdot \log \|W\|)$	O(k)	O(k)	4*	$O(\|W\| \cdot \|ID\|)$	$O(\|W\|)$

Tabella 5.7: Complessità dei diversi schemi

5.2.3 Tempi risultanti dal PoC

Il PoC è stato scritto in Typescript e opera su database MongoDB. Come dataset è stato utilizzato `sample_mflix`, offerto da MongoDB per effettuare test. Mi sono concentrato sulla collezione `movies` data la sua sufficiente popolosità (∼23 000 documenti) e la possibilità di poter spaziare tra tutte le tipologie di campi su cui costruire indici.

Per il testing ho selezionato:

- Il campo **`title`** è una stringa ed è quasi unica per ogni record, ci sono però delle eccezioni (ad es. Peter Pan), questo esempio serve a rappresentare tutta quella classe di campi in cui il rapporto tra keyword e document è fortemente lineare;
- Il campo **`genres`** è un array di stringhe, i generi totali sono molto limitati (solamente 25) e vengo ripetuti molte volte, questo esempio rappresenta l'esatto opposto rispetto al campo `title` dove poche keyword, il cui numero è indipendente dal numero di record, vengono ripetute molte volte.

Ho misurato i tempi impiegati da ogni modello per la costruzione dell'indice e per la ricerca e li ho comparati con l'equivalente operazione su una normale collezione in chiaro. Ho inoltre testato l'influenza della costruzione di un indice B-tree a supporto di ognuno degli indici ed il risultato è riportato in tabella 5.8.

Per effettuare questi test la piattaforma utilizzata è stata una Virtual Machine di Azure fornita di 1 vcore e 2 GiB di RAM come server ed il servizio gratuito MongoDB Atlas per il database.

	Without Index				With Index			
	Build		Search		Build		Search	
	Title	Genres	Title	Genres	Title	Genres	Title	Genres
Plain	—	—	57ms	395ms	—	—	21ms	864ms
Only Enc.	—	—	5.6s	5.8s	—	—	5.6s	5.8s
Hash Field	33s	31s	121ms	392ms	34s	31s	10ms	282ms
Enc.Inverted Intex	19s	6s	57ms	382ms	21s	7s	29ms	343ms
Chained Enc.Inverted Intex	—	—	—	—	—	—	—	—
Zheng et al. 2020	50s	6s	371ms	335ms	45s	7s	216ms	514ms

Tabella 5.8: Tempi ottenuti dal PoC

Osservando i tempi ottenuti dai test privi di indice si può notare come rispecchino abbastanza fedelmente ciò che ci si sarebbe potuto aspettare guardando le complessità.

Si può notare come l'aggiunta di un indice però non abbia favorito tutti i modelli i modelli allo stesso modo, questo perché ovviamente l'aggiunta di un indice (MongoDB) su un altro "indice" (facente parte del modello crittografico) non avrà lo stesso miglioramento di performance di un indice applicato direttamente sulla collezione (o tabella) avendo quest'ultima un numero nettamente superiore di record.

Si può anche osservare come il modello di Zheng et al. in casi molto al limite come quello dei generi dei film risulti quasi equivalente, come performance, alle sue alternative .

Capitolo 6

Architettura del PoC

6.1 Obiettivo del PoC

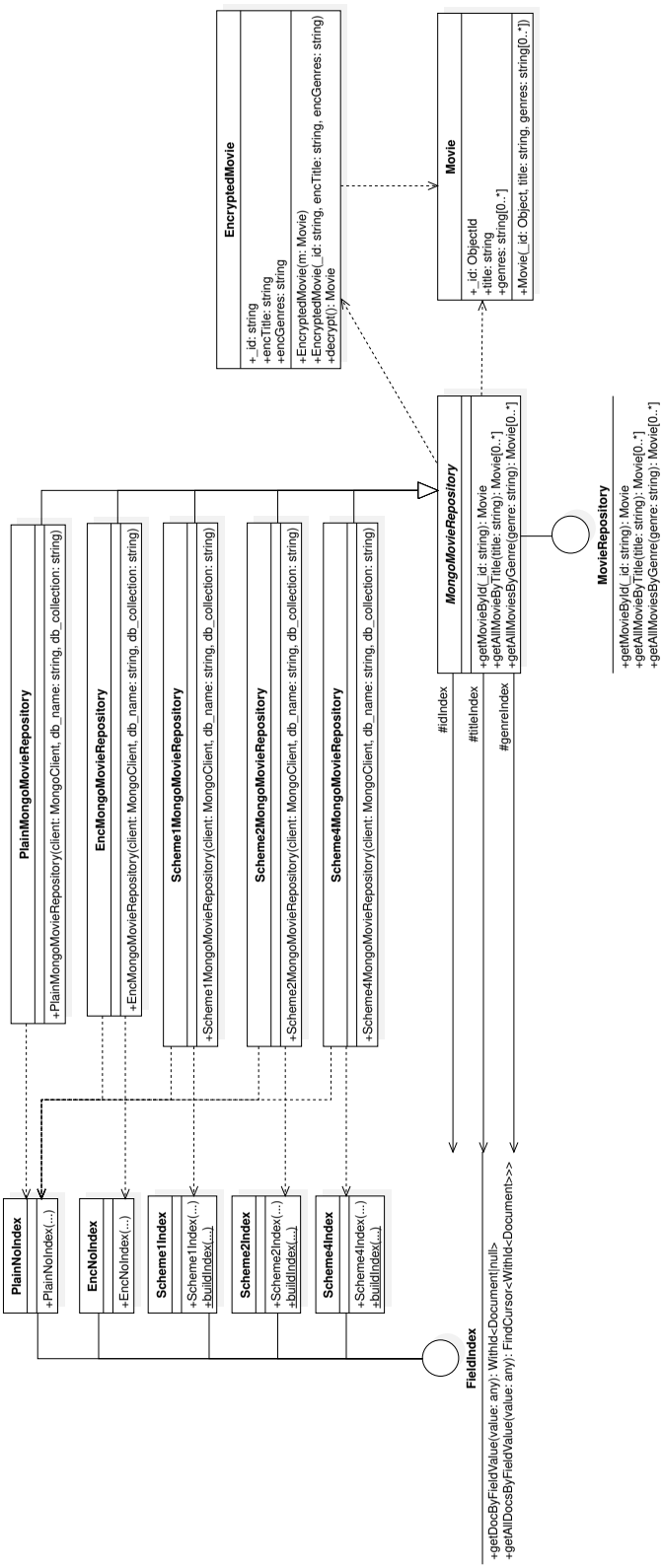
L'obiettivo del PoC è quello di comparare le performance dei diversi modelli specialmente nelle operazioni di ricerca. Per questo motivo, dati i tempi abbastanza stretti, ho deciso di limitare le funzionalità del PoC alla sola creazione dell'indice ed alla ricerca al suo interno, lasciando ad un'eventuale futura implementazione le parti di modifica dell'indice dovute ad inserimenti, modifiche o eliminazioni.

6.2 Decisioni generali

Per la creazione degli oggetti responsabili di comunicare con il database ho scelto di utilizzare il pattern *repository* considerato che il prodotto Monokee ne fa uso.

Inoltre, per una migliore riutilizzabilità, ho deciso di creare delle classi che abbiano la responsabilità della gestione dei singoli campi, al loro interno ho implementato i metodi per la ricerca e l'eventuale costruzione dell'indice, in modo che non sia la repository a doverlo implementare molteplici volte.

Ad esempio, la ricerca utilizzando uno specifico modello crittografico sarebbe dovuta essere implementata una volta per ogni campo, nel nostro caso sia `title` che `genres`, ma grazie alla classe di quello specifico indice mi è stato sufficiente implementarlo una unica volta.



Per riuscire ad aumentare la leggibilità del diagramma, dato che non è a scopo documentale, ho deciso di omettere alcuni dettagli come le firme dei metodi eccessivamente lunghe e alcune classi di supporto che ho utilizzato.

Figura 6.1: Diagramma delle classi del PoC

6.3 FieldIndex

Come anticipato ho creato l'interfaccia *FieldIndex* le cui implementazioni saranno responsabili della ricerca su un unico campo del database. Più precisamente ogni implementazione effettuerà la ricerca utilizzando uno specifico modello crittografico.

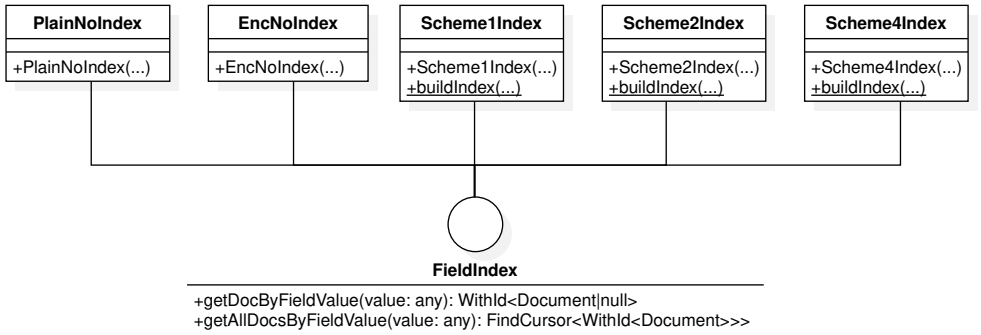


Figura 6.2: Dettaglio dello schema delle classi relativo a FieldIndex

Ho implementato 5 diverse tipologie di ricerca:

- ***PlainNoIndex:*** Permette la ricerca su un campo non cifrato, utilizza una normale ricerca ed eventualmente se è presente sfrutta l'indice creato da mongo;
- ***EncNoIndex:*** Permette la ricerca su un campo cifrato, non utilizzando un indice l'unica modalità possibile di ricerca richiede di decifrare tutti i campi ad ogni ricerca;
- ***Scheme1Index:*** Permette la ricerca su un campo cifrato utilizzando l'indice descritto nella sezione 'Aggiunta di un campo contenente l'hash' [5.1.1];
- ***Scheme2Index:*** Permette la ricerca su un campo cifrato utilizzando l'indice descritto nella sezione 'Indice invertito criptato' [5.1.2];
- ***Scheme4Index:*** Permette la ricerca su un campo cifrato utilizzando l'indice descritto nella sezione 'Schema da Zheng et al. (2020)' [5.1.4].

Le prime due implementazioni non utilizzano un indice crittografico, ma ho deciso di implementarle in modo da avere un confronto prestazionale dei modelli crittografici, logicamente più la velocità di un indice si avvicina a quella di *PlainNoIndex* più esso sarà efficiente, se invece si avvicina a quella di *EncNoIndex* possiamo reputarlo come inefficace.

Ho deciso di non implementare il modello descritto nella sezione 'Indice invertito criptato e concatenato' [5.1.3] in quanto molto simile al suo schema originale e inoltre data la natura del modello sarebbero stati necessari dei test molto più specifici che avrebbero dovuto tenere conto di quanto la varianza di p impattasse sull'algoritmo stesso.

I modelli che utilizzano al loro interno un indice hanno la necessità di crearlo e proprio per questo motivo ho codificato anche il metodo *buildIndex()* in aggiunta ai metodi dell'interfaccia FieldIndex.

6.4 Repository

Il pattern repository permette l'astrazione dei dati rispetto alla loro sorgente rendendo possibile un cambio del database o del suo metodo di accesso senza la necessità di modificare codice esterno alla repository stessa.

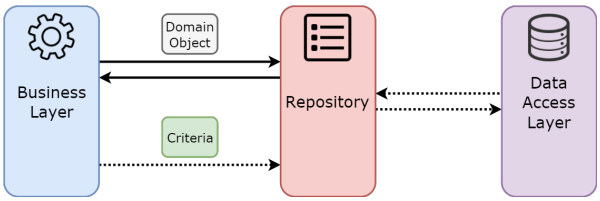


Figura 6.3: Pattern Repository
Fonte: https://codingsight.com/entity-framework-antipattern-repository/

Infatti, all'interno del PoC ho creato l'interfaccia *MovieRepository* che specifica le firme dei vari metodi, l'interfaccia viene implementata dalla classe (astratta) *MongoMovieRepository* che utilizza un database MongoDB come fonte dei dati. Qualora si dovesse cambiare il tipo della fonte dei dati passando ad un database SQL o a delle API basterebbe creare una nuova implementazione dell'interfaccia *MovieRepository*.

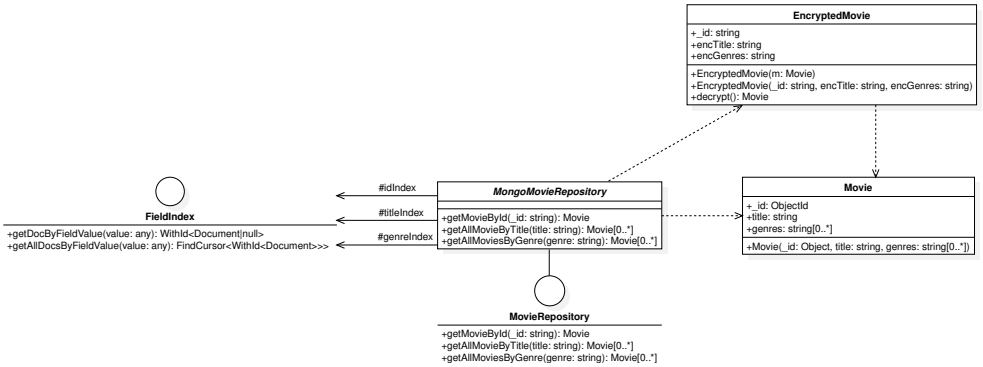


Figura 6.4: Dettaglio dello schema delle classi relativo a MovieRepository

Più precisamente, all'interno del PoC, *MongoMovieRepository* è una classe astratta che tramite i *FieldIndex* dei vari campi (`_id`, `title` e `genres`) esegue le ricerche ed estrae dai cursori mongo ritornanti i dati e li inserisce negli oggetti del modello *Movie*. I *FieldIndex* vengono istanziati nelle classi figlie concrete di *MongoMovieRepository* che a seconda del modello crittografico che rappresentano useranno le implementazioni di *FieldIndex* più adatte.

In totale ho implementato 5 diverse repository:

- ***PlainMovieRepository:*** Repository che utilizza *PlainNoIndex* per la ricerca su `title` e `genres`;
- ***Scheme0MovieRepository:*** Repository che utilizza *EncNoIndex* per la ricerca su `title` e `genres`;
- ***Scheme1MovieRepository:*** Repository che utilizza *Scheme1Index* per la ricerca su `title` e `genres`;

- ***Scheme2MovieRepository:*** Repository che utilizza *Scheme2Index* per la ricerca su `title` e `genres`;
- ***Scheme4MovieRepository:*** Repository che utilizza *Scheme4Index* per la ricerca su `title` e `genres`.

Si può notare come per le classi implementate l'accoppiamento tra repository e fieldIndex è praticamente 1:1 con l'eccezione di *PlainNoIndex* che è utilizzato da tutte le repository dato che il campo id è sempre in chiaro. Questo, però, non significa che non si possano creare delle repository "ibride" che al loro interno usino diversi tipi di indici.

6.5 Aspetti migliorabili

Dal punto di vista architetturale questa soluzione non è perfetta ed in questa sezione elencherò quelli che per me sono degli aspetti migliorabili.

6.5.1 Accoppiamento eccessivo tra FieldIndex e la libreria MongoDB

Nell'architettura attuale le classi FieldIndex sono responsabili degli algoritmi per la ricerca, ma nonostante questo al loro interno c'è anche un grande accoppiamento verso le classi della libreria di MongoDB. Questo purtroppo rende inutilizzabili queste classi qualora ci fosse un cambio di sorgente dei dati. Riuscire però a dividere queste due responsabilità della classe (algoritmo di ricerca e comunicazione col DB) non è così semplice dato che un algoritmo complesso come quello sviluppato da Zheng et al. al suo interno richiede molteplici interazioni con il database.

Un altro ostacolo alla risoluzione di questo problema è che per come ho progettato il PoC sarebbe necessario cambiare anche il tipo di interazione tra *MongoMovieRepository* e i *FieldIndex*, perché attualmente i metodi di ricerca ritornano degli oggetti della libreria mongo.

6.5.2 Utilizzo di un Builder per la creazione delle MongoMovieRepository

Un'alternativa alla creazione di una nuova classe per ogni nuova combinazione di FieldIndex sarebbe quella di utilizzare il design pattern del Builder per la creazione di un oggetto *MongoMovieRepository*. Questo metodo potrebbe risultare particolarmente comodo in ottica di test prestazionali, come quelli che sto svolgendo, dove non c'è una unica repository tutto sommato stabile nel tempo, ma è conveniente poter creare in maniera veloce e dinamica nuovi tipi di repository.

In ottica, invece, di un'applicazione stabile in cui gli indici utilizzati non cambiano, e se dovessero cambiare non si arriverebbe al caso di avere bisogno due implementazioni diverse allo stesso momento, si potrebbe valutare di decidere e creare i FieldIndex direttamente in *MongoMovieRepository* rendendola così concreta.

Capitolo 7

Conclusioni

7.1 Risultati raggiunti

Al termine dell'approfondito processo di ricerca applicativa i prodotti finali ottenuti al termine del tirocinio in questo tirocinio sono stati il PoC e la documentazione ad esso annessa. Questo perché ho impiegato la maggior parte del tempo a studiare prima l'attuale stato dell'arte delle tecnologie crittografiche ed in particolare della Searchable Encryption e poi ad imparare tecnologie a me nuove come *MongoDB* e *Typescript*.

Questi risultati sono in linea con quanto discusso inizialmente con l'azienda. Data la natura del tirocinio non è stata stilata una lista di requisiti in quanto gli obbiettivi dei periodi sono stati definiti progressivamente.

Il lavoro che ho svolto durante il tirocinio è stato apprezzato dall'azienda, che mi ha chiesto di continuare a lavorare per loro come sviluppatore all'interno della startup *Monokee*.

7.2 Conoscenze acquisite

Nello svolgimento del tirocinio ho dovuto fare un grande uso della letteratura scientifica e grazie a questo ho imparato come sono strutturati i paper e soprattutto come orientarsi tra di essi specialmente grazie a *Google scholar*.

Inoltre, per l'implementazione del PoC ho dovuto apprendere il linguaggio *Typescript* che non avevo mai utilizzato e che mi sarà essenziale per il proseguimento dell'esperienza lavorativa. Ho avuto anche l'occasione di approfondire i database NoSQL come *MongoDB* che ho studiato sempre per lo sviluppo del PoC.

7.3 Valutazione personale

Durante l'evento STAGE-IT questa proposta di tirocinio mi ha colpito particolarmente perché mi dava la possibilità di fare esperienza in un'attività di ricerca, permettendo di mettermi alla prova in questo ambito. La ricerca mi interessava ed ho potuto sfruttare l'occasione di questo tirocinio per capire se il mio interesse fosse reale.

Al termine di questa esperienza mi reputo soddisfatto sia delle nuove conoscenze acquisite ma anche di aver potuto sperimentare un accenno di quello che significa lavorare nel campo della ricerca. Ho trovato molto stimolante l'ambito della crittografia anche se ho sentito la mancanza di conoscenze algebriche più approfondite. Questo

tirocinio ha confermato il mio interesse verso una formazione, specialmente teorica, più approfondita che spero di trovare continuando il mio percorso di studi attraverso la Laurea Magistrale in Computer Science.

Acronimi e abbreviazioni

ABE Attribute-based Encryption.

CCA Choosen-ciphertext Attack.

CKA Choosen-keyword Attack.

CPA Choosen-plaintext Attack.

FHE Fully Homomorphic Encryption.

HE Homomorphic Encryption.

IBE Identity-based Encryption.

OPE Order Preserving Encryption.

ORE Order Revealing Encryption.

PRF Pseudorandom Function.

PRG Pseudorandom Generator.

SE Searchable Encryption.

SSE Symetric Searchable Encryption.

Glossario

decryption oracle Il dencryption oracle è una scatola nera, spesso usata nelle dimostrazioni di sicurezza degli algoritmi crittografici, che decifra un ciphertext in input e ne restituisce il relativo plaintext.

encryption oracle L'encryption oracle è una scatola nera, spesso usata nelle dimostrazioni di sicurezza degli algoritmi crittografici, che cifra un plaintext in input e ne restituisce il relativo ciphertext. Viene utilizzato solamente nei casi di crittografia simmetrica dato che negli altri casi di crittografia asimmettrica utilizzando una chiave pubblica la cifratura è già "pubblica".

HMAC L'HMAC è un *keyed-hash message authentication code* e viene utilizzato per l'autenticazione di messaggi. Serve a produrre un hash che garantisce sia l'integrità che l'autenticazione del messaggio inviato. Una caratteristica dell'HMAC è che non è legato a nessuna funzione di hashing, va quindi scelta al momento dell'utilizzo.

ind-CCA Un algoritmo crittografico soddisfa la proprietà ind-CCA (Indistinguishability under Chosen-ciphertext Attack) quando è resistente ad attacchi di tipo CCA, che sono attacchi che mirano ad indebolire la sicurezza attraverso l'accesso al *decryption oracle*.

ind-CKA Un algoritmo crittografico soddisfa la proprietà ind-CKA (Indistinguishability under Chosen-keyword Attack) quando è resistente ad attacchi di tipo CKA, che sono attacchi che mirano ad ottenere informazioni sulle *trapdoor* avendo a disposizione l'*encryption oracle* o la chiave pubblica.

ind-CPA Un algoritmo crittografico soddisfa la proprietà ind-CPA (Indistinguishability under Chosen-plaintext Attack) quando è resistente ad attacchi di tipo CPA, che sono attacchi che mirano ad indebolire la sicurezza attraverso l'accesso all'*encryption oracle* o alla chiave pubblica.

KDF La Key Derivation Function è un algoritmo che permette la creazione di più chiavi crittografiche partendo da una chiave principale..

SHA256 Lo SHA256 è un funzione crittografica di hashing della famiglia SHA (Secure Hashing Algorithm) e produce un digest di 256 bit.

Bibliografia

[1] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[2] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[3] Zhiniang Peng. Danger of using fully homomorphic encryption: A look at microsoft seal. *arXiv preprint arXiv:1906.07127*, 2019.

[4] Eu-Jin Goh. Secure indexes. *Cryptology ePrint Archive*, 2003.

[5] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, page 79–88, New York, NY, USA, 2006. Association for Computing Machinery.

[6] Seny Kamara, Charalampos Papamanthou, and Tom Roeder. Dynamic searchable symmetric encryption. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 965–976, 2012.

[7] Ke Zhang, Jiahuan Long, Xiaofen Wang, Hong-Ning Dai, Kaitai Liang, and Muhammad Imran. Lightweight searchable encryption protocol for industrial internet of things. *IEEE Transactions on Industrial Informatics*, 17(6):4248–4259, 2020.

[8] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public key encryption with keyword search. In *International conference on the theory and applications of cryptographic techniques*, pages 506–522. Springer, 2004.

[9] Raphaël Bost, Brice Minaud, and Olga Ohrimenko. Forward and backward private searchable encryption from constrained cryptographic primitives. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1465–1482, 2017.

[10] Liangliang Xiao and I-Ling Yen. Security analysis for order preserving encryption schemes. In *2012 46th annual conference on information sciences and systems (CISS)*, pages 1–6. IEEE, 2012.

[11] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 563–574, 2004.

[12] Dan Boneh, Kevin Lewi, Mariana Raykova, Amit Sahai, Mark Zhandry, and Joe Zimmerman. Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 563–594. Springer, 2015.

[13] Amit Sahai and Brent Waters. Fuzzy identity-based encryption. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 457–473. Springer, 2005.

[14] Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In *International conference on the theory and applications of cryptographic techniques*, pages 127–144. Springer, 1998.

[15] Yandong Zheng, Rongxing Lu, Jun Shao, Fan Yin, and Hui Zhu. Achieving practical symmetric searchable encryption with search pattern privacy over cloud. *IEEE Transactions on Services Computing*, 15(3):1358–1370, 2022.

[16] Scott Arciszewski. How to search on securely encrypted database fields. https://www.sitepoint.com/how-to-search-on-securely-encrypted-database-fields/, Jun 2017.