Private Set Intersection Analysis

Confronto prestazionale di differenti soluzioni

LL-2021-PPA-PSI - v1.1

24/11/2021

Start-up di

Sapienza
Università di Roma

Start-up di

Tor Vergata
Università degli Studi di Roma

Il presente documento è stato redatto secondo le informazioni e i dati in possesso di Lockless S.r.l., alla luce delle informazioni note, della situazione in essere e di quanto poteva essere ragionevolmente supposto, al momento della sua stesura. Si precisa che, in conformità con l'incarico ricevuto, tali informazioni sono state assunte dai materiali redattori acriticamente, ovvero senza svolgere alcun controllo in merito alla correttezza, completezza e validazione dei dati e informazioni ricevute.

Indice

Elenco delle figure

Elenco delle tabelle

Informazione sul documento e sugli autori

Società/Cliente	**PagoPA S.p.A.**
Titolo documento	Private Set Intersection Analysis
Codice documento	LL-2021-PPA-PSI
Creato il	22/10/2021
Ultimo aggiornamento	24/11/2021
Autore	Lockless S.r.l.
Versione	1.1

Stato del documento

Rev. nr.	Data	Dettagli
1.0	22/10/2021	Prima stesura
1.1	24/11/2021	Revisione del documento a valle della discussione con PagoPA.

1 Scopo del documento

In questo documento tecnico vengono presentati i risultati legati all'analisi teorica ed empirica di algoritmi per la realizzazione di una libreria di Private Set Intersection (PSI). Questa attività rientra all'interno del contratto PagoPA S.p.A / Lockless S.R.L., CIG: 86571076B1, identificata come "fase 1".

L'obiettivo di questo documento è quello di presentare l'attuale stato dell'arte degli algoritmi di PSI, con l'obiettivo di mostrare quali sono i benefici e le criticità legati sia alla necessità di processare dataset di dimensioni non minimali, sia ai requisiti di sicurezza di tale soluzione. La presentazione viene focalizzata sugli specifici use case di PagoPA, al fine di identificare delle scelte tecnologiche che possano essere pratiche per gli obiettivi aziendali. Nello specifico sono stati presi in considerazione gli use case relativi alla validazione delle transazioni nell'ambito del servizio Cashback, così come al calcolo di intersezioni tra codici fiscali per futuri possibili servizi di PagoPA.

Viene inoltre fornita una valutazione sperimentale di implementazioni prototipali di alcuni algoritmi identificati come di interesse, utilizzando metriche di valutazione ad-hoc e dataset sintetici ragionevoli rispetto agli use case di PagoPA. Tale valutazione sperimentale ha l'obiettivo di permettere una definizione congiunta da parte di PagoPA e Lockless degli obiettivi necessari al raggiungimento della "fase 2" del contratto. A tale scopo, le implementazioni prototipali utilizzate in questa fase del contratto potranno venire estese ed evolute, ad esempio al fine del rilascio di un pacchetto open source.

L'organizzazione di questo documento è la seguente. In sezione 2 viene presentato lo specifico problema di interesse, evidenziando anche quali sono i requisiti funzionali e non funzionali di massima di PagoPA rispetto all'identificazione di una soluzione tecnologica. In sezione 3 vengono discussi alcuni concetti fondamentali per la comprensione delle tecniche algoritmiche allo stato dell'arte che sono presentate in sezione 4. I risultati della valutazione sperimentale sono riportati in sezione 5. La sezione 6 discute di alcune implicazioni su sicurezza ed osservabilità delle soluzioni considerate.

2 Problem statement

Private Set Intersection (PSI) è un problema computazionale in cui due (o più) parti sono proprietarie di un insieme privato di elementi e sono interessate a calcolarne l'intersezione, pur senza rivelare alcun elemento (o in generale alcuna informazione aggiuntiva) del proprio dataset al di fuori dell'intersezione stessa. In altri termini, le due parti (un *mittente* ed un *ricevente*, anche chiamati *client* e *server*) posseggono due insiemi di dati, S ed S' rispettivamente, e vogliono calcolarne l'intersezione $I = S \cap S'$ senza rivelare all'altra parte alcuna informazione addizionale su S ed S', eccettuando al limite la dimensione degli insiemi. In alcune applicazioni, solo il ricevente sarà in grado di calcolare I mentre il mittente non sarà in grado di estrarre alcuna informazione. In altre applicazioni, entrambe (o tutte) le parti riescono a calcolare I. Dato l'obiettivo specifico del presente lavoro, in questo documento viene tralasciata questa seconda possibilità, per quanto alcune sue implicazioni relative alla sicurezza verranno trattate in Sezione 6.

In generale, il problema di PSI è perciò una specifica istanza di *secure multi-party computation*, ossia una generica computazione in cui più parti sono interessate a calcolare una generica funzione su input forniti dalle parti coinvolte senza rivelare informazioni riguardo agli input. Nel caso del problema di PSI, la funzione è quella di intersezione tra insiemi e gli input sono i dataset (insiemi) di ciascuna parte.

La PSI assume una particolare rilevanza in scenari in cui la riservatezza degli input forniti da ciascuna parte è fondamentale. Tale criticità può essere legata a molteplici fattori, ad esempio norme che vietano la diffusione in chiaro di specifiche informazioni (ad esempio, utenti iscritti ad un servizio di abbonamento) o, più semplicemente, le parti non intendono fidarsi l'una dell'altra. Il problema ha perciò una natura simmetrica, ossia ciascuna parte intende preservare la privacy dei propri input.

Per progettare un protocollo che sia efficiente oltre che sicuro è necessario definire un modello comportamentale dell'avversario al fine di non sovradimensionare i parametri di sicurezza a discapito delle prestazioni. Nel contesto di questo documento viene considerato un modello di comportamento *honest but curious* (onesto ma curioso), anche chiamato *semi-honest*. In questo modello, un utente semi-honest ha le seguenti caratteristiche:

1. aderisce al protocollo di sicurezza;
2. non altera gli input del protocollo;
3. tenta di estrapolare informazioni private dai dati lecitamente scambiati.

Nel contesto di PSI, il modello semi-honest implica che nessuna delle parti può alterare il proprio dataset, ad esempio rimuovendo elementi dal proprio insieme o aggiungendone di fittizi. Al contrario, ciascuna parte può eseguire qualsiasi tipo di operazione su dati lecitamente scambiati durante la computazione al fine di estrapolare informazioni aggiuntive all'intersezione (ad esempio, gli elementi). Questo modello comportamentale è sufficientemente generico da comprendere anche scenari di *data leakage*, ossia situazioni in cui i dati lecitamente scambiati diventano di dominio pubblico. Infatti un'entità che ottenga i suddetti dati senza aver partecipato al protocollo può attuare i medesimi approcci di una parte semi-honest per ottenere ulteriori informazioni riguardo agli input delle parti coinvolte. Infine, è importante sottolineare come, con queste ipotesi, è impossibile evitare che una delle parti coinvolte riveli tutto il proprio dataset se interamente incluso nell'intersezione, scenario da intendersi come *working-as-intended*, ossia ammissibile piuttosto che indesiderato.

Nel contesto di questo progetto si è interessati a valutare diverse soluzioni al problema di PSI tra sole due parti autenticate fra loro, le quali sono le uniche coinvolte nel protocollo. Si escludono perciò soluzioni di PSI che ricorrono ad una terza parte esterna e fidata, ad esempio per l'approvazione dei dati scambiati a livello di PSI o per il calcolo stesso dell'intersezione. Il problema di PSI viene formalizzato secondo quanto segue:

- Il mittente possiede un insieme di elementi S;
- Il ricevente possiede un insieme di elementi S';
- Il ricevente è interessato a calcolare $S \cap S'$.

Un protocollo per il calcolo di PSI possiede i seguenti requisiti.

Correttezza: il ricevente calcola l'esatta intersezione tra i due insiemi S ed S'

Recipient Privacy: il ricevente non apprende alcuna informazione riguardo elementi di S' non appartenenti all'intersezione, se non la relativa cardinalità, ossia $|S' \setminus (S' \cap S)|$.

Sender Privacy: il mittente non apprende alcuna informazione riguardo elementi di S non appartenenti all'intersezione, se non la relativa cardinalità, ossia $|S \setminus (S' \cap S)|$.

Esistono poi altri requisiti tipicamente definiti come opzionali.

Server unlinkability: date più esecuzioni di PSI, il mittente non è in grado di relazionare gli input forniti dal ricevente nelle diverse esecuzioni (per esempio, il set S' è invariato)

Client unlinkability: date più esecuzioni di PSI, il ricevente non è in grado di relazionare gli input forniti dal mittente nelle diverse esecuzioni (per esempio, il set S è invariato)

3 Concetti preliminari

In questa sezione vengono riportati alcuni concetti teorici di base legati a tecniche crittografiche ed algoritmi che verranno presi in considerazione per il calcolo di PSI, o che non verranno considerati a causa di motivazioni teoriche o prettamente pratiche. Alcune caratteristiche di tali metodologie, già evidenziate in questa sezione, suggeriscono già alcune scelte che verranno discusse successivamente in questo documento.

3.1 Hash

Un *hash* (anche chiamato *message digest*) è il risultato di una funzione unidirezionale f (chiamata *funzione hash*), ossia una funzione che può calcolare un output $h \leftarrow f(m)$ a partire da un dato messaggio m in tempo polinomiale, ma per cui non esiste alcun algoritmo *probabilisticamente polinomiale* che può calcolare una controimmagine di $f(m)$ a meno di una probabilità trascurabile, se m viene scelto in modo casuale.

Per poter considerare una funzione hash crittograficamente sicura deve quindi essere computazionalmente infattibile poter identificare, dato un certo hash $\bar{h}$, un messaggio m tale che $\bar{h} \leftarrow f(m)$. Allo stesso modo deve essere impossibile individuare due messaggi m_1 ed m_2 tali che $f(m_1) = f(m_2)$, ovverosia che generino una collisione nel dominio degli hash.

In generale, tutte le funzioni hash hanno lo stesso comportamento: prendono in input un messaggio m e lo trasformano in un output di lunghezza prefissata. Il modo in cui questa operazione viene svolta determina il grado di sicurezza delle funzioni. Tutte le funzioni hash, comunque, portano con sé un certo concetto di *casualità*: la trasformazione da input m ad output h deve essere tale da sembrare scelta in maniera del tutto casuale. Ciò significa, ad esempio, che:

- se si scelgono 100 messaggi m in maniera del tutto casuale, ciascun bit nei 100 output h deve valere 1 circa la metà delle volte;
- ciascun output h deve avere, con alta probabilità, almeno la metà dei bit impostati ad 1;
- due output h_1 e h_2 devono essere totalmente scorrelati, indipendentemente dalla similarità dei rispettivi input m_1 ed m_2. Ciò implica che se m_1 ed m_2 differiscono unicamente per un solo bit, gli output h_1 ed h_2 devono apparire come valori scelti in maniera del tutto casuale, con circa la metà dei bit differenti.

Questi aspetti implicano che non deve essere possibile, se non mediante il calcolo di $f(m)$, poter predire una qualsiasi porzione dell'output h. Inoltre deve essere vero che, per poter trovare due sottosequenze di bit in h_1 ed h_2 uguali, l'unica possibilità è quella di calcolare $f(m)$ per un gran numero di m differenti scelti casualmente, fintanto che non si identificano due output in cui le sottosequenze corrispondono.

Il numero di bit utilizzati per rappresentare gli hash ha anche un'implicazione sulla sicurezza dello specifico schema di hashing adottato. Infatti, se si utilizzano b bit per rappresentare un hash h, dopo aver generato circa $2^{b/2}$ hash, partendo da input scelti casualmente, è altamente probabile individuare due hash identici. Ad esempio, utilizzando funzioni hash a 64 bit, è necessario processare soltanto 2^{32} messaggi differenti per trovarne due associati allo stesso hash. Analizzare 2^{32} input è un calcolo considerato generalmente fattibile. Per questo motivo, funzioni hash tipiche hanno un hash di grandezza pari ad almeno 128 bit, poiché non si considera fattibile, con la tecnologia moderna, una ricerca tra 2^{64} messaggi. Tuttavia è possibile ridurre il costo computazionale di tale ricerca, rendendo spesso possibile identificare dei conflitti anche con hash a 128 bit. In generale, la comunità considera un algoritmo di hash crittografico come non più sicuro non appena vengono identificati due messaggi m_1 ed m_2 che generano lo stesso output h.

Riassumendo, proprietà importanti per le funzioni hash in ambito crittografico sono le seguenti:

- *resistenza alla controimmagine*: deve essere computazionalmente intrattabile la ricerca di un messaggio che dia un hash uguale ad un dato hash;
- *resistenza alla seconda controimmagine*: deve essere computazionalmente intrattabile la ricerca di un messaggio che dia un hash uguale a quello di un dato messaggio;
- *resistenza alle collisioni*: deve essere computazionalmente intrattabile la ricerca di una coppia di messaggi che diano lo stesso hash.

3.2 Gruppi e campi finiti

Un utile strumento matematico utilizzato per la realizzazione di schemi crittografici è quello dei gruppi e dei campi finiti, che vengono presentati in questa sezione.

Definiamo $\mathbb{Z}_p^*$ come l'insieme di tutti gli interi modulo p che sono coprimi di p (ad esempio $\mathbb{Z}_{10}^* = \{1, 3, 7, 9\}$) dove però p è un numero primo. Dati due elementi $i, j \in \mathbb{Z}_p^*$, si ha che $i \cdot j \in \mathbb{Z}_p^* \quad \forall i, j \in \mathbb{Z}_p^*$. In altre parole, vale il seguente teorema:

Teorema 3.1. $\mathbb{Z}_p^*$ è chiuso rispetto alla moltiplicazione modulo p.

Alcune caratteristiche ovvie di $\mathbb{Z}_p^*$ sono le seguenti:

1. *associatività*: $(ab)c = a(bc)$;
2. esistenza dell'*identità*: esiste almeno un elemento e tale che $ea = ae = a \quad \forall a \in \mathbb{Z}_p^*$. Nel caso di $\mathbb{Z}_p^*$ l'elemento identità è 1;
3. esistenza dell'*inverso*. Per ciascun elemento a esiste un elemento a^{-1} tale che $a^{-1}a = aa^{-1} = e$;
4. *commutatività*: per ogni a, b, $ab = ba$.

Possiamo definire *gruppo* una struttura algebrica $\langle G, \cdot \rangle$, un insieme di elementi G ed un'operazione $\cdot$ che prende due elementi di G e li trasforma in un singolo elemento di G, tale da soddisfare le proprietà 1, 2, 3 del precedente elenco. Nel caso in cui sia soddisfatta anche la proprietà 4, allora parliamo di un *gruppo commutativo*, conosciuto anche come *gruppo abeliano*. La maggior parte dei gruppi utilizzati in crittografia sono gruppi abeliani. In particolare, $\mathbb{Z}_p^*$ è un gruppo in cui l'operazione $\cdot$ è la moltiplicazione modulo p.

Un *sottogruppo* di un gruppo G è un sottoinsieme di G che è un gruppo secondo l'operazione del gruppo G—l'elemento e è sempre un sottogruppo di ciascun gruppo. Le potenze $\{g^n | n \in \mathbb{Z}\}$ di ciascun elemento $g \in G$ sono un sottogruppo chiamato *sottogruppo ciclico* generato da g. Se G è finito, è necessario includere soltanto le potenze non negative di g. Un gruppo G è detto *ciclico* se è sottogruppo ciclico di se stesso, ovvero se esiste $g \in G$ tale che $G = \{g^n | n \in \mathbb{Z}\}$. In questo caso, g viene chiamato *generatore* di G. Un gruppo ciclico può avere più di un generatore.

L'*ordine* di un gruppo $|G|$ è il numero di elementi in quel gruppo. Se G è un gruppo finito ed H è un sottogruppo di G, allora $|H|$ è divisore di $|G|$. L'*ordine di un elemento* è, invece, l'ordine del sottogruppo ciclico che esso genera. Se g ha un ordine finito, allora questo ordine è l'intero positivo λ più piccolo tale che $g^\lambda = e$. Si noti che se $g^k = e$, allora λ è divisore di k.

Consideriamo ora invece due operazioni, $+$ e $\cdot$. Una importante proprietà, comune alla matematica più tradizionale, è quella della *distributività* di $\cdot$ su $+$: per tutti i valori a, b, c, $a \cdot (b + c) = a \cdot b + a \cdot c$.

Possiamo ora definire *campo* una struttura composta da un insieme F e due operatori $+$ e $\cdot$ che soddisfino la proprietà distributiva e per i quali $\langle F, \cdot \rangle$ è un gruppo commutativo con identità 0, mentre $\langle F \setminus \{0\}, \cdot \rangle$ è un gruppo commutativo con identità 1. $\mathbb{Z}_p$, gli interi modulo p con p primo è, ad esempio, un campo. Un sottocampo di un campo F è un sottoinsieme di F che è un campo sotto le operazioni di F. Se E è un sottocampo di F, allora F è un'*estensione di campo* di E.

Se F è un campo finito di q elementi, allora ogni elemento di F è una radice di $x^q - x$. Infatti, $x^q - x = x(x^{q-1} - 1)$, pertanto 0 è chiaramente una radice. Ricordando che $F \setminus \{0\}$ è un gruppo sotto la moltiplicazione di ordine $q - 1$, poiché l'ordine di ciascun elemento del gruppo divide l'ordine del gruppo, ciascun $a \in F \setminus \{0\}$ soddisfa $a^{q-1} = 1$. Pertanto, ciascun elemento diverso da zero è una radice di $x^{q-1} - 1$. Poiché ci sono e elementi in F, e poiché $x^q - x$ ha grado q, si può concludere che $x^q - x = \prod_{a \in F}(x - a)$. Peraltro, $F \setminus \{0\}$ è ciclico. Consideriamo infatti l'ordine di ciascun elemento: sappiamo che esiste un qualche elemento g il cui ordine è il minimo comune multiplo λ di tutti gli ordini. Sappiamo inoltre ciascuno dei $q - 1$ elementi $a \in F \setminus \{0\}$ soddisfa $a^\lambda = 1$, poiché λ è un multiplo dell'ordine di a, e lo stesso vale per le radici di $x^\lambda - 1$. Però, $x^\lambda - 1$ ha grado λ, pertanto devono esserci al più λ radici. Pertanto, $\lambda \geq q - 1$. Ma poiché λ dipende da $q - 1$, allora $\lambda = q - 1$, pertanto $F \setminus \{0\}$ è ciclico con generatore g.

Dato un certo campo finito, risulta quindi evidente che se partiamo da 0 ed incominciamo ad aggiungere ripetutamente 1, prima o poi torneremo al valore 0. Il numero di volte che dobbiamo aggiungere 1 per tornare nuovamente a 0 è detto la *caratteristica* del campo. La proprietà distributiva ci permette di affermare che la caratteristica è un numero primo: infatti, se la caratteristica fosse ab in cui sia a che b fossero più piccoli della caratteristica, allora $ab = \sum_{i=1}^{a} 1 \cdot \sum_{j=1}^{b} 1 = \sum_{i=1}^{ab} 1 = 0$, pertanto $F \setminus \{0\}$ conterrebbe sia a che b ma non ab. La proprietà distributiva ci permette anche di affermare che se F ha caratteristica p, allora per ogni $c \in F$, $\sum_{i=1}^{p} c = (\sum_{i=1}^{p} 1)\, c = 0 \cdot c = 0$.

È possibile determinare la dimensione di un gruppo finito F di dimensione p costruendo una sequenza di elementi $a_i \in F$ tale che ogni elemento aggiunto alla sequenza durante la sua costruzione non possa essere espresso come combinazione lineare $\sum_i c_i a_i$ degli elementi a_i già presenti nella sequenza, con i coefficienti $c_i \in \mathbb{Z}_p$. Se, durante la sua costruzione, la sequenza è composta da n elementi, ci sono p^n possibili combinazioni lineari $\sum_{i<n} c_i a_i$ con $c_i \in \mathbb{Z}_p$. Nessuna coppia di queste può essere uguale, altrimenti potremmo trovare una soluzione in funzione di un a_i precedente per cui i coefficienti sono differenti, il che è contrario alla scelta degli a_i effettuata. Pertanto, $|F| = p^n$.

Ne consegue che per un dato numero primo p e per un intero positivo n esiste un solo campo di ordine $q = p^n$. Si tratta del *campo di spezzamento* (o *campo di riducibilità completa*) di $x^q - x$, ossia la più piccola estensione di campo di $\mathbb{Z}_p$ in cui $z^q - x$ si fattorizza completamente in polinomi di grado 1. Questo campo è anche chiamato *Campo di Galois* di ordine q, scritto $GF(q)$, per quanto ci siano vari modi per rappresentare questo campo. Per poterlo rappresentare in maniera tale da consentire dei calcoli utili per l'applicazione di tecniche crittografiche procediamo come segue.

Si sceglie un numero primo p ed un intero positivo n che determini il campo F di ordine $q = p^n$. Si prende un generatore moltiplicativo g di $F \setminus \{0\}$. Consideriamo ora $g^0, g^1, \ldots, g^{n-1}$: essi devono essere linearmente indipendenti su $\mathbb{Z}_p$, ovverosia non esiste alcuna loro combinazione lineare banale uguale a zero, altrimenti avremmo $g^k = \sum_{i<k} c_i g^i$ per qualche $k < n$. Pertanto, poiché ogni elemento non zero di F è G^m per un qualche valore di m, possiamo esprimere ogni elemento di F come combinazione lineare di $g^0, g^1, \ldots, g^{k-1}$ con coefficienti da $\mathbb{Z}_p$, ma abbiamo a disposizione soltanto p^k combinazioni lineari di questo tipo. Al contrario, poiché $|F| = p^n$, non si possono individuare più di n elementi linearmente indipendenti di F, pertanto $g^n = \sum_{i<n} c_i g^i$ per un qualche insieme unico di coefficienti $c_0, \ldots, c_{n-1}$ determinato dal generatore scelto.

Pertanto, possiamo rappresentare ogni elemento $a \in F$ come una sequenza di n elementi di $\mathbb{Z}_p$, ovverosia i coefficienti a_i di $g^0, g^1, \ldots, g^{n-1}$ per cui $a = \sum_{i<n} a_i g^i$. L'addizione può essere definita come addizione componente per componente, mentre la moltiplicazione può essere trattata come una normale moltiplicazione tra polinomi, a patto che i termini con esponente maggiore di n vengano convertiti utilizzando $g^n = \sum_{i<n} c_i g^i$.

Dobbiamo inoltre essere in grado di calcolare la negazione e l'inverso. La negazione è una semplice negazione componente per componente in $\mathbb{Z}_p$. Il calcolo dell'inverso, invece, è più complesso: occorre utilizzare l'algoritmo di Euclide per il calcolo del massimo comune divisore. In particolare, applicando l'algoritmo di Euclide su $\alpha(x)$ e $\gamma(x)$, otteniamo i polinomi $\mu(x)$ e $\nu(x)$ tali che $\alpha(x)\mu(x) + \gamma(x)\nu(x) = \text{MCD}(\alpha(x), \gamma(x))$. Si noti che, dal momento che $\gamma(g) = 0$, $\alpha(g)\mu(g) = \text{MCD}(\alpha(x), \gamma(x))(g)$. Se il massimo comune divisore di $\alpha(x)$ e $\gamma(x)$ è 1, allora $\mu(x)$ è a^{-1}. Ma se il massimo comune divisore non è 1, allora a non ha alcun inverso. Poiché $F \setminus \{0\}$ è un gruppo sotto la moltiplicazione, ogni valore di a diverso da zero ha un inverso, pertanto nessun $\alpha(x)$ non costante può dividere $\gamma(x)$, ovvero $\gamma(x)$ non ha alcun fattore banale. Un polinomio con nessun fattore non banale è detto *irriducibile*. Pertanto, il campo può essere pensato come dei polinomi su $\mathbb{Z}_p$ modulo il polinomio irriducibile $\gamma(x)$.

È equivalente affermare che, scegliendo un qualsiasi polinomio $\gamma(x)$ di grado n in $\mathbb{Z}_p$, questo produce una rappresentazione di $GF(p^n)$.

Alcuni algoritmi crittografici a chiave simmetrica, come ad esempio l'Advanced Encryption Standard (AES) (conosciuto anche come Rijndael) sfruttano la matematica su campi finiti secondo la costruzione descritta precedentemente.

Il funzionamento schematico di AES è riportato in Figura 1. Si tratta di un algoritmo con stato in cui il testo in chiaro viene sottoposto a un certo numero di *round* di criptazione, in cui vengono applicate ripetutamente alcune operazioni fondamentali. Il numero di round dipende

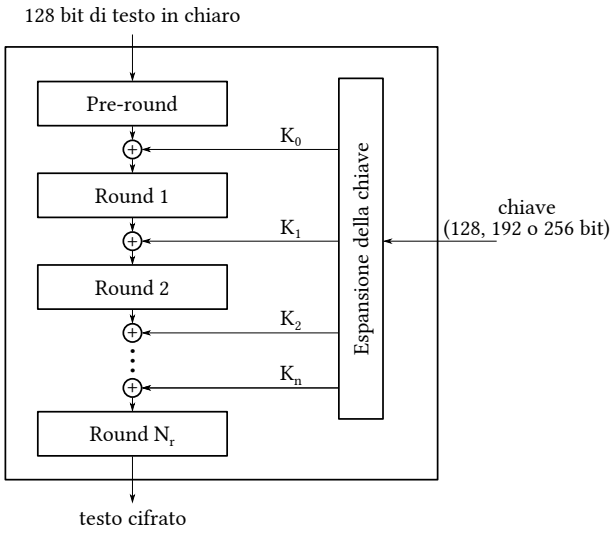


Figura 1: *Struttura a blocchi dell'algoritmo AES*

dalla dimensione della chiave scelta, così da rendere un attacco allo schema crittografico difficile quanto un attacco a forza bruta sulla chiave. Per chiavi a 128 bit si utilizzano tipicamente dieci round, mentre per AES-256 se ne utilizzano quattordici. La chiave fornita è sottoposta ad un'operazione di espansione per far sì che non vengano utilizzati sempre gli stessi bit nei vari round di criptazione. Le operazioni fondamentali con cui sono implementati i round sono:

1. lo xor bit a bit;
2. una sostituzione ottetto per ottetto basata su una tabella chiamata S-box;
3. una rotazione degli ottetti per riga o per colonna (lo stato viene mantenuto in una tabella);
4. un'operazione denominata *MixColumn* che sostituisce una colonna di quattro ottetti con un'altra colonna di quattro ottetti—tale operazione può essere implementata mediante una tabella di sostituzione per velocizzare le operazioni di mescolamento.

Nonostante le operazioni di AES possano sembrare delle sostituzioni definite in forma arbitraria, in realtà queste sono basate sull'aritmetica in $GF(2^8)$. L'aspetto interessante della scelta di questo gruppo è proprio quello che tutti gli elementi del campo possono essere rappresentati da un ottetto. I bit in questo ottetto sono i coefficienti di un polinomio su $\mathbb{Z}_2$ modulo il polinomio irriducibile in $\mathbb{Z}_2$ $m(x) = x^8 + x^4 + x^3 + x + 1$, con il bit meno significativo che è il coefficiente di 1 ed il bit più significativo che è il coefficiente di x^7.

In questa rappresentazione, l'addizione è lo xor bit a bit, poiché la somma modulo due corrisponde all'operazione di xor (il che giustifica l'operazione 1 nell'elenco sopra). Inoltre,

ciascun elemento è la propria negazione. L'inverso moltiplicativo viene tipicamente calcolato facilmente mediante l'utilizzo di una tabella di lookup, composta da 255 elementi grandi un ottetto (lo zero non ha un inverso). Una tabella di lookup per supportare la moltiplicazione avrebbe necessità di 65536 ottetti, ma poiché AES richiede soltanto la moltiplicazione per sei differenti costanti (appartenenti a $GF(2^8)$) è possibile ridurre l'impatto di tale tabella.

In AES vengono anche utilizzati dei polinomi su $GF(2^8)$, calcolati modulo il polinomio x^4+1 in $GF(2^8)$. Tali polinomi sono rappresentati come vettori di quattro ottetti, con il coefficiente di 1 che è il primo ottetto nel vettore di quattro. L'operazione di *MixColumn* corrisponde alla moltiplicazione per il polinomio $3x^3 + x^2 + x + 2$ modulo $x^4 + 1$. Con la rappresentazione dei polinomi descritta precedentemente, la moltiplicazione per x corrisponde ad una rotazione (il che giustifica l'operazione 3 nell'elenco sopra).

Per quanto riguarda l'operazione 2, ovvero la sostituzione basata su S-box, in realtà questa operazione è una forma compatta delle seguenti tre:

- una permutazione in cui ciascun ottetto viene associato al suo inverso moltiplicativo modulo $m(x)$ (eccettuando lo zero che è associato a sé stesso);
- una moltiplicazione per $x^4 + x^3 + x^2 + x + 1$ modulo $x^8 + 1$;
- somma di $x^6 + x^5 + x + 1$.

Per quanto AES utilizzi in maniera elegante le proprietà dei campi di Galois, essendo esso un algoritmo a chiave simmetrica, non verrà considerato nello schema di PSI obiettivo di questo documento, poiché non rispetta i requisiti definiti in sezione 2. Tuttavia, la teoria dei campi e dei gruppi è alla base di tecniche utili agli scopi di questo progetto che verranno descritte successivamente.

3.3 Funzione toziente di Eulero

La funzione toziente di Eulero $\varphi(n)$ è definita come il numero di elementi in $\mathbb{Z}_n$. Ad esempio, poiché come visto precedentemente $\mathbb{Z}_{10} = \{1, 3, 7, 9\}$, allora $\varphi(10) = 4$.

Una proprietà di $\varphi(n)$ importante per la crittografia è la possibilità di calcolare n a partire da $\varphi(n)$. Se n è primo, $\mathbb{Z}_n^* = \{1, 2, \ldots, n-1\}$, pertanto $\varphi(n) = n-1$. Se consideriamo invece $n = p^\alpha$, con p primo e $\alpha > 0$, solo i multipli di p non sono coprimi di p^α, ed ogni p-esimo numero è multiplo di p, pertanto ci sono $p^{\alpha-1}$ elementi minori di p^α. Pertanto, $\varphi(p^\alpha) = p^\alpha - p^{\alpha-1} = (p-1)p^{\alpha-1}$.

Se $n = pq$ e p e q sono coprimi, possiamo calcolare il valore di n a partire da $\varphi(n)$ applicando

il *teorema cinese del resto*:

Teorema 3.2 (Teorema cinese del resto)**.** Dati $z_1, \ldots, z_k$ a due a due coprimi, comunque si scelgano degli interi $a_1, \ldots, a_k$, esiste un intero x soluzione del sistema di congruenze:

$$x \equiv a_i \mod z_i \quad \forall i = 1, \ldots, k. \tag{1}$$

Tale teorema afferma che esiste una corrispondenza uno ad uno tra i numeri $m \in \mathbb{Z}_{pq}$ ed una coppia di numeri $m_p \in \mathbb{Z}_p$ ed $m_q \in \mathbb{Z}_q$ per cui $m_p = m \mod p$ e $m_q = m \mod q$. Se $m \in \mathbb{Z}_{pq}$ è coprimo di pq, allora esistono due interi u e v tali che $um + vpq = 1$. Sostituendo $m = m_p + kp$ si ottiene $um_p + (uk + vq)p = 1$, pertanto m_p è coprimo di p. Per lo stesso motivo, m_q è coprimo di q.

Viceversa, se $m_p \in \mathbb{Z}_p$ è coprimo di p ed $m_q \in \mathbb{Z}_q$ è coprimo di q, allora esistono quattro interi u_p, v_p, u_q, v_q tali che $u_p m_p + v_p p = 1$ e $u_q m_q + v_q q = 1$. Poiché $m_p = m - kp$ per qualche k e poiché $m_q = m - lq$ per qualche l, allora $u_p m + (v_p - u_p k)p = 1$ e $u_q m + (v_q - u_q l)q = 1$. Il loro prodotto è $(u_p u_q m + u_p(v_q - u_q l)q + u_q(v_p - u_p k))m + (v_p - u_p k)(v_q - u_q l)pq = 1)$, ossia m è coprimo di pq.

Esiste pertanto una corrispondenza biunivoca tra i numeri $m \in \mathbb{Z}^*_{pq}$ e coppie di numeri $m_p \in \mathbb{Z}^*_p$ e $m_q \in \mathbb{Z}^*_q$. Pertanto, $\varphi(pq) = \varphi(p)\varphi(q)$.

3.4 Crittografia a chiave asimmetrica

Gli schemi crittografici a chiave asimmetrica (o a chiave pubblica/privata) si basano sulle cosiddette "*funzioni botola*", ossia funzioni matematiche che risultano facili da calcolare in una direzione, ma il cui calcolo dell'inversa (ossia, nella direzione opposta) è invece computazionalmente molto complesso a meno di avere a disposizione, appunto, una botola.

Molte funzioni botola sono basate sulla difficoltà computazionale di calcolare la fattorizzazione di interi molto grandi, o di calcolare i logaritmi discreti. Nel seguito di questa sezione mostreremo alcuni schemi crittografici basati su questo concetto, illustrando come è possibile utilizzare funzioni botola per realizzare sistemi di scambio di informazioni private su canali pubblici. Tali approcci verranno in seguito utilizzati al fine di realizzare degli schemi di PSI di interesse per gli obiettivi prefissati in sezione 2.

3.4.1 Schema crittografico di Diffie-Hellman

Lo schema di scambio di chiavi di Diffie ed Hellman [7], sviluppato in contemporanea con Merkel [20], è uno tra i protocolli a chiave asimmetrica più utilizzati, che risulta essere parte integrante di molti standard di comunicazione privata su canali pubblici. La sua struttura matematica sottostante si basa su un gruppo ciclico $G = \langle g \rangle$, in cui g è un generatore noto. Alice e Bob scelgono due segreti $a, b \in \{1, 2, \ldots, |G|\}$, si scambiano g^a, g^b e stabiliscono così un elemento comune del gruppo $g^{ab} = (g^a)^b = (g^b)^a$. Pertanto, questo schema funziona poiché l'elevamento a potenza è commutativo.

Inoltre è abbastanza sicuro poiché è molto difficile computazionalmente ricavare g^{ab} a partire da g^a e da g^b. Supponiamo infatti che S sia un algoritmo che accetta in input il parametro di sicurezza 1^s, dove $s \in \mathbb{N}$, campiona un gruppo ciclico G di ordine opportunamente grande e un generatore g di G. A seconda della rappresentazione del gruppo, l'ordine del gruppo dovrebbe essere scelto in modo che il problema computazionale di Diffie-Hellman sia intrattabile:

Definizione 3.1 (Problema computazionale di Diffie-Hellman)**.** Sia $(G, g) \leftarrow \mathsf{S}(1^s)$, dove G è un gruppo ciclico e g è il generatore di G. Siano $a, b \leftarrow U(\mathbb{Z}_{|G|})$, dove $U(\cdot)$ rappresenta una funzione che sceglie un elemento in maniera uniformemente casuale. Dati $(g, g^a, g^b) \in G^3$, il problema computazionale di Diffie-Hellman richiede di trovare $y \in G$ tale che $y = g^{ab}$.

L'intrattabilità di tale problema computazionale è spesso però insufficiente: vorremmo infatti che l'avversario non sia in grado di determinare alcuna informazione su g^{ab}, ovvero siamo interessati alla seguente definizione:

Definizione 3.2 (Problema di decidibilità di Diffie-Hellman)**.** Sia G un grupo ciclico e sia g un generatore di G campionato tramite $(G, g) \leftarrow S(1^k)$. Sia $B \leftarrow U(\{0, 1\})$ e siano $a, b, c \leftarrow U(\mathbb{Z}_{|G|})$. Il problema di decidibilità di Diffie-Hellman richiede di determinare B dati:

$$\begin{aligned} (g, g^a, g^b, g^{ab}) &\in G^4 \text{ se } B = 0 \\ (g, g^a, g^b, g^c) &\in G^4 \text{ se } B = 1 \end{aligned} \tag{2}$$

Lo schema originale di Diffie-Hellman utilizza il gruppo $\mathbb{Z}_p^*$, dove p è un numero primo. Tuttavia, il problema del logaritmo discreto su questo gruppo (si veda la sezione 3.5.1) può essere risolto in tempo sub-esponenziale [4]. Per aumentare il livello di sicurezza di questo scambio di chiavi è quindi possibile ricorrere al gruppo $E(F_q)$ dei punti razionali su una curva

ellittica E (si veda la sezione 3.5) definita su un campo finito F_q. In questo modo è stato mostrato che il grado di sicurezza può essere aumentato significativamente [16].

3.4.2 RSA

RSA [2], dal nome dei suoi inventori Ronald Rivest, Adi Shamir and Leonard Adleman, è un algoritmo crittografico a chiave asimmetrica con chiave a lunghezza variabile. Il tradeoff principale dell'algoritmo RSA è legato alla lunghezza della sua chiave: una chiave lunga corrisponde ad un livello di sicurezza più elevato, mentre una chiave corta permette operazioni di cifratura/decifratura più veloci. Ad oggi, una chiave RSA è considerata sicura se è lunga almeno 1024 bit[1].

RSA effettua l'operazione di criptazione lavorando su *blocchi* di testo in chiaro. Anche la dimensione del blocco è variabile, ma deve essere più piccola della dimensione della chiave. La dimensione di un blocco di testo cifrato corrisponderà alla lunghezza della chiave.

Lo schema crittografico RSA si basa sui seguenti tre algoritmi fondamentali:

- **Generazione delle chiavi**: per prima cosa è necessario scegliere due numeri primi sufficientemente grandi p e q e calcolare $n = pq$. I fattori p e q devono rimanere segreti. L'algoritmo KeyGen accetta il parametro n e restituisce una chiave pubblica $\mathsf{pk} = \langle e, n\rangle$, dove e è un numero coprimo di $\varphi(n) = (p-1)(q-1)$—si veda la sezione 3.3—ed una chiave segreta $\mathsf{sk} = \langle d, n\rangle$, dove d è l'inverso moltiplicativo di $e \mod \varphi(n)$.
- **Criptazione**[2]: l'algoritmo Encrypt accetta pk ed un testo in chiaro m e restituisce un testo cifrato $c = m^e \mod n$.
- **Decrittazione**[3]: l'algoritmo Decrypt accetta sk e c e restituisce il testo in chiaro $m = c^d \mod n$.

Tale schema crittografico è corretto poiché d ed e sono scelti in maniera tale che $de = 1 \mod \varphi(n)$, pertanto per ogni x, $x^{de} = x \mod n$. La funzione di criptazione calcola x^e, mentre la funzione di decrittazione calcola $(x^e)^d = x$.

[1] In realtà, a causa di implementazioni carenti di molti algoritmi di generazioni delle chiavi, tale valore può essere considerato anche una sottostima [18]. Una regola empirica per calcolare la dimensione minima per rendere una chiave RSA sicura, considerando l'incremento della capacità computazionale che si osserva negli anni, è $y - 2000 \cdot 32 + 512$, dove y è l'anno corrente.

[2] Si considera qui unicamente la cifratura, tralasciando lo schema di firma.

[3] Stessa considerazione della nota precedente.

La sicurezza di RSA si basa sul fatto che la funzione KeyGen è considerata essere una funzione botola. Infatti, si assume che la fattorizzazione di un numero molto grande è un algoritmo computazionalmente molto costoso. Pertanto, la sicurezza è da associare unicamente alla dimensione della chiave: data una chiave piccola, infatti, è possibile rendere trattabile il processo di fattorizzazione: una volta determinati p e q, conoscendo la chiave pubblica $\langle e, n\rangle$, è possibile calcolare d come inverso moltiplicativo di $e \mod \varphi(n)$, replicando quindi parte dell'algoritmo KeyGen.

3.5 Crittografia a chiave asimmetrica basata su curve ellittiche

Geometricamente un curva ellittica è una curva tale che, presi due punti su di essa, tirando una retta che li attraversa, la retta intersecherà la curva in un unico terzo punto. Questo perché in un campo, se un polinomio di grado 3 ha due radici (ad esempio P e Q) è possibile dividere il polinomio per il prodotto $(x-r)(x-s)$ per ottenere un polinomio di grado 1, dal quale estrarre la terza radice.

In ambito crittografico vengono tipicamente considerate curve ellittiche nella forma di *Weierstrass*, ossia:

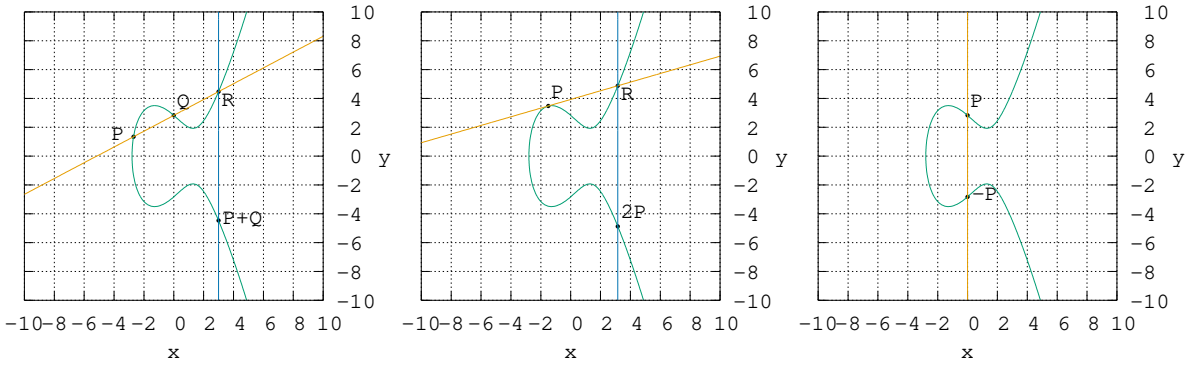
$$\mathcal{C} : y^2 = x^3 + Ax^2 + B \tag{3}$$

dove A e B sono tali che $4A^3 - 27B^2 \neq 0$. Questa condizione garantisce che la curva non ha punti di singolarità.

Dati i punti su una curva ellittica, è possibile definire un'operazione binaria chiamata addizione[4] $+$, tale da trasformare l'insieme dei punti $P \in \mathcal{C}$ e l'operatore $+$ in un gruppo, ossia vale la proprietà associativa ed esiste l'elemento identità ed inverso rispetto all'operatore. Il dettaglio dell'addizione di punti su curve ellittiche è mostrato in Figura 2. I passi, schematizzati in Figura 2a, per calcolare la somma $P + Q$ di due punti P e Q appartenenti ad una curva ellittica sono:

1. individuare la retta r passante per P e Q, dove $P, Q \in \mathcal{C}$;
2. individuare la terza intersezione in un punto $R \in \mathcal{C}$ tra la retta r e la curva $\mathcal{C}$;
3. individuare la retta y_R parallela all'asse y passante per R;

[4] In talune altre formulazioni delle curve ellittiche si utilizza la moltiplicazione al posto dell'addizione. In questo caso, la moltiplicazione per uno scalare viene sostituita dal calcolo dell'esponenziale. In questa rappresentazione alternativa, la rappresentazione diventa simile all'utilizzo di $\mathbb{Z}_p^*$ nel caso dello schema crittografico di Diffie-Hellman. In questa sezione abbiamo optato per l'utilizzo dell'addizione per semplificarne la trattazione.



(a) *Addizione tra due punti* **(b)** *Moltiplicazione scalare di punto* **(c)** *Sottrazione e punto infinito*

Figura 2: *Operazioni elementari tra punti di una curva ellittica.*

4. individuare il secondo punto $S \in \mathcal{C}$ d'intersezione tra y_r e $\mathcal{C}$.

Il punto S così trovato è definito come la somma tra P e Q, ossia $S = P + Q$. Il caso particolare in cui $P = Q$ è gestito considerando R come il punto di intersezione tra la curva ellittica e la retta tangente in P (Figura 2b).

Data una retta parallela all'asse y, questa interseca una curva ellittica in al più due punti P e Q con medesima coordinata x. Tuttavia, applicando lo schema discusso in precedenza per l'addizione, non è possibile individuare una terza intersezione tra la retta e la curva. Di conseguenza, viene aggiunto all'insieme dei punti un elemento all'infinito, denominato $\mathcal{O}$, che rappresenta l'elemento identità del gruppo. A questo punto, Q è l'elemento inverso di P, ossia $-P$ (come mostrato in Figura 2c). L'operatore $+$ così definito sui punti di una curva ellittica e l'elemento neutro $(\mathcal{C} \cup \mathcal{O})$ gode della proprietà commutativa, costituendo quindi un gruppo abeliano.

Date le proprietà geometriche su cui è costruita l'operazione di addizione è possibile identificare le seguenti proprietà elementari e relazioni tra le coordinate di due punti P e Q, e la loro somma $P + Q$:

$$P = (x, y) \Rightarrow -P = (x, -y) \tag{4}$$

$$P = (x, y_p), Q = (x, y_q) \Rightarrow P + Q = \mathcal{O} \tag{5}$$

$$P = (x, 0) \Rightarrow 2P = \mathcal{O} \tag{6}$$

$$\begin{cases} P + Q = \left(\lambda^2 - x_p - x_q, -\lambda^3 + \lambda(x_p + x_q) - \upsilon\right) & \\ \lambda = \frac{y_q - y_p}{x_q - x_p}, \upsilon = \frac{y_p x_q - y_q x_p}{x_q - x_p} & P \neq Q, x_p \neq x_q \\ \lambda = \frac{3x_p^2 + A}{2y_p}, \upsilon = \frac{-x_p^3 + Ax_p + 2B}{2y_p} & P = Q, y_p \neq 0 \end{cases} \tag{7}$$

Al fine di moltiplicare un punto P per un generico scalare k, è possibile ricorrere iterativamente all'uso dell'operatore + per sommare più volte il punto P ad un punto accumulatore R inizializzato a $\mathcal{O}$. Tuttavia, l'operazione di addizione tra punti è dispendiosa, in quanto richiede molteplici operazioni numeriche (moltiplicazione, divisione, potenze, sottrazioni ed addizioni). Una soluzione più efficiente consiste nell'approccio *Double-and-Add*, ossia raddoppia e aggiungi, dimostrato avere un costo logaritmo rispetto allo scalare k.

3.5.1 Il problema del logaritmo discreto ed il suo uso in crittografia

Le proprietà di gruppo e le relazioni algebriche mostrate per l'operatore + continuano a valere per curve ellittiche definite su campi finiti, ossia per curve con $x, y, A, B \in \mathbb{F}_p$. Una curva E così costruita viene indicata come $E(\mathbb{F}_p)$. Dato un punto $P \in E(\mathbb{F}_p)$, esiste un numero $n \in \mathbb{F}_p$ tale che $nP = \mathcal{O}$. n è l'ordine di P. Se n è primo, P è il generatore di un sottogruppo $\langle P \rangle$ con n elementi, ossia

$$\langle P \rangle = \{\mathcal{O}, 1P, 2P, 3P, \ldots, (n-1)P\}$$

Dati P e $Q \in \langle P \rangle$, il problema del logaritmo discreto consiste nell'individuare lo scalare k tale $kP = Q$. Una soluzione [12] a costo $O(n)$ per questo problema sarebbe quella di calcolare tutti i possibili P_i con $i \in [1, n-1]$ finché $P_j = Q$, quindi $j = k$. Tuttavia, il problema può

- Input comune: curva ellittica, relativo punto generatore $\vec{G}$, ordine n di $\langle G \rangle$

1. Alice:
$$a \in [1, n-1] \text{ via PRNG}$$
$$\text{pubblica } \vec{A} = a \cdot \vec{G}$$
2. Bob:
$$b \in [1, n-1] \text{ via PRNG}$$
$$\text{pubblica } \vec{B} = b \cdot \vec{G}$$
3. Bob:
$$\vec{S} = b \cdot \vec{A}$$
4. Alice:
$$\vec{S} = a \cdot \vec{B}$$

Figura 3: *Scambio di chiavi di Diffie-Hellman basato su curve ellittiche.*

essere ricondotto al cercare una collisione, ad esempio generando due liste di elementi:

$$\begin{aligned} L_1 &= [iP] \; \forall i \in [1, n-1] \\ L_2 &= [Q - iP] \; \forall i \in [1, n-1] \end{aligned} \tag{8}$$

finché non si verifica la condizione $iP = Q - jP$. Essendo questo un approccio che sfrutta il *birthday paradox*, è noto che il costo computazionale è $O\left(\sqrt{n}\right)$ e l'algoritmo di Pollard [27] può essere adottato per ridurre lo spazio richiesto. Inoltre, l'algoritmo di Pohlig-Hellman [26] riduce il problema di calcolare il logaritmo discreto per i sottogruppi con ordine n' primo.

Per queste ragioni, i parametri utilizzati per configurare un sistema crittografico a curve ellittiche devono essere accuratamente scelti, ad esempio assicurandosi che il più grande divisore q di n sia un numero primo con un numero sufficiente di bit. Nello specifico, per avere s bit di sicurezza offerti da un protocollo di sicurezza basato sulla difficoltà di calcolare il logaritmo discreto, è necessario che q abbia una taglia tale da far impiegare 2^s passi all'algoritmo di Pollard, quindi $q \approx (2^s)^2$, ossia q ha circa $2s$ bit.

Un esempio di protocollo che ricorre alle curve ellittiche e alla difficoltà di invertire il logaritmo discreto è l'algoritmo di generazione di chiave simmetrica ECDH mostrato in Figura 3.

Le due parti si accordano sui parametri da utilizzare per una specifica curva, un generatore G ed il relativo ordine n. Ognuna delle parti (Alice e Bob nell'esempio in figura) sceglie un numero

random incluso tra 1 e $n-1$ che diventa la propria chiave privata (rispettivamente a e b) e ne pubblica il prodotto con G (A e B), costituendo quindi la chiave pubblica. Infine, ciascuna parte calcola un segreto condiviso S con un prodotto scalare tra la propria chiave privata e la chiave pubblica dell'altra parte.

3.6 Crittografia omomorfica

La crittografia omomorfica permette di effettuare operazioni su dati criptati, senza rendere necessaria la loro decrittazione. Questa proprietà rende questi schemi crittografici utili in un largo ventaglio di applicazioni il cui obiettivo sia quello di preservare la riservatezza. In generale, un sistema di criptazione omomorfica, oltre ai tradizionali algoritmi crittografici per la generazione delle chiavi, per la criptazione e per la decriptazione, si basa sul seguente algoritmo fondamentale:

Definizione 3.3 (Valutazione omomorfica)**.** L'algoritmo $\mathsf{Evaluate}$ accetta la chiave pubblica pk, una funzione f ed un insieme di testi cifrati $c_1, \ldots, c_t$ e restituisce un testo cifrato c_f:

$$c_f \leftarrow \mathsf{Evaluate}(\mathsf{pk}, c_1, \ldots, c_t)$$

Il requisito minimo per uno schema di criptazione omomorfica è la correttezza:

Definizione 3.4 (Crittografia omomorfica corretta)**.** Uno schema di criptazione omomorfica $HE = \{\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{Evaluate}\}$ è corretto per una certa funzione f se è soddisfatta la condizione:

$$\mathsf{Decrypt}(\mathsf{sk}, \mathsf{Evaluate}(\mathsf{pk}, f, c_1, \ldots, c_t)) = f(m_1, \ldots, m_t)$$

La sola definizione 3.4, tuttavia, non può garantire una corretta implementazione di un sistema crittografico omomorfico. Consideriamo infatti lo scenario in cui $\mathsf{Evaluate}$ sia implementato come una funzione identità che restituisce semplicemente i testi cifrati senza alcun processamento: $(f, c_1, \ldots, c_t) \leftarrow \mathsf{Evaluate}(\mathsf{pk}, f, c_1, \ldots, c_t)$. $\mathsf{Decrypt}$ può essere allora implementata come una funzione che accetta in input sk ed $f, c_1, \ldots, c_t$, decritta $c_1, \ldots, c_t$ e restituisce $f(m_1, \ldots, m_t)$. Questo schema soddisfa sicuramente la precedente definizione di correttezza, ma è di fatto inutile per gli scopi di riservatezza che si prefigge la crittografia omomorfica.

Un modo per risolvere questo problema è quello di definire un limite superiore alla lunghezza del testo cifrato restituito da $\mathsf{Evaluate}$:

Definizione 3.5 (Crittografia omomorfica compatta)**.** Uno schema di criptazione omomorfica $HE = \{\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{Evaluate}\}$ è compatto se esiste un polinomio s tale che la lunghezza dell'output di $\mathsf{Evaluate}$ è lungo al più $s(\lambda)$ bit.

Questa definizione garantisce che la lunghezza di c_f è indipendente dalla dimensione di $c_1, \ldots, c_t$ o dalla complessità della funzione f.

Definizione 3.6 (Valutazione in forma compatta)**.** Uno schema di criptazione omomorfica $HE = \{\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{Evaluate}\}$ valuta in forma compatta la funzione f se è sia compatto sia corretto rispetto ad f.

A questo punto, possiamo definire la crittografia *completamente omomorfica* come segue:

Definizione 3.7 (Crittografia completamente omomorfica)**.** Uno schema di criptazione omomorfica $HE = \{\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{Evaluate}\}$ è completamente omomorfico se valuta in forma compatta qualsiasi funzione f.

La costruzione di uno schema completamente omomorfico non è banale. Per poter arrivare a questo schema, utile per PSI, è possibile inizialmente rilassare i vincoli sul sistema HE andando a definire uno schema *parzialmente omomorfico*. In questo senso, è utile iniziare dalla definizione, sulla base delle proprietà 3.4–3.6, della crittografia omomorfica additiva e moltiplicativa.

Definizione 3.8 (Crittografia omomorfica additiva)**.** Uno schema di criptazione omomorfica $HE = \{\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{Evaluate}\}$ è omomorfico rispetto all'addizione se esiste un'operazione $\circ$ tale che:

$$\mathsf{Decrypt}(\mathsf{sk}, \mathsf{Evaluate}(\mathsf{pk}, c_1 \circ \ldots \circ c_t)) = m_1 + \ldots + m_t$$

Definizione 3.9 (Crittografia omomorfica moltiplicativa)**.** Uno schema di criptazione omomorfica $HE = \{\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{Evaluate}\}$ è omomorfico rispetto alla moltiplicazione se esiste un'operazione $\bullet$ tale che:

$$\mathsf{Decrypt}(\mathsf{sk}, \mathsf{Evaluate}(\mathsf{pk}, c_1 \bullet \ldots \bullet c_t)) = m_1 \cdot \ldots \cdot m_t$$

Uno schema parzialmente omomorfico è quindi uno schema di criptazione omomorfico che è omomorfico additivo oppure omomorfico moltiplicativo. Il primo schema di criptazione parzialmente omomorfico è RSA [2], che è omomorfico rispetto alla moltiplicazione. Dati due testi

cifrati $c_1 = m_1^e \mod N$ e $c_2 + m_2^e \mod N$, è possibile ottenere un testo cifrato $c \leftarrow c_1 \cdot c_2$. Tuttavia, la versione deterministica di RSA—intesa nel senso che, per un dato messaggio c'è un solo testo cifrato—non è *semanticamente sicura*. La sua variante probabilistica, invece, non è più omomorfica.

Nel 1984, Goldwasser e Micali [11] hanno presentato il primo schema di criptazione probabilistico che ha anche proprietà omomorfiche rispetto all'addizione:

- KeyGen(λ): il loro algoritmo genera due numeri primi p e w. Siano $x = pq$ ed $y = -x$. La tupla $\langle x, y\rangle$ è la chiave pubblica pk, mentre p è la chiave privata sk.
- Encrypt(pk, m): dato un messaggio m ed un valore casuale r, per ciascun bit m_i di m, $c_i = y^{m_i} r^2 \mod x$.
- Decrypt(sk, c_i): per ciascun bit c_i di c_f:

$$m_i = \begin{cases} 1 & \text{se } c_i \text{ è un residuo quadratico} \mod x \\ 0 & \text{se} c_i \text{ è un non residuo quadratico} \mod x \end{cases} \tag{9}$$

- Evaluate(pk, $c_1, \ldots, c_t$): per ciascun bit c_i e c_j di due testi cifrati, $c_i \cdot c_j = (y^{m_i} r_i^2) \cdot (y^{m_j} r_j^2) = y^{(m_i + m_j)\cdot (r_i r_j)^2}$, che è la forma criptata di $m_i + m_j$.

Successivamente, nel 1985, ElGamal ha proposto [8] uno schema di criptazione basato sulla complessità della risoluzione dei logaritmi discreti—si veda Sezione 3.5.1. In questo caso, si tratta di uno schema di crittografia omomorfica moltiplicativa:

- KeyGen(λ): questa funzione restituisce un generatore g di un gruppo ciclico $\mathbb{G}$ di ordine q ed un intero $h = g^x$ dove x è un valore casuale in $\mathbb{Z}_q$. La tupla $\langle \mathbb{G}, q, g, h\rangle$ è la chiave pubblica pk, mentre la tupla $\langle \mathbb{G}, q, g, x\rangle$ è la chiave privata sk.
- Encrypt(pk, m): per effettuare la criptazione di m, questa funzione sceglie un numero casuale y e restituisce il testo cifrato $\langle c_1, c_2\rangle$ dove $c_1 = g^y \mod q$ e $c_2 = m \cdot h^y \mod q$.
- Decrypt(sk, c_i): dato un testo cifrato $c = \langle c_1, c_2\rangle$, questa funzione restituisce $\frac{c_2}{c_1^x} \mod q = \frac{m \cdot g^{xy}}{g^{xy}} \mod q = m$.
- Evaluate(pk, $c_1, \ldots, c_t$): dati due testi cifrati $\langle c_{i1}, c_{i2}\rangle$ e $\langle c_{j1}, c_{j2}\rangle$ che sono la versione crittografata di m_i ed m_j utilizzando due numeri casuali y_i e y_j, allora $\langle c_{i1} c_{j1}, c_{i2} c_{j2}\rangle = \langle g^{y_i} g^{y_j}, (m_i \cdot h^{y_i})(m_j \cdot h^{y_j})\rangle$, che è la forma criptata di $m_i m_j$.

Sulla falsariga di questi schemi crittografici, altre forme di crittografia parzialmente omomorfica sono stati presentati in letteratura. Per rendere la crittografia omomorfica appetibile,

ovverosia per permettere l'utilizzo di funzioni f più complesse all'interno di Evaluate, occorre estendere il concetto di crittografia parzialmente omomorfica rendendo possibile eseguire operazioni che sfruttino sia la moltiplicazione che l'addizione.

In tal senso, è possibile estendere lo schema appena discusso per creare un sistema crittografico "*in qualche modo*" omomorfico. Con questo termine, ci si riferisce alla possibilità per tale schema di supportare contemporaneamente sia la moltiplicazione che l'addizione, limitatamente però alla valutazione di polinomi di grado basso su dati criptati. Utilizzando esclusivamente l'aritmetica modulare elementare, Van Dijk *et al.* [31] uno schema di criptazione in qualche modo omomorfico sufficientemente semplice, che permette di operare sugli interi.

In questo schema, al parametro λ precedentemente discusso vengono aggiunti i seguenti parametri:

- η: è la lunghezza in bit della chiave segreta;
- γ: è la lunghezza in bit degli interi nella chiave pubblica;
- ρ: è la lunghezza in bit del rumore.

Lo schema crittografico funziona come segue:

- $\mathsf{KeyGen}(\lambda)$: la chiave segreta sk è un intero dispari p rappresentato con η bit scelto nell'intervallo $[2^{\eta-1}, 2^{\eta})$. La chiave pubblica pk è $p \cdot q$, dove $q \in [0, 2^{\gamma}/p)$.
- $\mathsf{Encrypt}(\mathsf{pk}, m)$: restituisce $c \leftarrow m + p \cdot q + 2r$, dove $r \in (-2^p, 2^p)$ è il rumore.
- $\mathsf{Decrypt}(\mathsf{sk}, c_i)$: restituisce $m_f \leftarrow (c_f \mod p) \mod 2$
- $\mathsf{Evaluate}(\mathsf{pk}, c_1, \ldots, c_t)$: data una qualsiasi funzione f con t testi cifrati, applica f ai testi cifrati e restituisce c_f.

Questo schema è omomorfico sia rispetto all'addizione che alla moltiplicazione. Rispetto all'addizione, infatti, otteniamo che $c_1 + c_2 = (m_1 + m_2) + p(q_1 + q_2) + 2(r_1 + r_2)$, che è il testo cifrato di $m_1 + m_2$, ma in cui il rumore è raddoppiato. Allo stesso modo, rispetto alla moltiplicazione, otteniamo che $c_1 c_2 = m_1 m_1 + p(q_1 c_2 + q_2 c_1 - p q_1 q_2) + 2(2 r_1 r_2 + r_1 m_2 + r_2 m_1)$, che è il testo cifrato di $m_1 m_2$, in cui però il rumore viene aumentato in forma quadratica.

Se f è un polinomio di grado troppo elevato, il rumore crescerà fino a rendere il testo cifrato risultante dall'operazione indecifrabile. Per rendere quindi uno schema crittografico in qualche modo omomorfico utilizzabile senza restrizioni su f, diventa necessario trovare un meccanismo per ridurre il rumore. Tale meccanismo, proposto da Gentry [10], consente di trasformare un sistema crittografico in qualche modo omomorfico in un sistema crittografico completamente omomorfico. Tale trasformazione si basa sulle seguenti definizioni.

Definizione 3.10 (Crittografia omomorfica "progredibile")**.** Uno schema di criptazione omomorfica $HE = \{\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{Evaluate}\}$ è "progredibile" se è in grado di valutare in forma omomorfica le sue funzioni di decrittazione.

Definizione 3.11 (Crittografia completamente omomorfica livellata)**.** Uno schema di criptazione completamente omomorfica livellata è uno schema $HE = \{\mathsf{KeyGen}, \mathsf{Encrypt}, \mathsf{Decrypt}, \mathsf{Evaluate}\}$ in cui KeyGen richiede un input k oltre a λ ed lo schema risultante è omomorfico per tutte le funzioni f, indipendentemente dalla loro complessità. Il limite $s(\lambda)$ sulla lunghezza del testo cifrato deve rimanere indipendente da k.

Gentry [10] ha mostrato come ogni schema di crittazione in qualche modo omomorfico e progredibile può essere trasformato in un sistema di crittazione completamente omomorfico livellato. L'algoritmo KeyGen genera inizialmente una sequenza di chiavi pubbliche $\langle \mathsf{pk}_1, \ldots, \mathsf{pk}_{k+1} \rangle$ ed una sequenza di chiavi segrete $\langle \mathsf{sk}_1, \ldots, \mathsf{sk}_{k+1} \rangle$ anziché una sola coppia di chiavi. Successivamente, KeyGen genera una sequenza di chiavi segrete criptate $\langle \overline{\mathsf{sk}}_1, \ldots, \overline{\mathsf{sk}}_{k+1} \rangle$, dove $\overline{\mathsf{sk}}_i \leftarrow \mathsf{Encrypt}(\mathsf{pk}_{i+1}, \mathsf{sk}_i)$. Dato un testo cifrato $c_1 \leftarrow \mathsf{Encrypt}(\mathsf{pk}_1, m)$, possiamo ottenere in maniera analoga $\bar{c}_1 \leftarrow \mathsf{Encrypt}(\mathsf{pk}_2, c_1)$. A questo punto, la funzione f utilizzata all'interno di $\mathsf{Evaluate}$ può essere (grazie alla definizione 3.10) la funzione $\mathsf{Decrypt}$ stessa, che accetterà come parametri in input $\overline{\mathsf{sk}}_1$ e $\bar{c}_1$, permettendo di calcolare c_2 come valore criptato di m utilizzando pk_2, ovverosia:

$$c_2 \leftarrow \mathsf{Evaluate}(\mathsf{pk}_2, \mathsf{Decrypt}, \overline{\mathsf{sk}}_1, \bar{c}_1) \tag{10}$$

Sfruttando questo fatto, diventa possibile valutare una funzione f su un testo cifrato senza incrementare il rumore. In particolare, f viene trattata come un circuito, composto da "porte" organizzate in livelli: ciascuna porta viene convertita in un "circuito di decrittazione", che per prima cosa decifra il testo cifrato e solo successivamente applica la funzione della porta originale. Ad esempio, se la porta originale è una add, il circuito di decrittazione equivalente verrà chiamato D_{add}, viceversa se la porta originale esegue un'operazione di moltiplicazione, il circuito equivalente sarà chiamato D_{mul}. Una porta al livello $i+1$ (ad esempio una add) avrà come input una chiave segreta criptata $\overline{\mathsf{sk}}_i$ ed un insieme di testi cifrati su cui effettuare operazioni a quel livello (ad esempio, $\bar{c}_{i1}$ e $\bar{c}_{i2}$). Abbiamo quindi:

$$c_{i+1} \leftarrow \mathsf{Evaluate}(\mathsf{pk}_{i+1}, D_{\mathsf{add}}, \overline{\mathsf{sk}}_i, \bar{c}_{i1}, \bar{c}_{i2}) \tag{11}$$

Ogni volta che la funzione della porta viene calcolata sui testi cifrati, il rumore verrà ricalcolato da capo. In questo modo, f può essere calcolata senza incremento di rumore.

4 Stato dell'arte degli algoritmi di PSI

In questa sezione vengono presentate le soluzioni al problema di PSI disponibili in letteratura aderenti alle ipotesi discusse in Sezione 2. Nello specifico, vengono discusse soluzioni che calcolano l'intersezione tra due dataset e non richiedono l'interazione con una terza parte fidata. Gli algoritmi di PSI possono essere classificati in accordo con le tecniche utilizzate per preservare la privacy degli input. In letteratura sono state individuate soluzioni basate su funzioni hash, crittografia asimmetrica, oblivious transfer e crittografia omomorfica, presentate rispettivamente nelle Sezioni refse:hashpsi, 4.2, 4.3 e 4.4.

4.1 Hash-based PSI

L'algoritmo di PSI più semplice e performante disponibile in letteratura, denominato Hash-PSI, è basato sull'utilizzo di una funzione hash h' secondo lo schema proposto in Figura 4. Il server S calcola $h(y_i)$ per ciascun elemento $y_i \in Y$ ed invia l'insieme Y_h di elementi così calcolato al client C. Quest'ultimo calcola l'insieme X_h, tale che ogni suo elemento $h(x_i)$ è l'hash dell'elemento $x_i \in X$, e poi calcola l'intersezione $X_h \cap Y_h$. Infine, il client, mantenendo l'associazione tra un elemento $x_i \in X$ e il relativo hash $h(x_i)$, calcola $X \cap Y$.

- Input comuni: funzione hash h
- Input client: $X = \{x_1, \ldots, x_n\}$
- Input server: $Y = \{y_1, \ldots, y_m\}$

1. Server:
$$Y_h = \{y' \mid \forall y \in Y, y' = h(y)\}$$
2. Server $\rightarrow$ Client: Y_h
3. Client:
$$X_h = \{x'_i \mid \forall x_i \in X, x'_i = h(x_i)\}$$

OUTPUT: $Y_h \cap X_h$

Figura 4: *Algoritmo di Hash-PSI*

Il suddetto algoritmo, schematizzato in Figura 4, fornisce:

- correttezza: in quanto due hash sono uguali se e solo se i relativi input sono uguali, stanti i presupposti discussi in Sezione 3.1;
- client privacy: il client non invia alcuna informazione al server.

Tuttavia, Hash-PSI non offre server privacy in quanto l'invio al client di Y_h espone quest'ultimo a:

- attacchi basati su rainbow table;
- attacchi basati su dizionario;
- attacchi di forza bruta.

Sebbene gli attacchi basati su rainbow table siano prevenibili ricorrendo all'uso di un sale scelto dal server, questo è un valore di dominio pubblico, ossia noto al client esattamente come la funzione hash utilizzata. Di conseguenza, l'utilizzo di un sale non tutela da attacchi di tipo dizionario o forza bruta. Infatti, il dominio degli elementi è noto e di conseguenza il client, o una qualsiasi entità che ha accesso ai dati inviati dal server al client, può sfruttare la conoscenza del dominio per esplorare l'intero spazio degli elementi (o una sua parte) per identificare l'intero dataset Y del server (o una sua parte). In altre parole, la server privacy è intrinsecamente legata all'entropia del dominio preso in considerazione, ossia all'effettiva randomicità e diversità degli elementi, che sono proprietà caratteristiche del dominio. Nel caso in cui l'entropia del dominio sia bassa, un client semi-honest può calcolare gli hash dell'intero dominio (o una sua parte) e confrontarlo con quanto lecitamente ricevuto dal server. Al contrario, se l'entropia del dominio è sufficientemente elevata, esplorarne l'interezza o una parte rilevante diventa un problema computazionalmente non trattabile. Tuttavia, un client semi-honest ha sempre la possibilità di verificare l'appartenenza o meno all'insieme Y di uno specifico elemento scelto a sua discrezione. In conclusione, l'Hash-PSI ha comunque la sua valenza in scenari in cui la server privacy non sia rilevante o in cui l'entropia del dominio sia elevata e la costruzione artificiale di un singolo elemento appartenente al dominio sia non banale (il client non può distinguere se l'elemento generato non appartiene all'insieme o è un elemento non appartenente al dominio).

4.2 PSI a crittografia asimmetrica

La limitazione principale di Hash-PSI è quindi quella di non fornire server privacy. Questo perché le funzioni hash lasciano trapelare comunque alcune informazioni caratteristiche del dominio. Per superare tale limitazione, si ricorre a schemi di cifratura. In questo modo, l'entropia del dominio viene mascherata dall'entropia caratteristica del dominio delle chiavi di cifratura. Tipicamente, schemi a chiave simmetrica non sono utilizzati in algoritmi di PSI in quanto la chiave segreta è nota ad entrambe le parti. Tuttavia, il suo utilizzo ha una sua valenza nel fornire un livello aggiuntivo di server privacy in scenari di data leakage, ma comunque inefficace verso un client semi-honest. Al contrario, schemi basati su chiave asimmetrica non presentano questa

criticità in quanto ricavare la chiave privata conoscendo la chiave pubblica è un problema computazionalmente oneroso. Questa proprietà ha però un costo, infatti cifrari a chiave asimmetrica hanno performance minori rispetto a cifrari a chiave simmetrica.

In letteratura, sono noti almeno due algoritmi di PSI basati su schemi crittografici a chiavi asimmetriche, ossia Diffie-Hellman PSI [19] (DH-PSI) e Blind-Signature PSI [6] (BS-PSI), discussi rispettivamente in Sezione 4.2.1 e 4.2.2. Questi due algoritmi sono stati inizialmente progettati utilizzando un sistema crittografico basato sull'algebra dei campi finiti, ma possono essere implementati anche tramite l'uso di crittografia a curve ellittiche (EC), fornendo due implementazioni alternative: ECDH-PSI e ECBS-PSI.

4.2.1 Diffie-Hellman-based PSI

L'algoritmo DH-PSI [19], mostrato in Figura 5, utilizza gli stessi principi dell'algoritmo Diffie Hellman per la generazione di chiavi simmetriche. Nello specifico le due parti si accordano sui parametri di sicurezza, ossia il modulo N da utilizzare per la operazioni crittografiche. A questo punto, ciascuna parte S e C identifica una propria chiave privata s e c secondo lo schema Diffie Hellman e cifra ogni elemento del proprio dataset, ottenendo rispettivamente due insiemi, definiti come Y_s e X_c, in cui ciascun elemento è cifrato. Questi due insiemi vengono poi scambiati tra le parti, le quali eseguono una operazione di cifratura con la propria chiave privata sull'insieme ricevuto. Di conseguenza, il server computa X_{cs} e il client Y_{sc}. Infine, il server invia al client l'insieme X_{cs} e il client calcola l'intersezione come $Y_{sc} \cap X_{cs}$. È rilevante sottolineare quanto, in questo algoritmo, sia fondamentale che l'ordine con cui vengono effettuate le operazioni di cifratura tramite chiave privata non sia importante. In altre parole, $k_{cs} = k_{sc}$ dove k_{ij} è l'elemento k cifrato prima con la chiave privata i e poi con la chiave privata j.

L'algoritmo DH-PSI fornisce:

- correttezza: un elemento k appartiene all'intersezione se $k = y_{sc} \in Y_{sc}$ e $k = x_{cs} \in X_{cs}$ se e solo se $x = y$;
- client privacy e server privacy: le parti si scambiano reciprocamente i propri dataset criptati con la propria chiave privata, che non sono decifrabili dalla controparte fintantoché lo schema di cifratura è ritenuto sicuro e l'hash non invertibile, ossia la soluzione al logaritmo discreto sia ritenuto un problema non trattabile.

Al contrario il presente schema non fornisce client e server unlinkability, in quanto una delle due parti può quantificare le variazioni nel dataset dell'altra parte osservando i dati ricevuti in

comunicazioni lecite su più esecuzioni dell'algoritmo di DH-PSI.

In Figura 6, viene riportata una possibile implementazione basata su crittografia a curve ellittiche, denominata ECDH-PSI.

4.2.2 Blind-signature-based PSI

Implementazioni puramente matematiche di alcuni schemi crittografici, ad esempio RSA, sono soggetti ad attacchi noti come blind signature. In questo tipo di attacco, l'attaccante ha la possibilità di richiedere la decrittazione di un testo cifrato con chiave pubblica. Sfruttando questa opportunità, l'attaccante può manipolare l'input al fine di ottenere la cifratura tramite chiave privata di un messaggio in chiaro a piacere, senza che l'altra parte possa rendersene conto. A tal scopo, vengono utilizzate proprietà matematiche specifiche dello schema crittografico utilizzato.

Si consideri il caso specifico di RSA. Il server S ha una chiave privata d, una chiave pubblica e e un parametro di sicurezza N. Il server espone un servizio per decifrare su richiesta un dato messaggio m crittato (m^e) con la chiave pubblica e, ossia calcolare $m^{ed} = m$ per conto del client. In questo scenario, un attaccante può ottenere la cifratura con chiave privata di un qualsiasi messaggio senza che il server ne sia consapevole. Si supponga che l'attaccante vuole ottenere il messaggio m cifrato con la chiave privata d, ossia m^d. A tal scopo, prepara un messaggio $n = m \cdot r^e \mod N$, costruito come il prodotto tra m e un messaggio casuale r crittato con la chiave pubblica e (ossia r^e). Il client richiede la decrittazione di n e il server calcola $n^d \mod N$ ed invia il risultato al client. A questo punto, l'attaccante ha ottenuto n^d, ossia, il messaggio n decriptato con la chiave privata d, ma, conoscendo la reale natura di n, può ottenere m^d moltiplicando n^d per l'inverso moltiplicativo di r. In particolare:

$$
\begin{aligned}
n^d \cdot r^{-1} &\mod N \\
(m \cdot r^e)^d \cdot r^{-1} &\mod N \\
m^d \cdot r \cdot r^{-1} &\mod N \\
m^d &\mod N
\end{aligned}
$$

Sebbene un'implementazione triviale di RSA è vulnerabile ad un attacco di tipo blind signature, questa può rappresentare una proprietà gradita in specifici scenari in cui sia richiesto

- Input comune: modulo N, funzione hash h
- Input client: $X = \{x_1, \ldots, x_n\}$, chiave privata c
- Input server: $Y = \{y_1, \ldots, y_m\}$, chiave privata s

1. Client:
$$X^c = \{x' \mid (\forall x \in X, x' = (h(x))^c \mod N\}$$
2. Client $\rightarrow$ Server: X^c
3. Server:
$$X^{cs} = \{x'' \mid \forall x' \in X^c, x'' = x'^s \mod N\}$$
$$Y^s = \{y' \mid \forall y \in Y, y' = (h(y))^c \mod N\}$$
4. Server $\rightarrow$ Client: Y^s,$X^c s$
5. Client:
$$Y^{sc} = \{y'' \mid \forall y \in Y^s, y'' = y'^c \mod N\}$$

OUTPUT:$Y^{sc} \cap X^{cs}$

Figura 5: *Algoritmo di DH-PSI*

- Input comune: curva ellittica e relativo punto generatore $\vec{G}$
- Input client: $X = \{x_1, \ldots, x_n\}$, chiave privata c, chiave pubblica $\vec{C} = \vec{G} \cdot c$
- Input server: $Y = \{y_1, \ldots, y_m\}$, chiave privata s, chiave pubblica $\vec{S} = \vec{G} \cdot s$

1. Client:
$$X^c = \{\vec{x'} \mid (\forall x \in X, \vec{x'} = \vec{C} \cdot x\}$$
2. Client $\rightarrow$ Server: X^c
3. Server:
$$X^{cs} = \{\vec{x''} \mid \forall x \in X^c, \vec{x''} = \vec{x'} \cdot s\}$$
$$Y^s = \{\vec{y'} \mid \forall y \in Y, \vec{y'} = \vec{S} \cdot c\}$$
4. Server $\rightarrow$ Client: Y^s,$X^c s$
5. Client:
$$Y^{sc} = \{\vec{y''} \mid \forall y \in Y^s, \vec{y''} = \vec{y'} \cdot c\}$$

OUTPUT:$Y^{sc} \cap X^{cs}$

Figura 6: *Algoritmo di ECDH-PSI, dove "·" è l'operatore di moltiplicazione scalare di un punto $\vec{p}$ su una curva ellittica.*

- Input comuni: funzione hash h, full-domain hash H
- Input client: $X = \{x_1, \ldots, x_n\}$,
- Input server: $Y = \{y_1, \ldots, y_m\}$, chiave privata d, chiave pubblica e, modulo N

1. Client:
$$R = \{r_1 \ldots r_n\} \text{ via PRNG}$$
$$X^c = \{x'_i \mid (\forall x_i \in X, x'_i = H(x_i) \cdot r_i^e \mod N\}$$
2. Client $\rightarrow$ Server: X^c
3. Server:
$$X^d = \{x''_i \mid \forall x'_i \in X^c, x''_i = x_i'^d \mod N\}$$
$$Y_h^d = \{y' \mid \forall y \in Y, y' = h(H(y)^d) \mod N\}$$
4. Server $\rightarrow$ Client: Y^s,X^d
5. Client:
$$X_h^d = \{x'''_i \mid \forall x''_i \in X^{cs}, x'''_i = h(x''_i \cdot r_i^{-1}) \mod N\}$$
$$\textbf{OUTPUT: } Y_h^d \cap X_h^d$$

Figura 7: *Algoritmo di BS-PSI*

firmare/crittografare una certa informazione senza che il firmatario venga a conoscenza del contenuto di tale dato. L'esempio di PSI ricade infatti in questa categoria. In particolare, nel BS-PSI [6] (anche chiamato RSA-PSI) il client può richiedere la blind signature x^d di tutti i suoi elementi $x \in X$ ottenendo un insieme chiamato X^d. Inoltre, il server invia l'insieme Y_h^d contenente tutti i suoi elementi $y \in Y$ crittografati con la chiave privata d e su cui è stata applicata una funzione hash h. In particolare, per ogni $y \in Y$ il server invia $h(y^d)$. A questo punto il client applica la medesima funzione hash h per ogni blind signature in X^d ottenendo l'insieme X_h^d e calcola l'intersezione come $X_h^d \cap Y_h^d$. Per semplicità si è assunto che gli elementi di X e Y siano il risultato di una funzione hash h'.

L'algoritmo BS-PSI, mostrato in dettaglio in Figura 7, fornisce:

- correttezza: un elemento k appartiene a $X_h^d \cap Y_h^d$ se e solo se esiste un $h(x^d) \in X_h^d$ e un $h(y^d) \in Y_h^d$ tale che $h(x^d) = h(y^d)$, predicato vero se e solo se $x = y$;
- client privacy: il server non conosce i numeri casuali utilizzati per la blind signature;
- server privacy: da un lato, il client non è in grado di invertire la funzione hash h per poi decrittare con la chiave pubblica un qualsiasi y^d al fine di ottenere y; d'altro canto non può generare autonomamente elementi cifrati con la chiave privata d.

Inoltre, l'approccio basato su blind signature e RSA ha il vantaggio di avere un costo ridotto per il client rispetto al DH-PSI, in quanto la chiave pubblica RSA ha tipicamente un valore assoluto minore rispetto alla chiave privata DH, riducendo il costo delle operazioni di esponenziazione. Al contrario, le operazioni lato server sono più costose in quanto le chiavi private di RSA sono

- Input comune: curva ellittica e relativo punto generatore $\vec{G}$, funzione hash h
- Input client: $X = \{x_1, \ldots, x_n\}$,
- Input server: $Y = \{y_1, \ldots, y_m\}$, chiave privata d, punto pubblico $\vec{D} = \vec{G} \cdot d$

1. Client:
$$R = \{r_1 \ldots r_n\} \text{ via PRNG}$$
$$X^c = \{\vec{x_i'} \mid (\forall x_i \in X, \vec{x_i'} = \vec{G} \cdot (h(x_i) + r_i)\}$$
2. Client $\rightarrow$ Server: X^c
3. Server:
$$X^{dr} = \{x'' \mid \forall x \in X^c, \vec{x''} = \vec{x'} \cdot d\}$$
$$Y^d = \{\vec{y'} \mid \forall y \in Y, \vec{y'} = \vec{D} \cdot h(y)\}$$
4. Server $\rightarrow$ Client: Y^s,X^{cs}
5. Client:
$$X^d = \{\vec{x'''} \mid \forall x \in X^{dr}, \vec{x_i'''} = \vec{x_i''} - \vec{D} \cdot r_i\}$$

OUTPUT:$Y^d \cap X^d$

Figura 8: *Algoritmo di ECBS-PSI, dove "·" è l'operatore di moltiplicazione scalare di un punto $\vec{p}$ su una curva ellittica.*

tipicamente più lunghe rispetto a quelle di DH.

Siccome lo schema di blind signature può essere riproposto anche per schemi di crittografia basati su curve ellittiche, è ragionevole assumere che esista un'algoritmo tipo BS-PSI a curve ellittiche. Tuttavia, questo algoritmo non è stato ad oggi individuato in letteratura dagli autori di questo documento, i quali propongono un possibile schema in Figura 8. Il vantaggio atteso è che, rispetto a ECDH-PSI, l'algoritmo ECBS-PSI abbia un costo invariato per il server e minore per il client in quanto, nello schema proposto, è possibile ricorrere ad operazioni di addizioni tra punti piuttosto che ricorrere a moltiplicazioni scalari, tipicamente più costose delle prime. Per costruire tale soluzione si è utilizzata la corrispondenza tra le operazioni di moltiplicazione e esponenziazione in uno schema RSA e le operazioni di addizione e moltiplicazione in uno schema a curve ellittiche. Inoltre, uno schema ECBS-PSI potrebbe essere preferibile ad un BS-PSI non solo per la dimensione ridotta delle chiavi, ma anche perché non è necessario calcolare l'hash degli elementi cifrati. Infatti, in un sistema crittografico a curve ellittiche la chiave pubblica non può essere utilizzata per decrittare.

Per dimostrare che l'algoritmo ECBS-PSI è corretto è sufficiente mostrare che se esiste un elemento k appartenente all'intersezione $Y^d \cap X^d$, allora $\vec{k} = \vec{D} \cdot h(y)$ per qualche $y \in Y$ e $\vec{k} = \vec{x_i''}$ per qualche $\vec{x_i'''} \in X^d$. Di conseguenza, mostrando che il predicato $\vec{D} \cdot h(y) = \vec{x_i'''}$ è vero se e solo

se $y = x_i$ si è dimostrata la correttezza dell'algoritmo ECBS-PSI proposto. A tal scopo, sono sufficienti pochi semplici passaggi:

$$\begin{aligned}
\vec{D} \cdot h(y) &= \vec{x'''} \\
\vec{D} \cdot h(y) &= \vec{x''} - \vec{D} \cdot r_i \\
\vec{D} \cdot h(y) &= \vec{x'} \cdot d - \vec{D} \cdot r_i \\
\vec{D} \cdot h(y) &= \vec{G} \cdot (x_i + r_i) \cdot d - \vec{D} \cdot r_i \\
\vec{D} \cdot h(y) &= \vec{D} \cdot (x_i + r_i) - \vec{D} \cdot r_i \\
h(y) &= h(x_i) \\
y &= x_i
\end{aligned}$$

Infine, è possibile mostrare che l'algoritmo ECBS-PSI preserva la client e server privacy basandosi sulla difficoltà di invertire la moltiplicazione di un punto per uno scalare (operazione impropriamente chiamata logaritmo discreto su curve ellittiche). Infatti, il client invia elementi nella forma $\vec{x'_i} = \vec{G} \cdot (x_i + r_i)$ e il server invia elementi nella forma $\vec{y'} = D \cdot y$. Di conseguenza entrambe le parti dovrebbero individuare una costante k dati due punti $\vec{A}$ e $\vec{B}$ tali che $\vec{B} = \vec{A} \cdot k$, problema ritenuto intrattabile.

4.3 Oblivious-Transfer PSI

Oblivous Transfer [28, 9], letteralmente trasferimento ignaro, è una generica tecnica di computazione sicura che permette ad un mittente di inviare messaggi ad un destinatario senza ottenere alcuna informazione riguardo i messaggi effettivamente ricevuti dal destinatario. D'altro canto, il destinatario non ricava alcuna informazione riguardo messaggi che non ha ricevuto dal mittente. La primitiva di base, chiamata OT, è tale per cui il mittente invia un messaggio M tale che: il destinatario riesce ad interpretarlo con probabilità 1/2 e la probabilità a posteriori per il mittente che il destinatario abbia ricevuto M è comunque pari a 1/2. Su questi principi vengono costruite una serie di primitive, grazie alle quali è possibile implementare molteplici istanze di *secure computation*, tra cui PSI.

- Input del destinatario: $b \in \{0,1\}$, r via PRNG,
- Input del mittente: m_0, m_1, chiave privata d, chiave pubblica e, modulo N dove $\langle d, e, N\rangle$ sono scelti secondo lo schema RSA.

1. Mittente:
$$\{x_0, x_1\} \text{ via PRNG}$$
2. Mittente $\rightarrow$ Destinatario: x_0, x_1
3. Destinatario $\rightarrow$ Mittente: $X = r^e \cdot x_b \mod N$
4. Mittente $\rightarrow$ Destinatario: $m'_0 = m_0 + X^d - x_0 \mod N, m'_1 = m_1 + X^d - x_1 \mod N$
5. Destinatario:
$$m_b = m'_b - r \mod N$$

OUTPUT: m_b

Figura 9: *Algoritmo di $\binom{2}{1}$-OT basato su RSA*

4.3.1 1-out-of-2 Oblivious Transfer PSI

Un'importante primitiva è *1-out-of-2 OT* [9] (indicata anche con $\binom{2}{1}$-OT) in cui il mittente invia uno tra due possibili messaggi al destinatario, ma quest'ultimo non apprende nulla riguardo al messaggio non ricevuto ed il mittente non è a conoscenza di quale messaggio sia stato ricevuto dal destinatario. Più precisamente, il mittente possiede due messaggi m_0, m_1 e il destinatario decide un bit b. Grazie alla primitiva $\binom{2}{1}$-OT, il destinatario riceve m_b senza apprendere nulla riguardo m_{1-b} ed il mittente non apprende b.

È possibile costruire una primitiva di $\binom{2}{1}$-OT utilizzando schemi di blind signature, ad esempio basati su RSA, come mostrato in Figura 9. In particolare, il mittente possiede una coppia di messaggi m_0 e m_1 ed invia due numeri casuali x_0 ed x_1 logicamente associati a ciascun messaggio. Il destinatario possiede un bit b utilizzato per richiedere uno dei messaggi. Quest'ultimo, richiede la blind signature del numero casuale x_b corrispondente al bit b, ossia x_b. A questo punto il mittente calcola la blind signature e la somma a ciascun messaggio m_i, a cui, inoltre, sottrae il numero casuale x_i. Il risultato m'_i così ottenuto viene inviato al destinatario. Quest'ultimo potrà ottenere il messaggio m_b di interesse semplicemente sottraendo il proprio numero casuale privato utilizzato per la blind signature a m'_b. Grazie allo schema di blind signature, il mittente non apprende nulla su b. D'altro canto, il destinatario riesce ad apprendere il messaggio m_b, ma nulla riguardo m_{1-b}. Infatti, considerando r il numero casuale utilizzato dal destinatario per la

- Input comune: g generatore, modulo N, funzione hash H
- Input del destinatario: $b \in \{0,1\}$, k via PRNG,
- Input del mittente: m_0, m_1, C via PRNG.

1. Destinatario:
$$K_b = g^k$$
$$K_{1-b} = C/g^k$$
2. Destinatario $\rightarrow$ Mittente: K_0
3. Mittente:
$$K_1 = C/K_0$$
$$r_0, r_1 \text{ via PRNG}$$
$$E_0 = m_0 \oplus H(K_0^{r_0})$$
$$E_1 = m_1 \oplus H(K_1^{r_1})$$
4. Mittente $\rightarrow$ Destinatario: $\langle g^{r_0}, E_0\rangle$, $\langle g^{r_1}, E_1\rangle$,
5. Destinatario:
$$m_b = E_b \oplus H(g^{r_b k})$$
OUTPUT: m_b

Figura 10: *Algoritmo di $\binom{2}{1}$-OT basato su DH*

blind signature, quest'ultimo riuscirebbe al più ad ottenere:

$$m'_{1-b} - r$$
$$m_{1-b} + (x_b \cdot r^e)^d - x_{1-b} - r$$
$$m_{1-b} + x_b^d - x_{1-b} - r$$

dove l'elemento x_b^d rimane ignoto fintanto che d rimane segreto al destinatario. Inoltre, è possibile implementare $\binom{2}{1}$-OT basandosi sui gruppi e sulla difficoltà di calcolare il logaritmo discreto, come proposto in [21], secondo lo schema mostrato in Figura 10.

Chiaramente, è possibile utilizzare la primitiva di $\binom{2}{1}$-OT per m coppie di messaggi lunghi l bit. Il risultante algoritmo viene denominato $\binom{2}{1}$-OT$_l^m$ed è sufficiente per calcolare il PSI tra due parti, seppur in modo inefficiente. Utilizzando $\binom{2}{1}$-OT$_l^m$ è possibile definire una computazione sicura per verificare l'uguaglianza tra due elementi (Private Equality Test, o più brevemente PEQT). Un algoritmo per calcolare PEQT tra due elementi x e y rappresentati con L bit consiste nell'utilizzare $\binom{2}{1}$-OT affinché ciascuna parte costruisca privatamente una stringa, rispettivamente s_x ed s_y, e ottenere l'uguaglianza tra x e y verificandola sulle stringhe s_x ed s_y. Nello specifico, per ciascun bit $x[i]$ di x, il mittente inizializza il protocollo $\binom{2}{1}$-OT per trasferire due stringhe casuali k_0^i, k_1^i lunghe L bit al destinatario, il quale utilizza come bit di decisione l'i-

esimo bit di y, ossia $y[i]$. Il destinatario potrà quindi calcolare una stringa s_y come XOR delle stringhe ricevute, ossia $s_y = \bigoplus_{i=0}^{L-1} k^i_{y[i]}$ senza che il mittente conosca alcun bit di y. Similmente, il mittente calcola s_x utilizzando la medesima logica, ossia $s_x = \bigoplus_{i=0}^{L-1} k^i_{x[i]}$. Infine, il mittente invia s_x al destinatario, il quale può verificare l'uguaglianza tra x ed y confrontando s_x ed s_y in chiaro.

Il suddetto algoritmo può essere riutilizzato per permettere al mittente di verificare l'inclusione (Private Set Inclusion) di y nell'insieme X di elementi input del mittente. Infatti, è sufficiente rieseguire il protocollo per ciascuna x appartenente al dataset del mittente. Inoltre, è possibile calcolare l'intersezione reiterando l'algoritmo di calcolo dell'inclusione per ciascun elemento del destinatario, calcolando quindi la PSI.

Tuttavia il calcolo così proposto sarebbe estremamente costoso in termini di comunicazione e numero di operazioni crittografiche. Infatti, la primitiva $\binom{2}{1}$-OT proposta in Figura 9 richiede il trasferimento di $5l$ bit, 2 operazioni crittografiche e 3 generazioni di numeri casuali—l rappresenta lunghezza in bit dei messaggi inviati dal mittente. Di conseguenza, il test di uguaglianza basato su $\binom{2}{1}$-OT^m_l richiede il trasferimento di $5lm$ bit, $2m$ operazioni crittografiche e la generazione di $3m$ numeri casuali—m è la lunghezza della rappresentazione in bit degli elementi ed l un parametro di sicurezza. Per calcolare la PSI servirà quindi la comunicazione di $5lmn^2$ bit, $2mn^2$ operazioni crittografiche e $3mn^2$ generazioni di numeri casuali—n è la cardinalità degli insiemi del destinatario e del destinatario. In altre parole, il suddetto schema ha un costo quadratico nella taglia di un insieme.

4.3.2 OT-extension PSI

Un aspetto cruciale per rendere fruibili le primitive di OT è quello di ridurre i costi (soprattutto di comunicazione) ad esempio riducendo la lunghezza delle stringhe trasferite tra mittente e destinatario. Le *OT extension* assolvono a questo compito adottando uno schema secondo cui inizialmente viene scambiato un numero ridotto di informazioni tramite schemi costosi per ottenere dei segreti condivisi, i quali sono poi utilizzati per trasferire in modo efficiente informazioni di dimensione maggiore. In altre parole, le OT extension applicano un principio simile a quando si utilizza uno schema crittografico a chiave asimmetrica per ottenere una chiave di sessione da usare per cifrare la seguente comunicazione con un meccanismo più efficiente (ad esempio, a chiave simmetrica).

Il compito di una OT extension è quello di ridurre il problema di $\binom{2}{1}$-OT^m_l a quello di

- Input comune: parametro k di sicurezza, funzione hash H, primitiva $\binom{2}{1}$-OT_m^k
- Input del destinatario: vettore di selezione $B = [b_1, \cdots b_m]$ tale che $b_i \in \{0, 1\}$,
- Input del mittente: m coppie $(x_{j,0}, x_{j,1})$, $j \in [1, m]$.

1. Destinatario:
$$\text{random } m \times k \text{ matrix } T \text{ via PRNG}$$
2. Mittente:
$$S \in \{0,1\}^k \text{ via PRNG}$$
3. Entrambi:
$$\text{usano la primitiva } \binom{2}{1}\text{-}\text{OT}_m^k \text{ dove il mittente si comporta da destinatario e viceversa}$$
$$\text{con input } S \text{ e le coppie } (t_i, B \oplus t^i), \text{ dove } t^i \text{ è la } i\text{-esima colonna di } T \text{ e } i \in [1, k]$$
4. Mittente:
$$\text{Riceve una matrice } Q \text{ dove ogni riga } q_j = (r_j \cdot S) \oplus t_j \text{ e } t_j \text{ indica la } j\text{-esima riga di } T$$
$$y_{j,0} = x_{j,0} \oplus H(j, q_j) \text{ per ogni } j \in [1, m]$$
$$y_{j,1} = x_{j,1} \oplus H(j, q_j \oplus S) \text{ per ogni } j \in [1, m]$$
5. Mittente $\rightarrow$ Destinatario: $y_{j,0}, y_{j,1}$ per ogni $j \in [1, m]$,
6. Destinatario:
$$\textbf{OUTPUT: } z_j = y_{j,r_j} \oplus H(j, t_j) \text{ per ogni } j \in [1, m]$$

Figura 11: *Algoritmo di OT-extension*

$\binom{2}{1}$-OT_m^k, con $k < m$ definito come un parametro di sicurezza. Avendo un simile costrutto, o più semplicemente imponendo $k = l$, $\binom{2}{1}$-OT_m^k può essere a suo volta ricondotto all'uso della primitiva $\binom{2}{1}$-OT_k^k.

In Figura 11 viene proposto una dei primi approcci per ottenere una OT extension efficiente, introdotto da Ishai *et al.* [13] e successivamente ottimizzato da Kolesnikov *et al.* [17]. In questo schema, mittente e destinatario utilizzano una primitiva $\binom{2}{1}$-OT_m^k per trasferire dal destinatario al mittente una matrice casuale T con m righe e k colonne utilizzata per offuscare le m decisioni del destinatario. In concreto, il trasferimento della matrice avviene per colonne, ad esempio utilizzando k volte $\binom{2}{1}$-OT per trasferire k stringhe di m bit. A questo punto il mittente può inviare due versioni di ciascun elemento offuscato con delle semplici operazioni di XOR e delle stringhe ottenute dalla righe della matrice ricevuta in precedenza. Il destinatario, utilizzando la sua matrice T e gli m bit rappresentanti le sue decisioni, riesce ad ottenere i relativi messaggi.

Un'ulteriore ottimizzazione volta a ridurre i costi di comunicazione è quella di usare la primitiva *Random OT*, grazie alla quale le coppie $(x_{j,0}, x_{j,1})$ non sono un input del mittente, ma piuttosto un suo output. In altre parole, le coppie vengono decise randomicamente durante il

calcolo della primitiva di OT. Considerando l'algoritmo mostrato in Figura 11, $y_{i,0}$, $y_{i,1}$ e z_j vengono computati utilizzando unicamente le funzioni hash e senza ricorrere ad alcuna operazione di XOR con altri elementi. Di conseguenza, entrambe le parti possono produrre gli output senza che il destinatario invii al destinatario alcun dato dopo il trasferimento della matrice, rendendo perciò superfluo lo step 5 e riducendo i costi di comunicazione.

Utilizzando perciò le OT-extension, l'algoritmo di OT-PSI discusso in Sezione 4.3.1 avrà costi ridotti in termini di comunicazione e computazione viste le ridotte dimensione delle stringhe scambiate tramite primitive OT. Tuttavia, la componente principale di costo è legata alla dimensione dei dataset che hanno un impatto quadratico sui costi di comunicazione. Al fine di ridurre questa componente, Pinka *et al.* [24] introducono due ottimizzazioni principali. La prima consiste nel definire e sfruttare l'efficiente primitiva $\binom{N}{1}$-OT in cui il destinatario sceglie un elemento di una tupla con N elementi anziché una coppia come nel caso di $\binom{2}{1}$-OT. Grazie a questa primitiva, gli autori forniscono un algoritmo per la set inclusion con costi comparabili a quelli di PEQT.

La seconda ottimizzazione coinvolge la fase finale di PSI. Nello specifico, piuttosto che verificare la presenza di un elemento del client nell'intero dataset del server, gli autori propongono l'utilizzo di hashtable per ridurre il numero di confronti da effettuare. Perciò le due parti, si accordano su come costruire una hashtable per ciascuno dei propri dataset ed eseguono una istanza di PSI per ogni *bin* delle hashtable. Il suddetto approccio è stato poi ulteriormente ottimizzato al fine di ridurre ulteriormente il numero di bit per ciascuna comunicazione [23, 15, 25].

4.4 PSI su crittografia omomorfica

Data l'importanza di PSI, negli ultimi anni si è osservato un crescente interesse verso l'utilizzo della crittografia omomorfica per l'implementazione di tali algoritmi (ad esempio, si vedano [5, 14, 3]). Il motivo di questo sforzo è legato alle proprietà fondamentali della crittografia omomorfica, ovvero la possibilità di effettuare operazioni su insiemi di dati criptati senza necessità di decrittarli: tale proprietà, infatti, può consentire una più diretta implementazione di PSI garantendo elevati livelli di confidenzialità. In particolare, molti degli schemi di PSI finora discussi richiedono la trasmissione di una quantità di dati criptati che è proporzionale alla dimensione di entrambi gli insiemi su cui calcolare l'intersezione. Si tratta di un significativo overhead, soprattutto se confrontato con una soluzione banale e non sicura che richiede soltanto l'invio dell'insieme più piccolo.

Input: il ricevente fornisce l'insieme S di dimensione N_S; il mittente fornisce l'insieme S' di dimensione $N_{S'}$. Entrambi gli insiemi sono formati da stringhe di bit di lunghezza σ. I valori N_S, $N_{S'}$, σ sono pubblici.
Output: Il ricevente ottiene $I = S \cap S'$, il mittente ottiene $\perp$.

1. **Setup**: il mittente ed il destinatario si accordano su uno schema di crittografia omomorfica completa. Il ricevente genera una coppia di chiavi, mantenendo privata quella segreta.
2. **Criptazione degli elementi**: il ricevente cripta tutti gli elementi $s_i \in S$ utilizzando lo schema crittografico omomorfico scelato ed invia gli N_S testi cifrati $(c_1, \ldots, c_{N_S})$ al mittente.
3. **Calcolo dell'intersezione**: per ogni c_i, il mittente:
   (a) genera un valore casuale r_i non-zero in chiaro;
   (b) calcola in forma omomorfica:
   $$d_i = r_i \prod_{s' \in S'} (c_i - s)$$
   Il mittente restituisce al ricevente i testi cifrati $d_i, \ldots, d_{N_{S'}}$.
4. **Estrazione della risposta**: il ricevente decifra i testi cifrati $(d_1, \ldots, d_{N_{S'}}$ e calcola:
   $$I = S \cap S' = \{s_i : \mathsf{Decrypt}(\mathsf{sk}, d_i) = 0\}$$

Figura 12: *Algoritmo di Homomorphic-PSI.*

Sfruttando uno schema crittografico completamente omomorfico livellato è possibile costruire un algoritmo per il calcolo di PSI che abbia un overhead di comunicazione proprorzionale alla taglia dell'insieme più piccolo, offrendo allo stesso tempo un costo computazionale contenuto.

Uno schema di base per il calcolo di PSI basato su crittografia omomorfica è riportato in Figura 12. Questo schema è basato sul fatto che il mittente calcola in forma omomorfica il prodotto delle differenze tra gli elementi nel suo insieme e quelli nell'insieme del ricevente, randomizzando il prodotto moltiplicandolo per un valore in chiaro diverso da zero estratto uniformemente a caso, ed invia il risultato al ricevente. Il risultato dell'operazione omomorfica vale zero esattamente quando un elemento dell'insieme del mittente è nell'insieme del ricevente. La riservatezza è garantita dal fatto che il valore di questa operazione è un valore uniformemente casuale nel caso in cui un elemento non sia presente nell'intersezione.

Questo protocollo è però molto inefficiente, poiché ha bisogno che il mittente effettui $O(N_S N_{S'})$ moltiplicazioni e addizioni omomorfiche. Inoltre, la dimensione del circuito necessario ad implementare tale schema crittografico è piuttosto alto. Infine, mittente e destinatario devono

scambiarsi $O(N_S)$ testi cifrati, il che può rendere proibitivo l'utilizzo di questo schema.

Tale schema può comunque essere ottimizzato. Ad esempio, è possibile applicare la tecnica del *batching* [30] allo schema crittografico, che ricorda l'utilizzo di schemi di processamento SIMD (Single Instruction, Multiple Data). Per scelte appropriate del modulo t, esiste un isomorfismo ad anello dallo spazio R_t del testo in chiaro a $\mathbf{Z}_t^n$. Ad esempio, un polinomio costante $a \in R_t$ corrisponde al vettore $(a, \ldots, a) \in \mathbf{Z}_t^n$. Questo isomorfismo trasforma le addizioni e le moltiplicazioni tra polinomi in addizioni e moltiplicazioni componente per componente in ciascuno degli n campi di $\mathbf{Z}_t$.

La tecnica del batching può essere applicata per ridurre il costo computazionale e di comunicazione come segue. Il ricevente raggruppa i suoi elementi in un vettore di lunghezza n, li cripta ed infine invia N_S/n testi cifrati al mittente. Questo, quando processa ciascun testo cifrato c_i, campiona da $\mathbf{Z}_t$ un vettore $r_i = (r_{i1}, \ldots, r_{in}) \in (\mathbf{Z}_t^*)^n$ di elementi uniformemente casuali diversi da zero, calcola in forma omomorfica $d_i = r_i \prod_{s' \in S'}$ e li invia al ricevente. In questo modo, il mittente può operare contemporaneamente su n elementi del ricevente. Nella pratica, n può essere nell'ordine di grandezza delle migliaia, permettendo quindi un significativo miglioramento nell'efficienza del protocollo.

Tuttavia, pur adottando questa ottimizzazione, il mittente deve sempre codificare ciascuno dei suoi elementi separatamente e confrontarli uno ad uno con gli elementi forniti dal ricevente. Sarebbe invece opportuno permettere anche al mittente di sfruttare il batching. A questo scopo è possibile unire il batching ad una tecnica di hashing [24, 23].

Supponiamo che le due parti calcolino un hash degli elementi dei loro insiemi S ed S' utilizzando una qualche funzione hash precedentemente concordata. In questo caso, l'operazione di PSI deve essere effettuata solamente su elementi nello stesso bin, poiché elementi che non sono nello stesso bin sono necessariamente differenti. È importante però che i bin siano sottoposti a padding di una dimensione prefissata, altrimenti i bin avrebbero un numero di elementi sbilanciato, rivelando così informazioni aggiuntive oltre all'intersezione: tutti i bin devono quindi avere lo stesso numero di elementi. Per ottenere miglioramenti di performance significativi è opportuno utilizzare tecniche di hashing quali *cuckoo hashing* [22] o hashing basato sulla permutazione [1].

Per quanto le tecniche di PSI basate su crittografia omomorfica siano particolarmente interessanti e promettenti, ad oggi non è disponibile alcuna libreria open source di schemi crittografici omomofici implementati in linguaggi *memory safe* di larga adozione. Essendo stato questo uno dei requisiti discussi con PagoPA per la realizzazione di una possibile libreria software PSI, non si può introdurre questo schema all'interno delle tecniche da valutare sperimentalmente. Sotto-

lineiamo, tuttavia, che sarebbe interessante poter dedicare del tempo alla realizzazione di una variante memory-safe (o comunque alla validazione di alcune implementazioni parziali o non certificate disponibili online) al fine di poter riconsiderare in un prossimo futuro questa tecnica.

5 Assessment prestazionale

Numerosi fattori possono influenzare le prestazioni delle implementazioni degli algoritmi di PSI. Al fine di descrivere al meglio in che modo differenti elementi possono contribuire alle prestazioni si fa riferimento al seguente modello di esecuzione, in cui:

- ciascuna parte i è proprietaria di un dataset D_i;
- le due parti comunicano tra loro attraverso canali insicuri;
- le due parti sono autenticate tra di loro;
- ciascuna parte i dispone di una funzione f_i utilizzata per oscurare ciascun elemento del proprio dataset. La funzione di oscuramento è tale per cui la sua inversione è computazionalmente dispendiosa (ad esempio, funzione hash o crittografica).

Dato il suddetto modello, si può descrivere il numero massimo di fasi che un algoritmo di PSI deve implementare per permettere al client di ottenere l'intersezione del suo dataset con il dataset del server. In particolare:

1. Ogni parte i applica una propria funzione f_i di oscuramento ad ogni elemento del proprio dataset D_i (per brevità, l'insieme di elementi oscurati viene indicato con $f_i(D_i)$);
2. Le parti si scambiano i relativi $f_i(D_i)$;
3. La parte i applica la propria funzione f'_i ai dati ricevuti dalla parte j, ossia $f_j(D_j)$;
4. Il server invia il risultato della funzione $f'_i(f_j(D_j)$ al client;
5. Il client calcola la Set Intersection (SI) come $f'i(f_j(D_j) \cap f'_j(f_i(D_i)$;
6. Il client traduce la SI in chiaro tramite reverse mapping.

Questo processo di esecuzione ingloba tutti gli algoritmi di PSI finora discussi, considerando che alcune implementazioni possono avere un numero ridotto di fasi. Di conseguenza, gli elementi di costo per una parte di un algoritmo di PSI possono essere caratterizzati in due categorie principali:

- computazione (step 1, 3, 5 e 6);
- comunicazione (step 2 e 4).

In questo contesto, l'obiettivo di questo studio sperimentale è quello di valutare l'impatto di diverse scelte algoritmiche sulle differenti fasi dell'algoritmo.

5.1 Ottimizzazioni considerate

Basandosi sulla struttura precedentemente definita di un algoritmo di PSI, è possibile identificare diverse ottimizzazioni al fine di ridurre l'impatto delle componenti di costo discusse nella precedente sezione. Per questo scopo, sono state evidenziate le seguenti opportunità di ottimizzazione:

- riduzione degli elementi candidati per il PSI;
- parallelizzazione tramite partizionamento degli insiemi in input (fasi 1 e 3).

Inoltre, nell'ipotesi di scenari in cui il PSI venga ricalcolato periodicamente senza dover aggiornare i parametri di sicurezza (per esempio, security bit, chiavi crittografiche, sali), potrebbe essere possibile ottenere un ulteriore miglioramento delle prestazioni attraverso il key-value caching di elelenti oscurati (fasi 1 e 3) volto ad evitare la ripetizione del calcolo di oscuramento. Sulla base di queste osservazioni sono state proposte tre diverse ottimizzazioni basate su:

1. utilizzo del parallelismo, discussa nel Paragrafo 5.4.4;
2. utilizzo di key value-caching, discussa nel Paragrafo 5.4.5;
3. utilizzo del Bloom Filter, discussa nel Paragrafo 5.4.6.

Nel seguito di questa sezione verrà analizzato in modo estensivo l'impatto dell'utilizzo di queste ottimizzazioni sulle prestazioni per tutti gli algoritmi considerati e verranno discusse le implicazioni del loro uso sulla privacy del protocollo.

5.2 Metriche di performance

Al fine di valutare le prestazioni di differenti implementazioni di PSI risulta necessario utilizzare delle metriche come criterio di confronto. In particolar modo verranno considerati:

- il *tempo di esecuzione*, inteso come tempo di wall-clock time richiesto per completare una fase o l'interezza di un algoritmo.
- lo *speedup*, inteso come guadagno percentuale sul tempo di esecuzione rispetto ad una soluzione base, particolarmente utile per calcolare l'impatto delle ottimizzazioni proposte.

5.3 Setup sperimentale ed algoritmi considerati

Gli algoritmi Hash-PSI, DH-PSI, ECDH-PSI, BS-PSI ed ECBS-PSI sono stati implementati in Java 8. Le funzioni hash e di generazione di chiave utilizzate sono fornite dalla libreria Java Security. Tuttavia, non è possibile utilizzare direttamente cifrari offerti dalla libreria stessa per implementare le primitive crittografiche in quanto questi attuano delle operazioni (per esempio, aggiunta di padding) sui dati che rompono gli schemi su cui si basano gli algoritmi discussi in Sezione 4. Quindi, tutte le operazioni crittografiche, ossia esponenziazioni e moltiplicazioni per RSA e DH, e addizioni e moltiplicazioni per algoritmi basati su curve ellittiche, sono state implementate utilizzando le operazioni matematiche offerte dalla classe Java BigInteger e dal package ec.math offerto dalla libreria Java BouncyCastle. Per il Bloom Filter è stata utilizzata l'implementazione offerta da Google nella libreria Guava (Google Core Libraries for Java).

Gli esperimenti discussi in questo documento sono stati eseguiti su una infrastruttura Cloud noleggiata da Amazon Web Services. In particolare, ove non indicato, è stata utilizzata una istanza EC2 m5.4xlarge caratterizzata da 16 vCore (di cui 8 core fisici) e 64 GB di memoria. Al momento della stesura di questo documento, il costo di questa macchina è pari a 475€ al mese se noleggiata in modalità on-demand, oppure 340€ al mese nel caso di pagamento anticipato per un anno. Dal punto di vista software, le macchine noleggiate sono state configurate con Debian 10 (debian-10-amd64-20210329-591) e AdoptOpenJDK build 1.8.0_292-b10. Come punto di partenza dell'ambiente software, è stata utilizzata una installazione pulita di Debian 10 disponibile come AMI gratuita (ami-0245697ee3e07e755).

5.4 Analisi comparativa degli algoritmi di PSI e delle ottimizzazioni

In questa sezione vengono discusse e confrontate le prestazioni di Hash-PSI, DH-PSI, ECDH-PSI, BS-PSI, ECBS-PSI presentati in Sezione 4, differenziando i risultati ottenuti eseguendo gli algoritmi nel ruolo di server da quelli ottenuti da client. Escluso il paragrafo 5.4.8, l'analisi dei tempi di esecuzione considera unicamente i tempi della componente computazionale, escludendo quindi i tempi di comunicazione.

Per quanto riguarda gli elementi dei dataset, ove non indicato, viene considerato lo scenario applicativo dello scambio di informazioni relative a numeri di carte di credito (PAN). Senza considerare proprietà peculiari nella struttura dello specifico spazio di input, ciascun elemento degli insiemi viene rappresentata da 16 caratteri numerici casuali. Di conseguenza, l'entropia del dominio è pari a 54 bit.

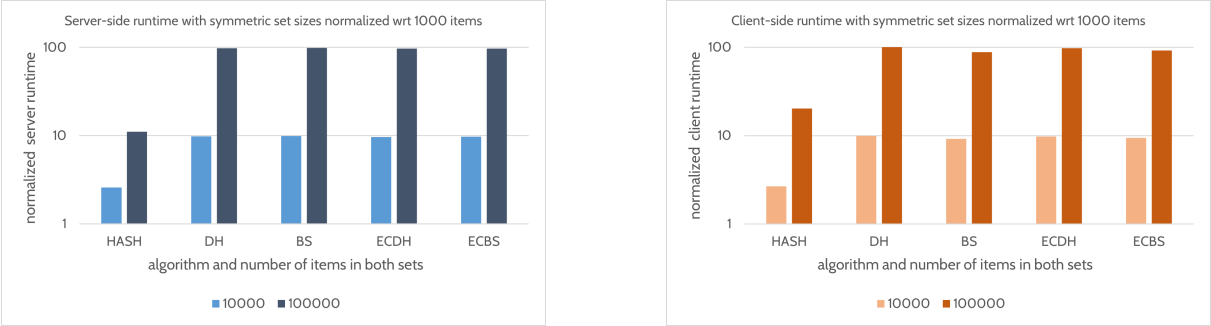


Figura 13: *Tempo di esecuzione al variare del numero di elementi presenti in entrambi gli insiemi normalizzati rispetto al tempo di esecuzione ottenuto con 1.000 elementi.*

Inizialmente vengono studiati gli algoritmi senza alcuna ottimizzazione al variare di alcuni parametri generali, quali la dimensione dei dataset, la differente percentuale di sovrapposizione, o differenti livelli di sicurezza espressi in termini di security bit. Questo studio iniziale permette di identificare come questi parametri influenzano i tempi di esecuzione, con l'obiettivo di identificare dei valori di default da utilizzare per studiare l'efficacia delle ottimizzazioni al variare dei suoi parametri specifici senza dover esplorare l'intero spazio delle configurazioni.

Nella fase finale dell'analisi vengono confrontate alcune delle soluzioni più interessanti in termini di trade-off tra livello di sicurezza e prestazioni, con l'obiettivo di supportare il processo decisionale atto a determinare la configurazione ideale da utilizzare in differenti contesti applicativi.

5.4.1 Dimensione degli insiemi

In questo paragrafo viene studiato in che modo la dimensione degli insiemi del server e del client influenza i tempi di esecuzione per entrambi i ruoli.

In Figura 13 vengono mostrati i tempi di esecuzione considerando dataset con lo stesso numero di elementi sia per il client che per il server e una percentuale di sovrapposizione del 10%. In particolare, i grafici mostrano il tempo di esecuzioni del server e del client normalizzato rispetto ai risultati ottenuti con 1.000 elementi in entrambi gli insiemi. Sia per il client che per il server possiamo osservare che il tempo di esecuzione è proporzionale alla somma della dimensione dei due insiemi per DH-PSI, BS-PSI, ECDH-PSI e ECBS-PSI. Questo comportamento è conforme ai risultati di letteratura discussi nella Sezione 4. Differentemente, Hash-PSI mostra una crescita inferiore. Questo è dovuto al minor costo nelle operazione di hashing rispetto a

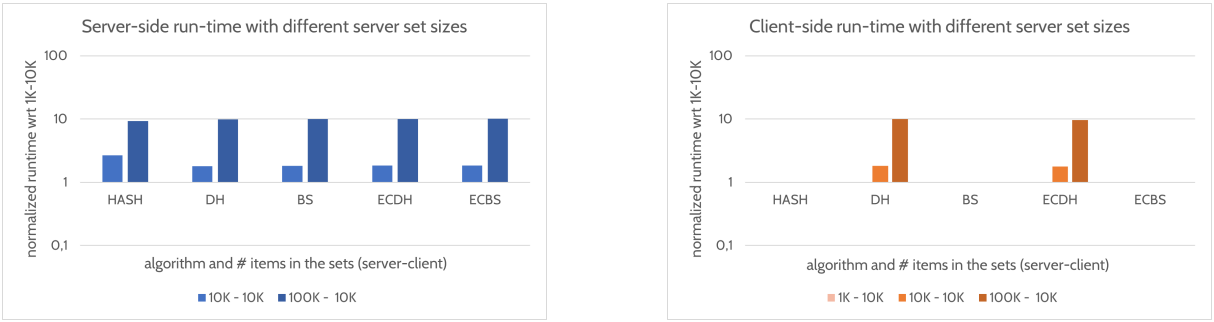


Figura 14: *Tempo di esecuzione al variare della dimensione del dataset del server normalizzato rispetto al tempo di esecuzione ottenuto con 1.000 elementi. La dimensione del dataset del client è costante e pari a 10.000 elementi.*

quelle crittografiche che comportano, per input relativamente piccoli, un'alta incidenza dei costi fissi indipendenti dalla dimensione degli insiemi. Infatti, con Hash-PSI si ottengono tempi di esecuzione di 19, 49 e 211 millisecondi per il server con rispettivamente 1.000, 10.000 e 100.000 elementi, mentre con gli altri algoritmi si ottengono tempi nell'ordine dei secondi, decine di secondi, o minuti. Maggiori dettagli sui tempi di esecuzione dei differenti algoritmi vengono mostrati nel seguito di questa sezione.

Oltre a considerare l'andamento dei tempi di esecuzione al crescere della dimensione di entrambi gli insiemi, è di interesse analizzare come l'aumento nel numero di elementi di un solo insieme alla volta, ovvero solo dell'insieme del client o del server, lasciando la dimensione dell'altro insieme invariato, può influenzare i tempi di esecuzione per entrambi i ruoli.

In Figura 14 viene rappresentato il tempo di esecuzione al variare della dimensione dell'insieme del server mantenendo la dimensione dell'insieme del client costante e pari a 10.000 elementi. Per quanto riguarda i risultati del server, passando da 1.000 a 10.000 elementi si osserva un aumento dei tempi di esecuzione dell'80%, mentre il tempo di esecuzione aumenta di circa dieci volte incrementando la dimensione dal dataset del server da 1.000 a 100.000 elementi. Questi risultati mostrano ancora una volta una crescita del tempo di esecuzione proporzionale alla somma del numero di elementi negli insiemi. Differentemente, il tempo di esecuzione del client non è influenzato in alcun modo dalla variazione nel numero di elementi nell'insieme del server per Hash-PSI, BS-PSI e ECBS-PSI, mentre mostra un andamento simile a quanto osservato per il server per DH-PSI ed ECDH-PSI. Questo risultato è conforme alla struttura della componente computazionale dei rispettivi algoritmi. Infatti, DH-PSI e ECDH-PSI richiedono l'applicazione di una funzione di oscuramento ad entrambi i dataset sia da parte del client che del server, mentre Hash-PSI, BS-PSI e ECBS-PSI richiedono al client di applicare una funzione di oscuramento

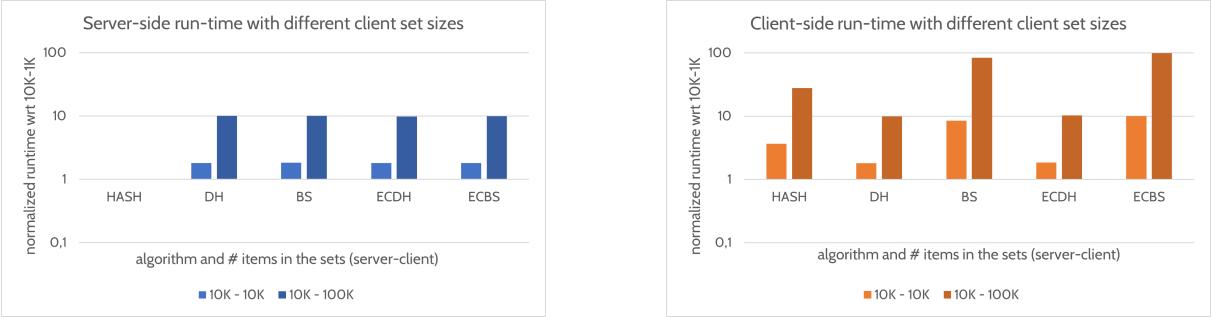


Figura 15: *Tempo di esecuzione al variare della dimensione del dataset del client normalizzato rispetto al tempo di esecuzione ottenuto con 1.000 elementi. La dimensione del dataset del server è costante e pari a 10.000 elementi.*

solo agli elementi presenti nel proprio dataset.

In Figura 15 viene descritto l'andamento del tempo di esecuzione al variare della dimensione dell'insieme del client mantenendo fissa la dimensione dell'insieme del server (10.000 elementi). Per tutti gli algoritmi escluso Hash-PSI, l'aumento della dimensione dell'insieme del client comporta un aumento nel tempo di esecuzione proporzionale alla somma del numero degli elementi presenti nei due insiemi. Di conseguenza, un aumento nel numero di elementi nell'insieme del client ha le stesse conseguenze sul tempo di esecuzione del server di aumento nel numero di elementi presenti nel suo dataset. Differentemente, in Hash-PSI, il costo computazionale, sia per il server che per il client, è indipendente dalla dimensione dell'insieme dell'altra parte. Per quanto riguarda il client, per DH-PSI e ECDH-PSI, l'aumento nel tempo di esecuzione è equivalente all'aumento della somma del numero di elementi nei due insiemi, mentre cresce linearmente rispetto al numero di elementi dell'insieme del client per BS-PSI e ECBS-PSI. Come discusso in precedenza, questi risultati sono causati dal fatto che gli algoritmi basati su Diffie-Hellman richiedono l'applicazione di funzioni crittografiche ad entrambi gli insiemi, per cui moltiplicare per una costante k la dimensione dell'insieme del client non comporta un aumento di k volte del tempo di esecuzione, mentre questo avviene per gli algoritmi basati su Blind Signature. Nonostante il tempo di esecuzione di Hash-PSI scali linearmente con il numero di elementi nell'insieme del client, l'incremento nel tempo di esecuzione è inferiore rispetto a BS-PSI ed ECBS-PSI a causa della maggiore incidenza di costi fissi non direttamente imputabili al calcolo degli hash.

In conclusione, gli andamenti generalmente uniformi e predicibili dei tempi di esecuzione al variare delle dimensioni dei dataset permette di procedere considerando un unico valore nella dimensione di entrambi gli insieme, lasciando al lettore la possibilità di calcolare tramite le relazioni discusse in questo paragrafo i risultati previsti con dimensioni dei dataset differenti. Di

Figura 16: *Tempo di esecuzione al variare della percentuale di sovrapposizione.*

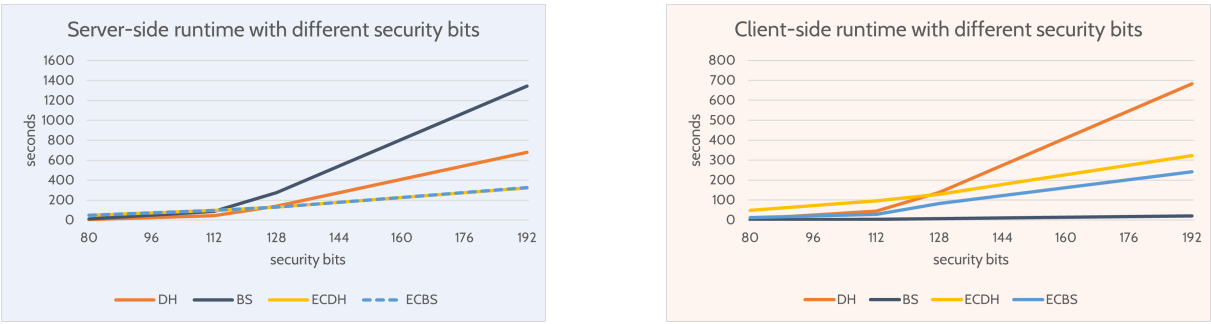


Figura 17: *Tempo di esecuzione al variare del numero di security bit.*

conseguenza, per il resto di questa sezione, ove non differentemente indicato, vengono considerati insiemi di 10.000 elementi per entrambi i dataset.

5.4.2 Percentuale di sovrapposizione

In questo paragrafo viene analizzato come la percentuale di sovrapposizione influenza i tempi di esecuzione del server e del client per algoritmi senza ottimizzazione. Come mostrato in Figura 16, il tempo di esecuzione è indipendente dalla percentuale di sovrapposizione. Di conseguenza, nel seguito di questa sezione, ove non differentemente indicato, viene considerata una percentuale di sovrapposizione del 10%.

Security Bit	NIST	ECRYPT-CSA	Lenstra Updated Equations	Network Working Group RFC3766
80	Legacy	Legacy	2016	1981
112	2030	-	2064	2029
128	2030 & beyond	2028	2088	2053
192	2030 & beyond	-	2184	2149
256	2030 & beyond	2068	2280	2245

Tabella 1: *Raccomandazioni riguardo la sicurezza offerta da un dato numero di bit di sicurezza.*

5.4.3 Security bit

Il numero di security bit (S), da non confondere con la lunghezza delle chiavi crittografiche, tipicamente viene identificato come:

- numero di passi che l'algoritmo più efficiente noto in letteratura deve compiere, oppure
- il numero di bit che un algoritmo a forza bruta deve identificare per rompere deterministicamente uno schema di sicurezza (per esempio, cifratura e/o secure hash).

In Tabella 1 vengono riportate le raccomandazioni temporali riguardo l'efficacia di un dato numero di security bit fatte da diverse agenzie (NIST ed ECRYPT-CSA) o stimate tramite calcoli teorici (equazioni di Lenstra e di Network Working Group)[5]. Ogni cella perciò riporta, ove espressa, la data entro cui un certo numero di security bit è ritenuto sufficiente secondo una determinata agenzia o equazione (la dicitura Legacy indica che il corrispettivo numero di security bit non deve essere utilizzato per nuovi sistemi, ma solo per motivi di compatibilità con software legacy).

Dalla lettura della sopracitata tabella risulta chiaro che è raccomandato utilizzare almeno 112 bit di sicurezza e che 192 bit di sicurezza possono essere considerati sufficienti per almeno altri 10 anni. Di conseguenza si è scelto di considerare come possibili valori di security bit 112, 128 e 192, considerando anche 80 per mostrare i differenti andamenti delle prestazioni degli algoritmi in fase di studio al crescere del livello di sicurezza. Considerando che il numero di

[5]Si ricorda che quelle riportate in Tabella 1 sono raccomandazioni fatte da agenzie estere e/o individui non direttamente connessi con gli autori di questo documento, i quali non si assumono responsabilità sulle conseguenze di seguire tali raccomandazioni. Infatti, queste ultime non possono garantire l'effettiva sicurezza fino alla data indicata, perciò sono da intendere a titolo informativo.

security bit offerti da uno schema di sicurezza è tipicamente minore rispetto alla lunghezza delle chiavi espressa in bit ed è caratteristico dello specifico schema utilizzato, risulta necessario convertire i security bit in lunghezza delle chiavi utilizzate (ad esempio per schemi di cifratura). Infatti, in letteratura è noto che:

- Schemi di cifratura basati su curve ellittiche (EC) offrono un numero di security bit pari alla metà della lunghezza della chiave. Di conseguenza, i protocolli PSI basati su EC utilizzeranno chiavi pari lunghe 160, 224, 256 e 384 e 512 bit.
- La stima dei security bit offerti da uno schema basato sulla difficoltà di inversione della fattorizzazione (RSA) o del logaritmo discreto (Diffie-Hellman) risulta legato alla complessità dell'algoritmo General Number Field Sieve (GNFS). Nello specifico verranno considerati moduli a 1024, 2048, 3072 e 7680.

Infine per schemi di PSI basati su hashing, la sicurezza è offerta direttamente dai bit di entropia intrinseci nel dominio. In altre parole, per invertire un dato hash H tramite approccio a forza bruta è necessario calcolare, per ogni valore appartenente al dominio, il suo hash H' e verificare se corrisponde a quello di interesse H. Tuttavia, non è garantito che i bit da identificare siano effettivamente quelli richiesti per rappresentare un valore nel dominio. Prendiamo ad esempio il Codice Fiscale (CF). Tipicamente questo è costituito da 16 caratteri alfanumerici. Siccome per rappresentare un carattere sono richiesti circa 5.1 bit, potrebbe risultare necessario identificare circa 82 bit. Tale quantità risulta già al limite dell'insicurezza se si considerano le raccomandazioni riportate in Tabella 1. Tuttavia il numero di bit di reale entropia è assai minore. Infatti, la struttura del CF è tale per cui:

- 6 caratteri alfabetici sono derivati da nome e cognome (al più 25 bit di entropia invece di 30.6);
- 5 caratteri alfanumerici derivati dalla data di nascita (al più 16 bit di entropia invece di 25.5);
- 4 caratteri alfanumerici derivati dal comune di nascita (al più 13 bit di entropia invece di 20.4);
- 1 carattere alfanumerico di controllo derivato dai 15 caratteri precedenti (0 bit di entropia invece di 5.1).

Perciò, il CF offre in totale al più 52 bit di entropia, ben al di sotto della soglia (80 bit) ritenuta legacy in Tabella 1.

In Figura 17 vengono confrontati i tempi di esecuzione degli algoritmi crittografici considerando configurazioni da 80 a 192 security bit. Non viene incluso in questo studio Hash-PSI

poiché nel contesto dei PAN, il suo livello di sicurezza in termini di security bit non dipende dall'algoritmo di hashing utilizzato, ma dall'entropia dell'insieme degli input.

Per quanto riguarda il tempi di esecuzione del server, possiamo osservare tempi inferiori per DH-PSI e BS-PSI per un numero di security bit uguale o inferiore a 112 (equivalente a chiavi da 2048 per DH-PSI e BS-PSI, 224 bit per ECDH-PSI ed ECBS-PSI), mentre ECDH e ECBS offrono prestazioni migliori per più di 256 security bit (equivalente a chiavi da 3072 per DH-PSI e BS-PSI, 256 bit per ECDH-PSI ed ECBS-PSI). Osserviamo inoltre risultati identici per ECDH-PSI ed ECBS-PSI. Questo risultato è conforme alle aspettative poiché, per quanto riguarda il ruolo del server, entrambi gli algoritmi effettuano le medesime operazioni matematiche su dati della stessa dimensione, mentre, a parità di security bit, BS-PSI richiede operazioni più onerose dal punto di vista computazionale rispetto a DH-PSI.

I risultati per il client mostrano un comportamento differente: i tempi di esecuzione degli algoritmi basati su Blind Signature offrono sempre prestazioni superiori a quelli basati su Diffie-Hellman per tutti i valori di security bit considerati. In particolare, BS-PSI offre prestazioni significativamente superiori rispetto alle alternative, richiedendo solo $3,7$ secondi per l'esecuzione con 112 security bit e $7,4$ secondi per l'esecuzione con 128 security bit, contro, rispettivamente, i $44,3$ secondi e 140 secondi necessari per l'esecuzione di DH-PSI (considerando insiemi da 10.000 elementi e il 10% di percentuale di sovrapposizione). Per quanto riguarda gli algoritmi basati su Diffie-Helmann osserviamo un andamento similare a quanto osservato per il server: DH-PSI offre prestazioni migliori quando il numero di security bit è inferiore a 128 mentre ECDH-PSI richiede tempi di esecuzione inferiori per livelli di sicurezza superiori. Differentemente, BS-PSI richiede tempi di esecuzioni inferiori di ECBS-PSI per qualsiasi numero di security bit nell'intervallo considerato. In particolare, ECBS-PSI richiede un tempo di esecuzione 7 volte superiore rispetto a BS-PSI con 112 security bit, e 11 volte superiore con 128 security bit.

Considerati i requisiti di performance che motivano questo studio e il costo potenzialmente elevato degli algoritmi di PSI per dataset relativamente grandi, nell'analisi dei risultati delle ottimizzazioni verranno considerate esecuzioni con 112 security bit, mentre nelle comparazioni finali verranno inclusi sia i risultati con 112 che con 128 security bit. Questi valori, oltre a rappresentare i valori minimi di sicurezza attualmente considerati sicuri, sono di particolare interesse poiché rappresentano, quantomeno per il server, il confine in cui le implementazioni basate su curve ellittiche superano in prestazioni le configurazioni basate su crittografica asimmetrica tradizionale.

Figura 18: *Tempo di esecuzione al variare del numero di thread utilizzati.*

5.4.4 Ottimizzazione: parallelismo

L'utilizzo di risorse di calcolo parallele per ridurre i tempi di esecuzione è una ottimizzazione particolarmente interessante poiché permette di aumentare le prestazioni attraverso un utilizzo più efficiente delle risorse hardware senza introdurre alcun trade-off dal punto di vista della sicurezza. In Figura 18 viene illustrato lo speed-up degli algoritmi passando da 1 a 16 thread. Ricordiamo che lo studio sperimentale è stato effettuato su una macchina con 16 vCore e considerato che in questa analisi vengono ignorate le operazioni di I/O (es. letture da file o da network), è possibile ignorare esecuzioni con un numero di thread superiori al numero di vCore disponibili poiché non porterebbe alcun vantaggio prestazionale.

Sia per il client che per il server, i risultati mostrano una crescita lineare delle prestazioni fino ad 8 thread per tutti gli algoritmi tranne Hash-PSI, con uno speed-up equivalente al numero di thread utilizzati. Differentemente, da 8 a 16 thread, l'aumento prestazionale è limitato o assente. Tuttavia, la causa di questa scalabilità sub-ottimale non è di natura algoritmica ma micro-architetturale. Infatti, la maggioranza delle CPU moderne, incluso l'Intel Xeon Platinum 8259CL utilizzato per questo studio, sfrutta il Simultaneous Multithreading (SMT) per permettere l'esecuzione di molteplici thread su un singolo core fisico condividendo le principali risorse di calcolo (in questo caso 2 thread per core, anche detto 2-way SMT). Come discusso in [32, 29], l'utilizzo del 2-way SMT ha un impatto sulle prestazioni altamente dipendente della caratteristiche dell'applicazione. Infatti, nella maggioranza dei casi esso permette di ottenere uno speed-up tra il 20% e il 60% rispetto ad esecuzioni in cui viene eseguito solo un thread per core fisico, ma può portare un beneficio minimo, o perfino un degrado delle prestazioni, per applicazioni in cui la componente di calcolo domina sulle operazioni di scrittura o lettura dalla memoria. Questo è il caso degli algoritmi di PSI considerati in questo studio, nei quale la maggioranza del

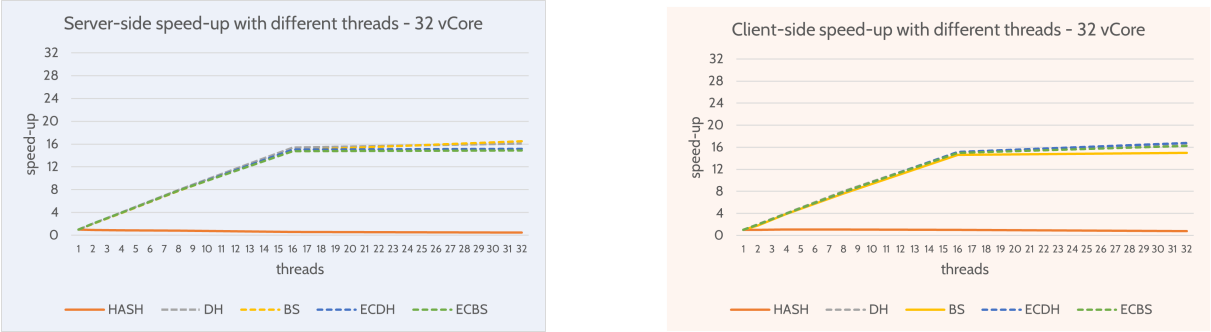


Figura 19: *Tempo di esecuzione al variare del numero di thread utilizzati con 32 vCore.*

tempo d'esecuzione è speso in operazioni crittografiche, le quali utilizzano in modo intensivo i sottosistemi di calcolo ed in modo solo marginale il sottosistema di memoria.

Per verificare questo fenomeno, in Figura 19 vengono rappresentati gli speed-up ottenuti eseguendo lo stesso test su una macchina da 32 vCore della stessa famiglia di quella utilizzata per il resto degli esperimenti. Come è possibile osservare dai grafici, avendo a disposizione 16 core fisici, si riesce ad ottenere uno speed-up lineare fino a 16 thread, e come previsto, un incremento minimo passando da 16 a 32 thread. Conseguentemente, possiamo concludere che, sfruttando il parallelismo, è possibile ottenere uno speed-up pari (o leggermente inferiore) al numero di core fisici della macchina per DH-PSI, BS-PSI, ECDH-PSI ed ECBS-PSI.

Per quanto riguarda Hash-PSI, osserviamo minimi cambiamenti nei tempi di esecuzione al variare del numero di thread utilizzati. Questo è causato dal basso costo delle operazioni di hashing, che comportano una maggiore incidenza delle porzioni di codice seriali e un maggior peso relativo degli overhead associati al multi-threading (es. generazione dei thread), i quali bilanciano negativamente i possibili vantaggi ottenibili dall'esecuzione parallela.

5.4.5 Ottimizzazione: cache

In contesti applicativi che prevedono una esecuzione periodica di algoritmi di PSI, se una porzione dei dataset rimane costante tra differenti esecuzioni, è possibile ricorrere a tecniche di caching per ridurre i tempi di esecuzione. L'idea alla base di questa ottimizzazione è quella di salvare il risultato delle funzioni di oscuramento in memoria o su disco all'interno di mappe chiave-valore (le cache) che associano i valori in input (le chiavi) ai rispettivi valori risultanti dall'applicazione della funzione di oscuramento. Prima di applicare la funzione di oscuramento ad un determina-

to elemento, viene prima cercata la sua presenza all'interno della cache. Se il valore in chiaro dell'elemento è presente come chiave viene utilizzato direttamente il valore presente nella cache anziché calcolarlo. Altrimenti, viene applicata la funzione di oscuramento al valore in chiaro e viene salvata la coppia nella cache in modo da renderla disponibile per esecuzioni successive.

In caso di operazioni onerose dal punto di vista computazionale, come l'applicazione di funzioni crittografiche, la lettura di un elemento dalla cache (cache hit) può comportare una risparmio significativo rispetto all'esecuzione di funzioni di oscuramento. Tuttavia, l'impatto sulle prestazioni dell'utilizzo della cache può dipendere da molteplici fattori:

- probabilità di cache hit: rappresenta la probabilità di trovare un determinato elemento all'interno della cache. Può dipendere dalla variabilità dei dataset tra le diverse esecuzioni dell'algoritmo di PSI, dal numero di elementi memorizzabili nella cache per limiti di spazio o dalla tecnica di sostituzione dei dati in cache una volta raggiunto il massimo spazio utilizzabile. Supponendo che la lettura dalla cache sia computazionalmente più economica rispetto all'applicazione delle funzioni di oscuramento, a probabilità di cache hit più elevate corrispondono maggiori prestazioni.
- tempo medio di ricerca nella cache: rappresenta il tempo medio necessario per cercare una determinata coppia chiave-valore all'interno della cache. Può dipendere dalla struttura dati utilizzata e dal supporto fisico utilizzato per il salvataggio della cache. A valori inferiori corrispondono prestazioni maggiori.
- tempo medio di esecuzione delle funzioni di oscuramento: rappresenta il tempo medio necessario per calcolare la funzione di oscuramento per un elemento del dataset. Rappresenta il costo computazionale risparmiato attraverso l'utilizzo della cache in caso di cache hit. Il suo valore varia in base all'algoritmo di oscuramento e alla dimensione della chiave utilizzata. Maggiore è il suo valore, maggiore è lo speed-up ottenuto utilizzando la cache.

Per quanto riguarda il supporto fisico, accedere ad una cache residente in memoria permette di minimizzare i tempi di ricerca rispetto a cache salvate su disco. Tuttavia, questo approccio presenta i seguenti svantaggi:

- richiede una quantità di memoria sufficientemente grande da contenere la struttura dati;
- viene eliminata ad ogni riavvio dell'applicazione o della macchina.

Per attenuare le implicazioni di questi limiti è possibile ridurre la quantità di dati salvati nella cache definendo una dimensione massima dei dati salvati, implementando poi una politica di sostituzione che permetta di mantenere in cache solo i dati acceduti più di recente. Inoltre, per ovviare alla cancellazione al riavvio dell'applicazione o della macchina, è possibile effettuare un

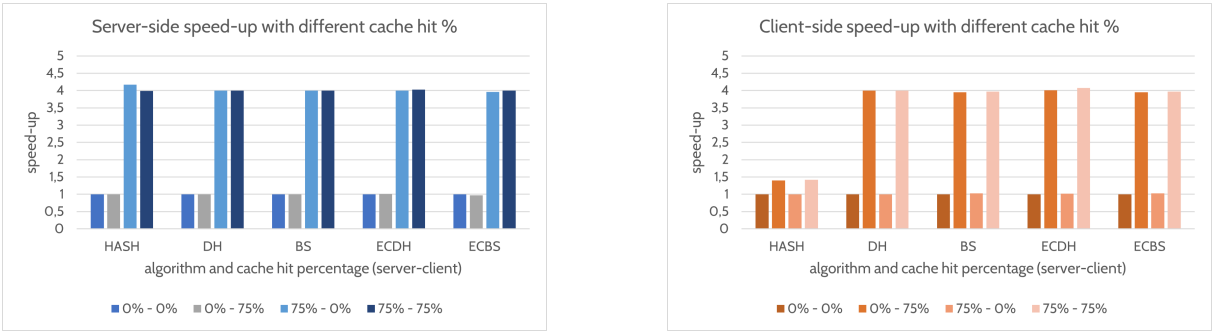


Figura 20: *Speed-up con differenti configurazioni della cache rispetto alla configurazione con cache disabilitata per entrambi i ruoli.*

salvataggio periodico su disco dell'immagine della cache, la quale può essere caricata in memoria all'avvio dell'applicativo. Considerate queste possibili ottimizzazioni, la disponibilità crescente di memoria nei sistemi moderni ed i forti requisiti di performance che motivano questo studio, nel seguito di questo studio sperimentale si assume l'utilizzo di una cache residente in memoria.

Un vantaggio rilevante della cache è rappresentata dal fatto che è una tecnica applicata localmente sulla specifica istanza dell'applicazione e, di conseguenza, può essere utilizzata solo sul server, solo sul client, o per entrambi i ruoli, in modo completamente trasparente rispetto all'altra parte. Oltre alle prestazioni, i principali aspetti da tenere in considerazione per scegliere se utilizza la cache sono la disponibilità di memoria non volatile (nel caso di cache residente in memoria necessaria per effettuare salvataggi periodico su disco) e la possibile degradazione del livello di privacy. Infatti, per poter utilizzare i valori oscurati letti dalla cache è necessario che l'istanza dell'applicazione utilizzi gli stessi parametri delle funzioni di oscuramento (e in particolare le stesse chiavi) utilizzate nelle esecuzioni precedenti, cioè quando è stato effettuato il salvataggio del relativo elemento all'interno della cache. In particolare, l'utilizzo degli stessi parametri per le funzioni di oscuramento tra più esecuzioni invalida il requisito opzionale della *unlinkability* per la parte (client o server) che utilizza la cache, poiché permette all'altra parte di relazionare gli input tra diverse esecuzioni, per esempio permettendo di identificare la rimozione di un valore oscurato o l'inserimento di un elemento oscurato precedentemente non presente. Come anticipato, questo è un requisito opzionale che non invalida i requisiti essenziali di *server privacy*, *client privacy* o *correttezza* discussi nella Sezione 2, ma può comunque rappresentare uno svantaggio rilevante in contesti applicativi che richiedono un livello di privacy particolarmente alto.

Gli algoritmi basati su funzioni crittografiche come DH-PSI, BS-PSI, ECDH-PSI ed ECBS-

PSI richiedono l'applicazione di funzioni di oscuramento sia al proprio dataset che agli elementi ricevuti dall'altra parte già in formato oscurato. In questo scenario, la cache può essere utilizzata non solo per ridurre il costo delle operazioni crittografiche applicate al proprio insieme, ma anche per velocizzare il processamento di elementi ricevuti dall'altra parte. Tuttavia, per ottenere dei cache hit su quest'altro insieme di dati, per le stesse motivazioni discusse in precedenza, è necessario che l'altra parte utilizzi gli stessi parametri della funzione di oscuramento tra più esecuzioni. Nel contesto di questo studio sperimentale abbiamo considerato l'utilizzo degli stessi parametri per le funzioni di oscuramento in esecuzioni successive dell'algoritmo di PSI indipendentemente dall'utilizzo o meno della cache sul singolo ruolo. Di conseguenza, la percentuale di cache hit deve essere interpretata come la probabilità di trovare in cache un valore già precomputato, sia esso appartenente al proprio dataset o ad elementi ricevuti dall'altra parte. Pur non avendo effettuato un'analisi sperimentale estensiva in scenari in cui solo una delle due parti utilizza parametri delle funzioni di oscuramento differenti tra più esecuzioni, possiamo aspettarci speed-up dimezzati rispetto a quello ottenuti nello scenario considerato.

In Figura 20 vengono mostrati gli speed-up ottenuti con diverse configurazioni della cache rispetto all'esecuzione con la cache disabilitata per entrambi i ruoli, il che è equivalente alla configurazione con una probabilità di cache hit dello 0% sia per il client che per il server. In particolare, viene considerata la configurazione con la cache disabilitata sul server e con il 75% di cache hit sul client, la configurazione con il 75% di cache hit sul server e la cache disabilitata sul client, e la configurazione con il 75% di cache hit per entrambi i ruoli.

Per quanto riguarda il server, possiamo osservare uno speed-up di 4 per le configurazioni in cui il server ha una probabilità di cache hit del 75%. Allo stesso modo, osserviamo uno speed-up di 4 sul client per le configurazioni che utilizzano la cache sul client, escluso Hash-PSI dove, a causa del basso costo della funzione di oscuramento, il guadagno nell'utilizzo della cache è limitato. Come previsto, l'utilizzo della cache influenza unicamente lo speed-up per il ruolo in cui viene utilizzata, e non influenza il tempo di esecuzione dell'altra parte. Inoltre, per tutti gli algoritmi escluso Hash-PSI, l'impatto della lettura dalla cache è irrilevante rispetto al costo delle funzioni di oscuramento, permettendo dunque di ottenere uno speed-up equivalente ad $1/(1 - CHP)$ dove CHP è la probabilità di cache hit.

Al fine di valutare la fruibilità del salvataggio in memoria della cache è essenziale stimare lo spazio in memoria che essa occupa al variare della dimensione degli input, della tipologia di algoritmo e del livello di sicurezza. Per questo scopo è stato ritenuto preferibile un approccio analitico rispetto ad uno meramente sperimentale a causa dell'elevata variabilità introdotta dalla Garbage Collection nella lettura da software della memoria utilizzata.

Algoritmo	Security Bit	Taglia Elemento A (B)	Taglia Elemento B (B)
Hash-PSI	N/A	48	N/A
DH-PSI	112	272	512
	128	400	768
BS-PSI	112	272	512
	128	400	768
ECDH-PSI	112	74	116
	128	82	132
ECBS-PSI	112	74	116
	128	82	132

Tabella 2: *Dimensioni in byte degli elementi della cache afferenti al proprio dataset (Elemento A) o al dataset dell'altra parte (Elemento B) al variare dell'algoritmo e del numero di bit di sicurezza considerando valori in chiaro di 16 byte.*

Come discusso, escludendo Hash-PSI che richiede unicamente il salvataggio in cache degli elementi appartenenti al proprio dataset, ogni ruolo salva in cache due tipologie di elementi differenti: una relativa agli elementi del proprio dataset e l'altra afferente agli elementi dell'altra parte ricevuti in formato oscurato. Di conseguenza, per stimare la dimensione della cache è necessario calcolare per entrambe le tipologie di elementi la dimensione in byte delle rispettive coppie chiave-valore inserite nelle mappe. Per quanto riguarda i valori, essi assumono sempre la dimensione in byte dell'output delle funzioni di oscuramento. Differentemente, le chiavi assumono la dimensione degli elementi in chiaro quando riferite ad elementi del proprio dataset (16 byte per i PAN), mentre assumono una dimensione pari alla lunghezza in byte dell'output della funzione di oscuramento quando associate ad elementi del dataset dell'altra parte. In Tabella 2 vengono indicate le dimensioni in byte degli elementi, come somma della dimensione della chiave e del valore, al variare dell'algoritmo e del numero di bit di sicurezza, differenziando gli elementi riferiti al proprio dataset (indicati come Taglia Elemento A) da quelli afferenti all'altra parte (indicati come Taglia Elemento B). La dimensione dei valori in chiaro viene considerata pari a 16 byte per risultare conformi al contesto dello scambio confidenziale di PAN. Come mostrato nella tabella, le dimensioni degli elementi sono equivalenti per DH-PSI e BS-PSI e per ECDH-PSI ed ECBS-PSI. Inoltre, come conseguenza della maggior compattezza nella rappresentazione dei valori oscurati, osserviamo una dimensione degli elementi significativamente inferiore per gli algoritmi basati su curve ellittiche rispetto a DH-PSI e BS-PSI.

Indicando con set_A il dataset dalla parte considerata, e con set_B il dataset dell'altra parte, per ottenere una percentuale di cache hit pari a CHP_A rispetto agli elementi di A, e di CHP_B

Algoritmo	Security Bit	Min. Cache Size w/ 1M Items (MB)
Hash-PSI	N/A	34.33
DH-PSI	112	560.76
	128	835.41
BS-PSI	112	560.76
	128	835.41
ECDH-PSI	112	135.89
	128	153.06
ECBS-PSI	112	135.89
	128	153.06

Tabella 3: *Dimensioni minima in megabyte della cache considerando il 75% di cache hit per entrambe le tipologie di elementi e dataset da un milione di elementi al variare dell'algoritmo e del numero di bit di sicurezza.*

rispetto agli elementi di B, è necessario che:

$$dim(cache_A) \geq dim(set_A) * CHP_A + dim(set_B) * CHP_B \tag{12}$$

La disequazione assume il segno uguale solo nel caso in cui tutti gli elementi della cache vengano effettivamente lette durante l'esecuzione dell'algoritmo di PSI. Tuttavia, sopratutto per quanto riguarda gli elementi dell'insieme dell'altra parte, è ragionevole sia necessario un numero di elementi significativamente maggiore. Al fine di fornire un'indicazione orientativa sulla dimensione della cache, in Tabella 3 viene indicato lo spazio minimo in megabyte occupato della cache al variare dell'algoritmo e numero di bit di sicurezza, considerando il 75% come percentuale di cache hit per entrambe le tipologie di elementi e dataset con un milione di elementi. I dati esposti nella tabella sono conformi alle dimensione degli elementi mostrati in Tabella 2. Considerata la disponibilità di decine o anche centinaia di gigabyte di memoria nei sistemi enterprise moderni, essi validano la possibilità di utilizzare cache residenti in memoria anche per dataset relativamente grandi.

5.4.6 Ottimizzazione: Bloom Filter

Il Bloom Filter è una struttura dati probabilistica che permette di testare l'appartenenza di un elemento ad un insieme. Per sua costruzione, l'interrogazione del Bloom Filter può ottenere come risultato dei falsi positivi ma non dei falsi negativi, per cui può distinguere elementi che non fanno sicuramente parte dell'insieme da elementi che potrebbero farne parte. Una caratteristica

essenziale del Bloom Filter è la sua dimensione molto inferiore rispetto alla rappresentazione tradizionale dell'intero insieme, il che lo rende una struttura dati molto efficace per velocizzare applicativi che richiedono di verificare l'appartenenza ad insiemi troppo grandi per essere salvati in memoria in modo estensivo.

Il Bloom Filter è costituito da un array di m bit, inizialmente tutti impostati a 0, e k differenti funzioni hash che associano ogni elemento dell'insieme ad una delle m celle dell'array. Per aggiungere un elemento, vengono calcolate le k funzioni hash e vengono impostate ad 1 tutte le celle dell'array corrispondenti ai risultati delle funzioni hash. Per interrogare il Bloom Filter, è necessario calcolare le k funzioni hash (le stesse usate per l'inserimento) passando l'elemento da testare come input, e verificare che tutte le celle dell'array corrispondenti ai risultati delle funzioni hash siano impostate ad 1. Come conseguenza di questa struttura, il Bloom Filter non supporta la rimozione di elementi, poiché risulta impossibile definire quale dei k bit dovrebbero essere reimpostati a 0 senza invalidare altri elementi precedentemente inseriti. Un altro aspetto di interesse è che la selezione dei parametri k e m, dipendentemente dalla dimensione degli elementi inseriti, permette di ottenere differenti trade-off tra la compattezza della struttura dati e la percentuale di falsi positivi.

Nel contesto dell'ottimizzazione proposta, l'idea è quella di utilizzare un Bloom Filter generato a partire dal dataset del server, per filtrare il dataset del client prima dell'esecuzione effettiva dell'algoritmo PSI. Infatti, una peculiarità di questa ottimizzazione è che viene eseguita direttamente sui dati in chiaro, risultando dunque ortogonale rispetto all'algoritmo specifico. In termini prestazionali, il Bloom Filter permette di eseguire l'algoritmo di PSI con un dataset del client di dimensione inferiore che, come osservato nel Paragrafo 5.4.1, permette di ridurre i tempi di esecuzione sia per il client che per il server, pagando però il costo computazionale della sua generazione (sul server) e interrogazione (sul client). Definiti FPP la percentuale di falsi positivi, set_{client} il dataset del client ed OP la percentuale di sovrapposizione rispetto al dataset del client, la dimensione attesa del dataset del client in seguito all'interrogazione del Bloom Filter ($set_{client-filtered}$) è la seguente:

$$dim(set_{client-filtered}) = dim(set_{client}) * OP + (1 - OP) * dim(set_{client}) * FPP \qquad (13)$$

Come si evince da questa relazione, a valori maggiori della percentuale di sovrapposizione o della probabilità di falsi positivi corrispondono dimensioni maggiori del dataset filtrato e, di conseguenza, minori vantaggi prestazionali.

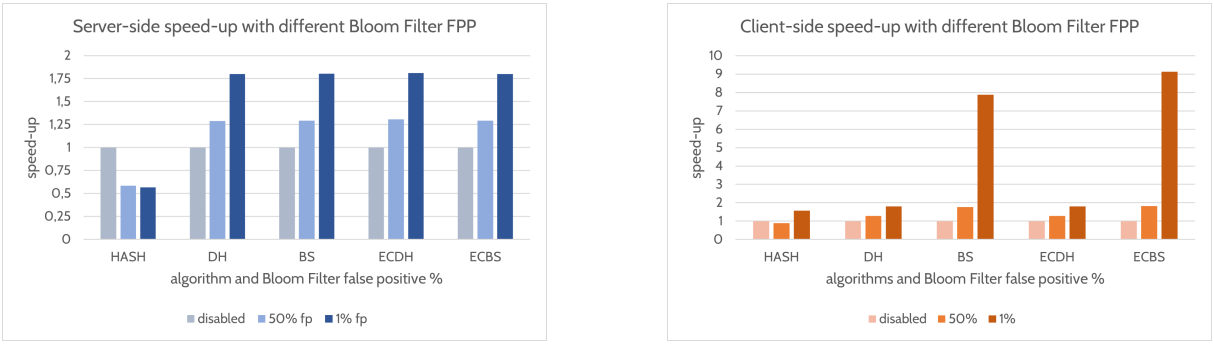


Figura 21: *Speed-up ottenuto con l'utilizzo del Bloom Filter considerando il 50% e l'1% come probabilità di falsi positivi e una percentuale di sovrapposizione del 10%.*

In Figura 21 viene rappresentato lo speed-up ottenuto sul server e sul client utilizzando il Bloom Filter considerando una probabilità di falsi positivi del 50% e dell'1%, con una percentuale di sovrapposizione del 10% e dataset da 10.000 elementi. Applicando l'Equazione 13, il valore atteso della dimensione del dataset del client in seguito all'applicazione del filtro è pari rispettivamente a 5500 e 1090 con il 50% e l'1% di probabilità di falsi positivi. Escluso Hash-PSI, i risultati del server mostrano un incremento prestazionale del 25% circa con la probabilità di falsi positivi del 50% e del 75% con la probabilità di falsi positivi all'1%. Differentemente, l'utilizzo del Bloom Filter sul server mostra una riduzione delle prestazioni per Hash-PSI. Questo è dovuto al costo della generazione del Bloom Filter stesso che, essendo costante per tutti gli algoritmi, ha un peso relativo molto maggiore per Hash-PSI. Per quanto riguarda il client, osserviamo risultati equivalenti a quelli ottenuti per il server per DH-PSI ed ECDH-PSI, mentre BS-PSI ed ECBS-PSI mostrano rispettivamente uno speed-up del 76% e 81%, con la probabilità di falsi positivi al 50%, e rispettivamente uno speed-up del 680% e del 812% con la probabilità di falsi positivi all'1%. Questo elevato speed-up per gli algoritmi basati su Blind Signature è conforme ai risultati discussi nel paragrafo 5.4.1, dove abbiamo osservato che per questi algoritmi il tempo di esecuzione del client cresce linearmente al crescere della dimensione del dataset del client. D'altra parte, la differenza dei risultati ottenuti per BS-PSI e ECBS-PSI sono giustificati dal diverso costo relativo dell'interrogazione del Bloom Filter, che è maggiore per BS-PSI (sul client) poiché contraddistinto da un costo computazionale dell'algoritmo inferiore. A differenza di quanto ottenuto sul server, l'esecuzione con l'1% di probabilità di falsi positivi permette di ottenere uno speed-up del 57% sul client per Hash-PSI, mentre causa un leggero degrado delle prestazioni con la probabilità di falsi positivi al 50%. Questo comportamento è causato, da una parte, dal costo quasi costante dell'interrogazione del Bloom Filter al variare della percentuale di falsi positivi e, dell'altra, dal minor costo dell'interrogazione del Bloom Filter rispetto alla

Algorithm	FPP	BF Weight Server	BF Weight Client
Hash-PSI	50%	33.55%	33.87%
	1%	34.26%	55.71%
DH-PSI	50%	0.08%	0.02%
	1%	0.13%	0.03%
BS-PSI	50%	0.04%	0.04%
	1%	0.06 %	1.91 %
ECDH-PSI	50%	0.03%	0.01%
	1%	0.05 %	0.01%
ECBS-PSI	50%	0.03%	0.05%
	1%	0.05%	0.28%

Tabella 4: *Costo relativo della generazione del Bloom Filter (BF Weight Server) rispetto al tempo di esecuzione totale del server, e costo relativo per l'interrogazione del Bloom Filter (BF Weight Client) rispetto al tempo di esecuzione totale del client, al variare della probabilità di falsi positivi e con una percentuale di sovrapposizione del 10%.*

sua generazione. Come conseguenza, l'utilizzo del Bloom Filter può migliorare le prestazioni del client per Hash-PSI in scenari in cui la dimensione del dataset in seguito all'applicazione del filtro è molto inferiore rispetto a quella originale.

Al fine di fornire maggiori informazioni sul costo delle operazioni di generazione e interrogazione del Bloom Filter, in Tabella 4 viene riassunto il peso percentuale di queste operazioni rispetto al tempo totale di esecuzione del ruolo relativo (server per la generazione e client per l'interrogazione). In conformità ai risultati di speed-up precedentemente discussi, osserviamo che la generazione e l'interrogazione del Bloom Filter ha un peso minimo rispetto al tempo di esecuzione totale per tutti gli algoritmi escluso Hash-PSI e, solo in parte, il client di BS-PSI. Per quanto riguarda i tempi di esecuzione in termini assoluti, come anticipato, essi sono indipendenti dall'algoritmo, mentre subiscono un leggero aumento al decrescere della percentuale di falsi positivi. Considerando dataset da 10.000 elementi e una percentuale di sovrapposizione del 10%, abbiamo registrato un tempo medio per la generazione del Bloom Filter di $29,32$ millisecondi con la probabilità di falsi positivi al 50%, e di $25,6$ millisecondi con la probabilità di falsi positivi all'1%, mentre abbiamo osservato un tempo medio per l'interrogazione del Bloom Filter di $8,16$ millisecondi con la probabilità di falsi positivi al 50%, e di $8,76$ millisecondi con la probabilità di falsi positivi all'1%.

Come osservato, il vantaggio sulle prestazioni offerto dall'utilizzo del Bloom Filter è molto significativo, soprattutto per il ruolo client degli algoritmi basati su Blind Signature. Tuttavia, scegliere o meno se utilizzare questa ottimizzazione in un determinato contesto applicativo può

essere complesso a causa delle sue implicazioni sulla privacy del server. Infatti, il client potrebbe interrogare il Bloom Filter generato dal server per ottenere informazioni su elementi non presenti nel dataset del client. In caso di esito negativo dell'interrogazione, il client ottiene la certezza che l'elemento verificato non è presente all'interno del dataset del client, mentre in caso di esito positivo, ottiene l'informazione parziale che l'elemento verificato potrebbe far parte del dataset del server. Se consideriamo l'utilizzo del Bloom Filter come componente dell'algoritmo di PSI, questo attacco non rientra nel threat model *semi-honest* discusso nella Sezione 2 ma viene generalmente riferito come *augmented semi-honest*, poiché, a differenza di quest'ultimo, prevede la possibilità dell'attaccante di modificare il proprio dataset aggiungendo, modificando o rimuovendo elementi. In particolare, il client potrebbe aggiungere al proprio dataset elementi non originariamente presenti per la porzione dell'algoritmo che riguarda l'interrogazione del Bloom Filter. Un aspetto critico di questo attacco è la possibilità di attuarlo offline senza alcuna interazione con il server, rendendone il rilevamento impossibile. Questa proprietà aumenta la praticità e l'efficienza di un attacco a forza bruta che, in caso di domini di input di dimensione ridotta, permetterebbe di escludere l'appartenenza al dataset del server di una una porzione significativa dello spazio degli input. Tuttavia, a causa delle caratteristiche del Bloom Filter, questo attacco non fornisce informazioni certe sulla presenza di un determinato elemento. Infatti, considerando un attacco a forza bruta su uno spazio degli input di soli 40 bit, supponendo dataset da un milione di elementi, una percentuale di sovrapposizione del 10% e una probabilità di falsi positivi dell'1%, l'interrogazione del Bloom Filter permetterebbe di restringere gli elementi che potrebbero essere contenuti nel dataset del server a 11 miliardi, un numero molto superiore rispetto alla dimensione dell'intersezione (100.000 elementi).

Al fine di valutare correttamente l'impatto di questo attacco, è necessario sottolineare che in uno scenario conforme al threat model *augmented semi-honest* nel quale le parti possono inserire elementi a piacere nel proprio dataset, non è possibile garantire la server privacy o la client privacy per nessuno degli algoritmi considerati. Infatti, è sufficiente che una parte inserisca gli elementi da verificare all'interno del proprio dataset ed esegua l'algoritmo di PSI (come client o come server) per ottenere risposte certe sulla sua presenza nel dataset dell'altra parte. Tuttavia, escludendo Hash-PSI, per sfruttare questa vulnerabilità è necessaria una interazione online tra le parti, mentre il Bloom Filter permette di ottenere informazioni sul dataset del server completamente offline. La vulnerabilità intrinseca degli algoritmi di PSI nel threat model *augmented semi-honest*, e possibili tecniche di rilevamento di questa tipologia di attacchi, sono discusse in maggior dettaglio del Paragrafo 6. Un ulteriore aspetto da tenere in considerazione in relazione alla vulnerabilità del Bloom Filter è associato all'utilizzo di funzioni hash differenti in generazioni successive del Bloom Filter per dataset equivalenti o simili. Infatti, ogni Bloom Filter

basato su differenti funzioni hash permette di escludere diverse porzioni dello spazio dell'input, permettendo dunque di convergere nel tempo ad un insieme di elementi sempre più simile al dataset del server. Fortunatamente, per evitare questa ulteriore criticità, è sufficiente utilizzare le stesse funzioni hash in generazioni successive del Bloom Filter.

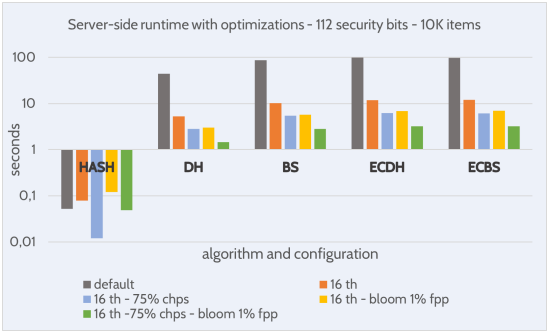
In conclusione, l'utilizzo del Bloom Filter può offrire un significativo aumento delle prestazioni a discapito di una riduzione del livello di privacy del dataset del server. Di conseguenza, scegliere se utilizzarlo o meno in un determinato contesto applicativo non è banale, e può dipendere dal livello di fiducia tra le parti, dai requisiti di riservatezza del client o dalle differenti necessità in termini di prestazioni.

5.4.7 Combinazione di ottimizzazioni

In questo paragrafo vengono confrontate tra loro alcune combinazioni delle ottimizzazioni precedentemente discusse, caratterizzate da differenti trade-off in termini di sicurezza e prestazioni, con l'obiettivo di fornire gli strumenti per determinare la configurazione ideale dipendentemente dal singolo contesto applicativo. Tali combinazioni sono mostrate in relazione alla loro configurazione base, ovvero quella in cui non è applicata nessuna ottimizzazione, con il fine di saggiare i loro benefici in termini di prestazioni.

Poiché l'uso del calcolo parallelo per ridurre i tempi di esecuzione non comporta alcuna conseguenza in termini di sicurezza o privacy, tutte le combinazioni considerate fanno uso del massimo numero di vCPU presenti sulla macchina di test. Per quanto concerne le restanti ottimizzazioni, ovvero l'utilizzo della cache e del Bloom Filter, di queste sono state esplorate differenti combinazioni. Coerentemente con quanto riportato nei capitoli precedenti, nei test la cache del server è stata pre-popolata in modo da garantire una percentuale di cache-hit del 75%; differentemente, sul client la cache è stata disabilitata simulando uno scenario in cui non si ha alcun controllo sull'ambiente di esecuzione di quest'ultimo, ovvero non si hanno garanzie sulla presenza di memoria non volatile. Per quanto riguarda il Bloom Filter, si è considerato unicamente lo scenario con la percentuale di falsi positivi pari all'1%.

In Figura 22 e Figura 23 sono riportati su scala logaritmica i tempi di esecuzione ottenuti rispettivamente con 112 e 128 bit di sicurezza, su insiemi di 10.000 elementi ed una percentuale di sovrapposizione del 10%. Analizzando l'uso del solo calcolo parallelo (barra arancione) registriamo, in linea con quanto riportato nel Paragrafo 5.4.4, un incremento medio delle prestazioni di circa otto volte per tutti gli algoritmi, sia lato client che server, fatta eccezione per Hash-PSI



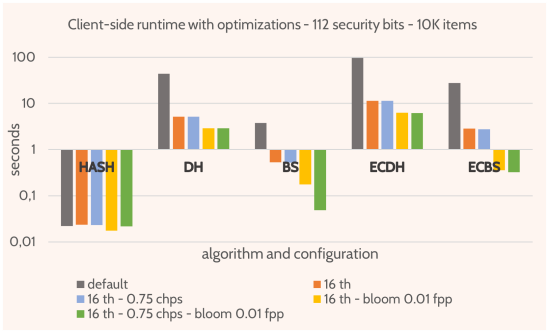
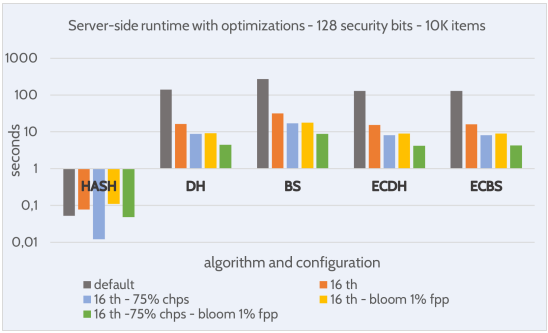


Figura 22: *Tempo di esecuzione con differenti combinazioni di ottimizzazioni, dataset da 10.000 elementi e 112 security bit.*



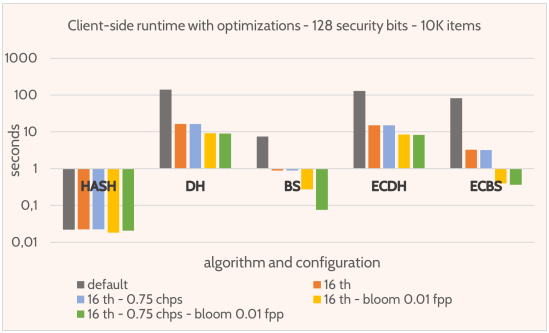


Figura 23: *Tempo di esecuzione con differenti combinazioni di ottimizzazioni, dataset da 10.000 elementi e 128 security bit.*

che subisce un lieve degrado delle prestazioni causato dall'overhead introdotto dalla gestione del multi-threading. L'aggiunta dell'utilizzo della cache sul server (barra azzurra) fornisce un beneficio aggiuntivo su tutti gli algoritmi, dimezzando ulteriormente i tempi di esecuzione rispetto alla soluzione con il solo parallelismo attivo; rimangono invece invariati i tempi dei client in cui la cache è disabilitata. Analizzando singolarmente i benefici riportati dal Bloom Filter (barra gialla), applicato sempre in combinazione con il calcolo parallelo su 16 thread, osserviamo uno scenario più complesso. Per quanto riguarda il server registriamo una riduzione delle prestazioni omogenea di circa il 43% per DH-PSI, BS-PSI, ECDH-PSI e ECBS-PSI; infatti, sebbene il Bloom Filter sia utilizzato per ridurre l'insieme di elementi su cui il client deve applicare le funzioni di oscuramento, questo ha un beneficio indiretto sul server che, di conseguenza, si trova a dover analizzare un insieme ridotto durante la seconda fase di offuscamento, ovvero quella che coinvolge l'insieme degli elementi già offuscati dal client durante la prima fase. Similmente a quanto accaduto con l'introduzione del parallelismo, Hash-PSI sul server subisce un ulteriore degrado delle prestazioni legato alla generazione del filtro stesso; ricordiamo inoltre che, essendo assente una seconda fase di processamento, mancano totalmente i benefici indiretti che caratterizzano gli altri algoritmi. Data la simmetricità degli algoritmi basati su DH, ovvero DH-PSI e ECDH-PSI, sul client registriamo una riduzione dei tempi di esecuzione similare a quella ottenuta dai rispettivi server. Gli algoritmi BS-PSI e ECBS-PSI, che invece non presentano una seconda fase di processamento dell'insieme ricevuto dal server, hanno un incremento delle prestazioni rispettivamente di 8 e 9 volte. Hash-PSI è invece quello che registra un minor beneficio lato client. Infine, andando ad analizzare la soluzione che combina tra loro parallelismo, caching e Bloom Filter (barra verde), notiamo una riduzione dei tempi di esecuzione uguale, ed a volte leggermente superiore, al prodotto delle riduzioni ottenute con le due precedenti soluzioni analizzate (16th-75% chps e 16th-bloom 1% fpp). In particolare, sul server otteniamo una riduzione dei tempi di esecuzione del 97% (30 volte) per DH-PSI, BS-PSI, ECDH-PSI ed ECBS-PSI, ed una riduzione dell'8% per Hash-PSI. Differentemente, sul client osserviamo una riduzione del 93% (15 volte) dei tempi di esecuzione di DH-PSI ed ECDH-PSI, del 98,5% (65 volte) per BS-PSI, del 98,6% (75 volte) per ECBS-PSI e del 23% (1,3 volte) per Hash-PSI.

In Figura 24 e Figura 25 sono riportati i risultati ottenuti eseguendo gli stessi test su un insieme di un milione di elementi. Senza voler scendere nel dettaglio dei singoli scenari, ritroviamo una riduzione dei tempi di esecuzione similare a quanto ottenuto precedentemente, con un miglioramento marginale (circa il 3%) legato esclusivamente alla maggior durata del test e quindi ad una riduzione della porzione di esecuzione non impattata dalle ottimizzazioni. Differentemente, per Hash-PSI osserviamo un miglioramento rispetto al precedente test, specialmente sul client, con una riduzione totale dei tempi di esecuzione del 17% (1,2 volte) sul server e dell'86%

Figura 24: *Tempo di esecuzione con differenti combinazioni di ottimizzazioni, dataset da 1.000.000 di elementi e 112 security bit.*

Figura 25: *Tempo di esecuzione con differenti combinazioni di ottimizzazioni, dataset da 1.000.000 di elementi e 128 security bit.*

(7,5 volte) sul client.

Per quanto concerne il numero di bit di sicurezza, questi non mostrano alcun impatto sull'efficacia delle ottimizzazioni mentre, in linea con quanto visto precedentemente, cambiano i rapporti tra i tempi registrati dai vari algoritmi. In particolare, sul server notiamo un netto vantaggio degli algoritmi basati su curve ellittiche al crescere del numero di bit di sicurezza. Tale vantaggio, tuttavia, non viene registrato sul client dove risultano più efficienti gli algoritmi basati su crittografia asimmetrica, specialmente nel caso di BS-PSI. Proprio quest'ultimo algoritmo ci fornisce il risultato più interessante e promettente: grazie all'uso combinato delle ottimizzazioni in esame è possibile ottenere tempi di esecuzioni del client sotto la soglia dei 7 secondi per un milione di elementi con 128 bit di sicurezza. Altrettanto interessante rimane il risultato riportato da ECBS-PSI che raggiunge i 35 secondi nello stesso scenario di esecuzione. Infatti, sebbene quest'ultimo abbia un costo maggiore per il client, esso è bilanciato dalle migliori prestazioni ottenute sul server. Questi risultati sono difformi rispetto a quanto osservato con 112 bit di sicurezza, dove BS-PSI offre migliori prestazioni della sua controparte basata su curve ellittiche per entrambi i ruoli.

5.4.8 Costo della comunicazione

Il costo della comunicazione può variare significativamente in base all'algoritmo di PSI e può dunque rappresentare un aspetto rilevante nella selezione della soluzione da utilizzare. L'effettivo impatto della comunicazione sul tempo di esecuzione può variare significativamente in base alla larghezza di banda disponibile nella comunicazione tra client e server. La larghezza di banda di 10 Gb/s offerta dall'infrastruttura Cloud utilizzata per questo studio sperimentale rende i tempi associati alla comunicazione marginali rispetto ai tempi di esecuzione. Di conseguenza, è stato considerato preferibile un approccio analitico in cui, a partire dalla dimensione in byte dei singoli elementi, viene stimato il costo della comunicazione al variare della larghezza di banda.

Definendo come set_{server} e set_{client} rispettivamente i dataset del server e del client e come k la dimensione in byte dell'output della funzione di oscuramento, è possibile calcolare la quantità di dati trasferiti tra il client e il server per DH-PSI, BS-PSI, ECDH-PSI ed ECBS-PSI nel seguente modo:

$$communication_{cryptographic} = k * dim(set_{server}) + 2k * dim(set_{client}) \tag{14}$$

Differentemente, per Hash-PSI, che non richiede l'applicazione di funzioni di oscuramento da

Algoritmo	Security Bit	Size (MB)	Time w/ 100 Mb/s (s)	Time w/ 10 Gb/s (s)
Hash-PSI	N/A	30.51	2.44	0.02
DH-PSI	112	732.42	58.59	0.58
	128	1098.63	87.89	0.87
BS-PSI	112	732.42	58.59	0.58
	128	1098.63	87.89	0.87
ECDH-PSI	112	165.93	13.27	0.13
	128	188.82	15.01	0.15
ECBS-PSI	112	165.93	13.27	0.13
	128	188.82	15.01	0.15

Tabella 5: *Dimensione e tempo della comunicazione tra client e server con dataset da un milione di elementi al variare dell'algoritmo, numero di security bit e larghezza di banda.*

parte del server sugli elementi del dataset del client, la quantità di dati che è necessario trasferire è pari a:

$$communication_{Hash-PSI} = k * dim(set_{server}) \tag{15}$$

Utilizzando queste formule, in Tabella 5 vengono stimate le dimensioni in megabyte dei dati da trasferire, e il loro tempo di trasferimento, considerando un milione di elementi per entrambi i dataset e larghezze di banda pari a 100 Mb/s e 10 Gb/s. Evidenziamo che la larghezza di banda è definita in termini di bit invece che di byte come da prassi per il settore delle telecomunicazioni. Come previsto, i risultati sono equivalenti per DH-PSI e BS-PSI e per ECDH-PSI ed ECBS-PSI a parità di security bit. Similmente a quanto discusso nel Paragrafo 5.4.5, la maggior compattezza nella rappresentazione dei valori oscurati per gli algoritmi basati su curve ellittiche permette di ottenere un costo della comunicazione significativamente ridotto per ECDH-PSI ed ECBS-PSI rispetto a DH-PSI ed BS-PSI. In termini generali, questi dati confermano il peso marginale della comunicazione rispetto ai tempi di esecuzione se si considera una larghezza di banda di 10 Gb/s. Infatti, considerando dataset da un milione di elementi, i tempi di esecuzione della componente computazionale, pur combinando le ottimizzazioni precedentemente discusse, sono almeno nell'ordine dei secondi per il client e dei minuti per il server (escluso Hash-PSI), mentre i tempi di comunicazione sono inferiori ad un secondo per tutte le configurazioni considerate. Differentemente, considerando una larghezza di banda di 100 Mb/s, che può essere considerato un lower bound per le infrastrutture moderne, il costo della comunicazione è superiore a quello computazionale solo per il client di BS-PSI con ottimizzazioni e per Hash-PSI, cioè per le esecu-

zioni caratterizzate dal minor peso computazionale. In tutti gli altri casi, il tempo associato alla comunicazione è sempre comparabile o inferiore al tempo necessario per il processamento. In questi scenari è possibile ammortizzare quasi completamente il costo della comunicazione adottando un approccio a pipeline, secondo il quale il processamento e la comunicazione vengono effettuati in parallelo su diverse porzioni dei dataset.

Un ulteriore aspetto che può influenzare il costo della comunicazione è l'utilizzo del Bloom Filter. Infatti, dipendentemente dalla percentuale di sovrapposizione e dalla probabilità di falsi positivi, il suo utilizzo può portare ad una riduzione più o meno significativa del numero di elementi del dataset del client da trasferire. Supponendo una percentuale di sovrapposizione del 10%, una probabilità di falsi positivi dell'1% e dataset da un milione di elementi, l'utilizzo del Bloom Filter permette di ridurre la quantità di dati trasferiti, e di conseguenza i tempi di comunicazione, del 54%. Questo calcolo include il trasferimento del Bloom Filter dal server al client, che tuttavia, avendo la dimensione di 1.01 MB per la configurazione considerata, ha un impatto marginale sul costo di comunicazione totale.

In conclusione, il costo della comunicazione assume un peso rilevante solo per esecuzioni con tempi di esecuzione relativamente brevi e, di conseguenza, in scenari non particolarmente critici: dovendo scegliere quale algoritmo di PSI utilizzare appare ragionevole considerare primariamente il rapporto tra il livello di sicurezza e il costo computazionale offerto dalle diverse alternative e, solo in seguito, prendere in considerazione le implicazioni sui costi di comunicazione.

6 Sicurezza e Osservabilità

Come discusso nel Paragrafo 5.4.6, il threat model *augmented semi-honest* si differenzia dal *semi-honest* per la possibilità che i partecipanti modifichino il proprio dataset—ad esempio aggiungendo o rimuovendo elementi. All'interno di questo threat model, nessuno degli algoritmi PSI considerati può garantire server privacy o client privacy. Infatti, in questo scenario, è sufficiente fingere che il proprio dataset sia pari all'intero spazio degli elementi per ottenere, come risultato dell'intersezione, l'intero insieme della controparte. Questa vulnerabilità è indipendente dall'algoritmo utilizzato, poiché è intrinseco nella definizione standard di PSI, nel quale non è prevista di un terzo partecipante del protocollo capace di validare la proprietà degli elementi inviati dalle due parti.

In questo scenario, non potendo garantire in senso assoluto la server privacy, può essere inte-

ressante delineare i confini di questa vulnerabilità ed evidenziare possibili tecniche di rilevamento di attacchi.

Infatti, pur rimanendo oscurato, entrambe le parte di un algoritmo di PSI hanno un informazioni sulla cardinalità degli insiemi. In questo scenario, l'invio di un dataset troppo ampio rispetto a quello atteso può rendere evidente questa tipologia di attacco. Di conseguenza, per poter ottenere informazioni su elementi fuori dal proprio dataset senza destare sospetto, l'attaccante deve distribuire tali interrogazioni lungo più interazioni, dilatando così in modo significativo il tempo necessario per ottenere l'intero dataset dell'altra parte. Allo stesso modo, bisogna considerare che questa tipologia di attacco risulta attuabile solo se l'insieme della controparte non varia in modo significativo e che potrebbe prolungarsi per un tempo indeterminato laddove la cardinalità dello spazio degli elementi è di vari ordini di grandezza superiore al numero di elementi presenti nell'insieme dell'attaccante.

A questo punto, possiamo ulteriormente distinguere gli algoritmi considerati in base alla loro capacità di fornire visibilità sulle azioni della controparte. Infatti, sebbene nella presente trattazione abbiamo considerato che il risultato dell'elaborazione, ovvero l'intersezione, sia calcolato solo da una delle due parti, ovvero il client, gli algoritmi DH-PSI e ECDH-PSI, grazie alla loro struttura simmetrica, offrono la possibilità di svolgere il calcolo dell'intersezione anche lato server, aggiungendo l'invio, da parte del client, del dataset del server offuscato da entrambe le parti. Anche se la conoscenza da parte del server dell'intersezione non è sufficiente ad avere visibilità sulle azioni del client, questa può essere sfruttata in concomitanza con altre informazioni provenienti dal client stesso o da altre fonti. Un esempio è quello in cui, a seguito del calcolo dell'intersezione, il client debba inviare delle informazioni al server che permettano di inferire una loro corrispondenza con l'intersezione stessa: in questo scenario, se il client invia un dataset allargato rispetto a quello realmente in possesso, con lo scopo di inferire gli elementi in possesso del server, quest'ultimo sarà in grado di verificare tale comportamento confrontando l'intersezione da lui calcolata con le informazioni successivamente ricevute dal server. Un altro scenario esemplificativo è quello in cui sono presenti molteplici client i cui elementi devono essere necessariamente disgiunti; in tale scenario, la presenza di un elemento su più intersezioni rappresenterebbe un evento anomalo imputabile ad un tentativo di attacco.

In conclusione, sebbene lo studio sperimentale abbia mostrato come BS-PSI e ECBS-PSI possano essere algoritmi prestazionalmente più promettenti quando il tempo di esecuzione del client rappresenta l'aspetto più critico, in contesti in cui il livello di fiducia tra le parti è limitato, o in scenari che richiedono un livello particolarmente elevato di riservatezza, DH-PSI e ECDH-PSI potrebbero rappresentare un'alternativa migliore.

7 Conclusioni

In questo documento sono stati analizzati differenti schemi crittografici per la realizzazione di algoritmi di Private Set Intersection. L'analisi è stata guidata dagli scenari applicativi discussi con PagoPA, andando a considerare nell'insieme di soluzioni da valutare sperimentalmente quelle che potessero fornire, anche da un punto di vista teorico, un giusto bilanciamento tra proprietà di riservatezza e prestazioni computazionali. Nell'ottica delle prestazioni sono state anche vagliate alcune ottimizzazioni, al fine di mostrare come (eventualmente sacrificando alcuni aspetti di riservatezza) fosse possibile ottenere miglioramenti nei tempi di calcolo dell'intersezione.

I risultati dell'analisi sperimentale, tenendo conto anche dei differenti trade-off tra prestazioni e privacy offerti dalle diverse ottimizzazioni proposte, non permettono di scegliere un unico algoritmo, o un'unica configurazione di questi, adeguata per qualsiasi contesto d'uso.

Ad esempio, laddove l'obiettivo principale fosse quello di minimizzare il tempo di esecuzione del client, BS-PSI si presenta come la soluzione migliore sia per 112 che 128 security bit. Tuttavia, adottando ECBS-PSI si potrebbero ridurre notevolmente i tempi di esecuzione sul server a discapito del client, i cui tempi rimarrebbero tuttavia contenuti. Differentemente, laddove l'obiettivo fosse quello di minimizzare i tempi di esecuzione del server, o aumentare il grado di sicurezza tramite meccanismi di detection, DH-PSI e ECDH-PSI potrebbero essere le soluzioni migliori rispettivamente con 112 e 128 security bit. Infine, in funzione della natura del dato—in particolare della dimensione del dominio degli elementi e dei dataset in possesso delle parti—, delle garanzie offerte dalle parti (es. presenza di memoria non volatile) e dei trade-off ammissibili tra privacy e prestazioni, è possibile decidere se adottare o meno ottimizzazioni quali caching e Bloom Filtering.

Risulta quindi evidente che, dai risultati di questo studio, non è possibile definire aprioristicamente quale algoritmo sia migliore degli altri: è necessario considerare singolarmente ogni ambito di applicazione al fine di adottare la soluzione e configurazione più adatta agli obiettivi del contesto specifico.

Riferimenti bibliografici

[1] Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 787–796. IEEE, 2010. 46

[2] Ernest F Brickell and Yacov Yacobi. On privacy homomorphisms. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 117–125. Springer, 1987. 22, 28

[3] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1243–1255, 2017. 44

[4] Don Coppersmith, Andrew M Odlzyko, and Richard Schroeppel. Discrete logarithms in gf (p). *Algorithmica*, 1(1):1–15, 1986. 21

[5] Dana Dachman-Soled, Tal Malkin, Mariana Raykova, and Moti Yung. Efficient robust private set intersection. In *International Conference on Applied Cryptography and Network Security*, pages 125–142. Springer, 2009. 44

[6] Emiliano De Cristofaro and Gene Tsudik. Practical private set intersection protocols with linear computational and bandwidth complexity. *IACR Cryptol. ePrint Arch.*, 2009:491, 2009. 34, 37

[7] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976. 21

[8] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985. 29

[9] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, June 1985. 39, 40

[10] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford university, 2009. 30, 31

[11] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984. 29

[12] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006. 25

[13] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 145–161, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. 43

[14] Florian Kerschbaum. Outsourced private set intersection using homomorphic encryption. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, pages 85–86, 2012. 44

[15] Ágnes Kiss, Jian Liu, Thomas Schneider, N Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. *Proc. Priv. Enhancing Technol.*, 2017(4):177–197, 2017. 44

[16] N Kobliz. Elliptic curve cryptography. *Math. Comput.*, 48:203–209, 1987. 22

[17] Vladimir Kolesnikov and Ranjit Kumaresan. Improved ot extension for transferring short secrets. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 54–70, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 43

[18] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung, and Christophe Wachter. Ron was wrong, whit is right. Cryptology ePrint Archive, Report 2012/064, 2012. `https://ia.cr/2012/064`. 22

[19] Catherine Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *1986 IEEE Symposium on Security and Privacy*, pages 134–134, 1986. 34

[20] Ralph C Merkle. Secure communications over insecure channels. *Communications of the ACM*, 21(4):294–299, 1978. 21

[21] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *SODA*, volume 1, pages 448–457, 2001. 41

[22] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004. 46

[23] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 515–530, 2015. 44, 46

[24] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 797–812, San Diego, CA, August 2014. USENIX Association. 44, 46

[25] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on ot extension. *ACM Trans. Priv. Secur.*, 21(2), January 2018. 44

[26] S. Pohlig and M. Hellman. An improved algorithm for computing logarithms overgf(p)and its cryptographic significance (corresp.). *IEEE Transactions on Information Theory*, 24(1):106–110, 1978. 26

[27] John M. Pollard. A Monte Carlo method for factorization. *BIT*, 15:331–334, 1975. 26

[28] Michael O Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptol. ePrint Arch.*, 2005(187), 2005. 39

[29] S. Saini, H. Jin, R. Hood, D. Barker, P. Mehrotra, and R. Biswas. The impact of hyper-threading on processor resource utilization in production applications. In *2011 18th International Conference on High Performance Computing*, pages 1–10, 2011. 58

[30] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014. 46

[31] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 24–43. Springer, 2010. 30

[32] Xiao Zhang, Ani Li, Boyang Zhang, Wenjie Liu, Xiaonan Zhao, and Zhanhuai Li. The cost performance of hyper-threading technology in the cloud computing systems. In *International Conference on Swarm Intelligence*, pages 581–589. Springer, 2016. 58