UNIVERSITÀ DI PISA
FACOLTÀ DI SCIENZE MATEMATICHE FISICHE E NATURALI

CORSO DI LAUREA IN INFORMATICA

SVILUPPO DI STRUMENTI PRIVACY PRESERVING PER DATA MINING DISTRIBUITO

Candidato: DANIELE SGANDURRA
Tutore Accademico: DOTT. SALVATORE RUGGIERI
Tutore Aziendale: DOTT.SSA FOSCA GIANNOTTI

ANNO ACCADEMICO 2003-2004

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Questa relazione è stata redatta a conclusione del tirocinio formativo effettuato presso il Consiglio Nazionale delle Ricerche di Pisa (ISTI-CNR), sullo sviluppo di strumenti di Privacy Preserving da applicarsi al Data Mining Distribuito.

Il progetto consisteva nello sviluppo di una libreria di funzioni e protocolli di comunicazione da utilizzarsi per effettuare calcoli distribuiti nell'ambito del Data Mining. L'implementazione si è basata su alcune pubblicazioni scientifiche [2] riguardanti il modello di calcolo distribuito detto *Secure Multiparty Computation* applicato al *Data Mining* distribuito.

Era richiesto lo sviluppo di codice (in linguaggio C, C++), per la realizzazione di una serie di funzioni basate sul modello SMC formanti la libreria finale e lo svolgimento di una serie di test a collaudo della libreria.

1.1 Analisi del problema

Con "Data Mining Distribuito" si intende la capacità di effettuare analisi su insiemi di basi di dati distribuite per ottenere da questi dati informazioni utili e conoscenze aggiuntive. I vantaggi derivanti da questa disciplina sono molteplici: primo fra tutti, è quello per cui aziende, enti od organizzazioni possono trarre dei benefici dall'unione delle loro basi di dati in quanto è possibile estrarre numerose informazioni di utilitità

generale proprio perché esiste una comune base di dati su cui effettuare delle analisi di tipo statistico, conoscitivo.

Spesso però questioni di privacy possono far sì che questo approccio, in cui tutti i dati sono disponibili a tutti i partecipanti del protocollo di Data Mining, non venga seguito in quanto le organizzazioni possono non vedere di buon occhio il fatto che i loro dati vengano acceduti da terze parti, anche se per scopi di comune profitto.

Ecco che quindi nasce l'esigenza di sviluppare un protocollo di comunicazione che consenta ugualmente di effettuare queste analisi e ricerche sui dati appartenenti a più entità che vogliono sì avere degli interessi nel condividere le loro informazioni, ma non vogliono compromettere la riservatezza dei loro dati.

Possiamo formulare con un esempio il problema: due banche possono decidere di comune accordo di ottenere informazioni generali aggiuntive sui loro clienti, analizzando congiuntamente la grande quantità di dati che hanno nei loro archivi. Possono cercare di capire in quali occasioni alcuni clienti chiudono il loro conto (qual è la loro età, il reddito annuo, da quanto tempo il conto è aperto) ed in seguito a quali eventi (abbassamento dei tassi di interesse, spese aggiuntive mensili per il bancomat); possono interessare informazioni legate al mondo della borsa ed alle abitudini che hanno i clienti rispetto a questa e così via. In questo modo, oltre ad offrire un servizio migliore alla loro clientela, possono anche trarre benefici economici, in quanto possono, per esempio, decidere di quanto aumentare certe tariffe in relazione al numero di persone che in conseguenza di questa azione chiudono il conto corrente, e così via.

Lo scenario è quello in cui due o più parti che hanno basi di dati confidenziali desiderino sottoporre i loro dati ad algoritmi di data mining sull'unione di questi per ottenere informazioni ritenute interessanti, senza però rivelare agli altri partecipanti ulteriori informazioni non necessarie. La priorità è quindi quella di proteggere queste informazioni, ma è altresì necessario rendere i dati accessibili per ricerche: infatti i partecipanti a questi algoritmi sono ben consapevoli di ottenere benefici reciproci grazie all'unione dei loro dati, ma nessuno desidera svelare il contenuto dei propri archivi agli altri.

1.2 Il modello della Secure Multiparty Computation

L'idea che sta alla base della Secure Multiparty Computation (SMC) è quella per cui un calcolo effettuato tra più parti è sicuro se alla fine di esso nessuno dei partecipanti conosce nient'altro eccetto il risultato di questo calcolo ed i propri dati di input ad esso. Lo scenario ideale (descritto in [7]) è quello in cui esiste una terza entità fidata a tutti i partecipanti del protocollo, a cui ognuno affida i propri dati: questa calcola la funzione su di essi ed invia il risultato a tutti le parti. Oltre a questo, il protocollo utilizzato deve fare sì che nessuno dei partecipanti possa venire a conoscenza di informazioni che non siano deducibili solo dal risultato finale e dal proprio input dell'algoritmo.

Un famoso esempio di questo problema (riportato in [11]) è il cosidetto *problema del milionario,* in cui le due parti (appunto, due milionari) vogliono sapere chi dei due sia il più ricco senza però rivelare l'ammontare dei loro beni all'altro. Una soluzione è quella in cui le due persone comunicano l'entità dei loro valori ad una terza persona che, confrontando i due valori, riuscirà a determinare chi dei due sia il più ricco e comunicherà il risultato ai due. L'affidabilità di questa soluzione è legata quindi al grado di fiducia che entrambe le persone hanno di questa terza persona, ovvero se ritengono opportuno comunicare i loro dati a questa persona perché considerata affidabile, rispettosa della riservatezza e così via. Altri esempi di questi problemi sono proposti in [8].

Quello che invece noi vogliamo ottenere sono gli stessi risultati che otterremo avendo una terza entità assolutamente fidata, senza però rendere partecipe del protocollo questa entità. Ovvero, il protocollo dovrà essere eseguito solo ed esclusivamente tra i partecipanti attivi del protocollo (quelli che sono cioè in possesso delle basi di dati) ed i risultati dovranno essere corretti senza lasciar trapelare ad ognuna delle parti nessuna informazione sui dati posseduti dalle altri parti

Ci sarà naturalmente uno scambio di dati tra i partecipanti, ma questo sarà effettuato utilizzando tecniche che non ne pregiudichino la loro riservatezza.

1.2.1 Semi-honest model e malicious-model

Un'altra distinzione che viene riportata in letteratura [3] è quella che viene fatta tra il *semi-honest model* ed il *malicious model*: nel primo caso, ogni partecipante segue le regole del protocollo ed utilizza i propri dati di input in maniera corretta ma è libero di utilizzare più tardi le informazioni di cui viene a conoscenza durante l'esecuzione del protocollo per cercare di compromettere la riservatezza dei dati degli altri partecipanti. Nel secondo caso, ogni partecipante può eseguire il protocollo in maniera arbitraria, alterando i propri dati di input ed utilizzando alcune specifiche del protocollo in maniera tale da ottenere informazioni aggiuntive sui dati posseduti dagli altri, il tutto per cercare di violare la confidenzialità dei dati delle altre parti.

Noi considereremo il primo modello in quanto rappresenta l'effettivo scenario di applicazione di cui ci stiamo occupando e ci consente altresì di rendere più efficiente l'implementazione dei protocolli di comunicazione.

In [11] esiste la soluzione generale al problema tra due parti (nel semi-honest model) che si basa sulla rappresentazione tramite circuiti della funzione da calcolare: ma per essere efficiente, la funzione deve avere una rappresentazione, tramite circuito, molto piccola, e per questo non è scalabile con i problemi di data mining, di solito di grandi dimensioni per quantità di dati di input.

1.3 Obiettivi e requisiti del progetto

Quello che vogliamo realizzare è una libreria di funzioni che ci permetta di eseguire algoritmi di Data Mining distribuito che utilizzeranno queste funzioni per ottenere la riservatezza dei dati. Ovvero, una libreria che renda possibile la comunicazione su rete tra i vari host che eseguono questi algoritmi e che garantisca ai proprietari dei dati la riservatezza di questi mentre i dati vengono analizzati dagli algoritmi e scambiati tra i vari partecipanti durante le fasi del protocollo di comunicazione. La libreria deve inoltre essere semplice da utilizzare ed i risultati che essa fornisce devono essere corretti e disponibili in tempi ragionevolmente brevi.

Quindi i requisiti fondamentali (fig. 1.1) che devono essere soddifatti dalla libreria sono:

- Correttezza dei risultati: tutti i partecipanti devono essere in possesso del risultato corretto finale ottenuto eseguendo gli algoritmi di Data Mining distribuito che utilizzano le funzioni della libreria che vogliamo realizzare..
- Non dispersione di informazioni: durante l'esecuzione degli algoritmi, solo poche informazioni aggiuntive devono venire a conoscenza dei partecipanti (dati intermedi).
- Riservatezza dei dati: i dati in possesso delle parti non devono venir acceduti da nessuno eccetto i legittimi proprietari.
- Efficienza: il protocollo composto dagli algoritmi di Data Mining distribuito e dalle funzioni formanti la libreria deve dare risultati in una misura di tempo ragionevole, anche in rapporto alla dimensione dell'input.
- Semplicità: il protocollo deve essere eseguito in maniera semplice, ovvero l'interfaccia deve essere intuitiva e comprensibile a tutte le parti.

Sarà inoltre necessario testare le funzioni realizzate per verificarne la correttezza e valutarne l'efficienza in termini di tempi di esecuzione.

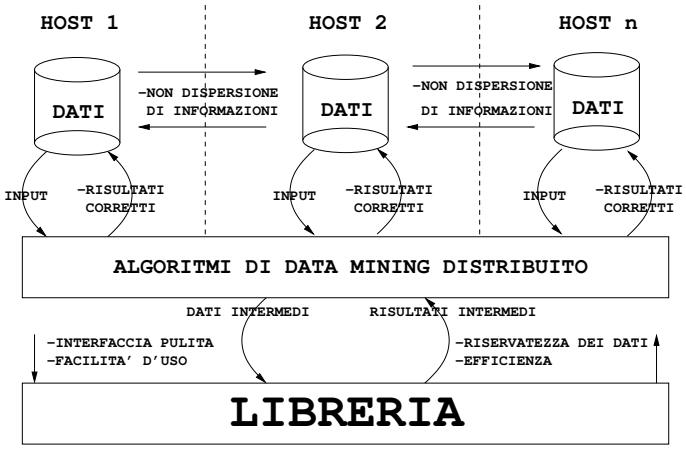

Figura 1.1: Requisiti della libreria

Capitolo 2

Progettazione

A partire dal già citato [2], la progettazione della libreria si è concentrata su quattro funzioni per i calcoli di privacy-preserving da utilizzarsi nell'ambito del data mining distribuito: queste funzioni sono in parte "insicure" dal punto di vista della SMC in quanto alcune informazioni aggiuntive (oltre ai risultati) vengono rivelate durante l'esecuzione dei calcoli. Tuttavia queste informazioni sono minime (lo vedremo più avanti nel dettaglio). I metodi qui descritti sono inoltre efficienti, nel senso che l'aggiunta delle parti di privacy-preserving all'interno delle funzioni non aumentano in maniera significativa il costo della computazione.

2.1 La libreria

La libreria è composta di quattro funzioni:

- Secure Sum
- Secure Set Union
- Secure Size of Set Intersection
- Scalar Product

Queste quattro funzioni, che risolvono problemi specifici, possono essere combinate insieme per risolvere problemi più generali: l'idea che sta alla base del progetto è

cioè quella di avere delle primitive, perfettamente funzionanti, che risolvano particolari problemi e che una loro specifica combinazione risolva problemi più generali, in particolare quelli inerenti al Date Mining distribuito.

2.2 Descrizione delle funzioni

2.2.1 Secure Sum

Spesso accade, in problemi di Data Mining distribuito, di voler ottenere la somma di valori dalle singole parti: per esempio, ci può essere bisogno di calcolare il supporto di una regola associativa (vedere a pag.13 per una lista di definizioni di termini inerenti il Data Mining distribuito), sommando il supporto di ogni sito. Ovvero, supponendo di avere s siti che partecipano al protocollo, numerati da 1 a s, quello che vogliamo ottenere è $v = \sum_{i=1}^{s} v_i$ dove v_i è il valore dell'i-esimo sito e v è noto essere compreso in un intervallo $[0...n]$.

Uno dei siti è nominato il *master site* ed è il numero 1. Gli altri siti sono ovviamente numerati da 2 ad s. Quello che avviene è che il *master site* genera un numero casuale R che è scelto uniformementa tra $[0...n]$. A questo punto il sito 1 aggiunge questo numero al suo valore v_1 e manda il risultato *mod n* al sito 2. Questi aggiungerà a questo valore il proprio valore v_2 ed invierà il risultato *mod n* al sito 3 e così via, sino a che il sito s invierà il risultato del valore ricevuto dal sito $s-1$ sommato a v_s *mod n* al sito 1, che alla fine, conoscendo R, sottrarrà questo valore per ottenere il risultato della somma, inviando questo numero agli altri siti (fig. 2.1).

Poiché il valore R compreso uniformemente tra $[0...n]$, il sito 2 non ottiene nessuno informazione aggiuntiva sul valore di v_1, dato che sia v_1 che $(R+v_1)$ *mod n* sono entrambi distribuiti uniformemente in $[0...n]$ ed alla stessa maniera gli altri siti non ottengono informazioni aggiuntive sui valori appartenenti agli altri partecipanti al protocollo. Naturalmente il numero di siti deve essere maggiore o uguale a tre, in quanto se ci fossero solo due siti il valore dell'altro sarebbe banalmente ottenibile tramite sottrazione tra il risultato finale ed il proprio valore.

A differenza di quanto riportato in [2], in cui è contemplata anche l'eventualità che

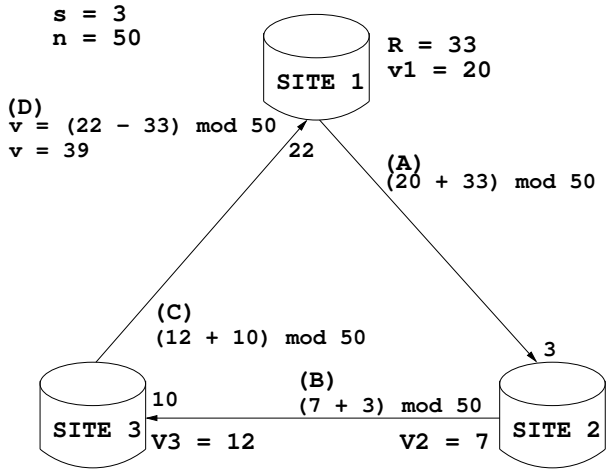

Figura 2.1: Esempio di secure sum

il valore possa essere un numero negativo, abbiamo deciso per semplicità di limitare il valore dei v_i ai soli numeri positivi: tuttavia questo non comporta nessuna limitazione perché, se questa funzione viene utilizzata per calcolare il supporto generale di un insieme di elementi, tale valore verrà utilizzato per indicare il supporto di tale insieme in termini di numero di transazioni che contengono tale insieme e quindi questo numero sarà sempre non negativo.

2.2.2 Secure Set Union

In alcuni casi nel Data Mining distribuito è utile sapere quali sono gli insiemi frequenti, quali sono le regole associative più numerose, senza sapere però a chi appartengono questi insiemi, regole. Ovvero, ci chiediamo qual è l'unione di un insieme di dati apparteneti a più siti, senza però sapere a chi appartengono questi dati. Un approccio per risolvere questo problema si basa sulla *crittografia commutativa* (si veda a questo proposito l'appendice A): si parla di crittografia commutativa se, dato un messaggio e più di una chiave di crittatura, l'ordine con cui viene crittato il messaggio con queste chiavi è ininfluente ai fini del risultato della crittazione, perché il crittogramma finale sarà sempre lo stesso qualunque sia l'ordine con cui il messaggio viene crittato utilizzando le chiavi.

In questa maniera, la crittografia commutativa ci dà il vantaggio di poter crittare gli

elementi in qualsiasi ordine e di ottenere sempre lo stesso risultato: questo è utile in quanto sarà possibile confrontare se due elementi sono uguali, crittandoli in un qualsiasi ordine e verificando che il risultato finale sia lo stesso. Quello che avviene, in particolare, è che ogni sito critta i propri elementi ed in seguito critta gli elementi che riceve (crittati) dagli altri siti. Alla fine, grazie alla proprietà sopra descritta della crittografia commutativa, si avrà un insieme di elementi, alcuni dei quali saranno uguali e sarà quindi possibile rimuovere i duplicati ed ottenere l'unione degli elementi. Per sapere il valore di questi elementi, basterà eseguire in un qualsiasi ordine la loro decrittazione da parte di tutti i siti.

Ovvero in sintesi (fig. 2.2):

1. Ogni sito critta i propri elementi con la propria chiave.

2. Il sito 1 invia i propri elementi crittati al sito 2, che provvederà a crittarli; il sito 2, dal canto suo, invierà i suoi elementi crittati al sito 3, che li critterà; analogamente per il sito 3 e così via.

3. Si ripete il passo precedente utilizzando gli elementi ricevuti dal sito precedente e che vengono crittati ed inviati al successivo dopo aver effettuato una permutazione casuale di tali elementi, sino a che tutti gli elementi di tutti i siti sono stati crittati da tutti i partecipanti.

4. Un sito (per esempio il primo) riceve tutti gli elementi crittati da tutti, rimuove i duplicati, e decritta tutti gli elementi con la propria chiave e li invia al sito successivo.

5. Questo sito provvederà a decrittare tutti gli elementi, inviandoli al sito successivo e così via, fino a che l'ultimo sito farà la decrittatura di tutti gli elementi, ottenendo il risultato in chiaro.

6. L'ultimo sito invia il risultato a tutti.

Questo algoritmo rivela però il numero di elementi che appartengono a più siti (se alla fine della prima fase di crittatura appare un elemento duplicato k volte significa che k siti posseggono lo stesso elemento), senza specificare però quali elementi siano (infatti

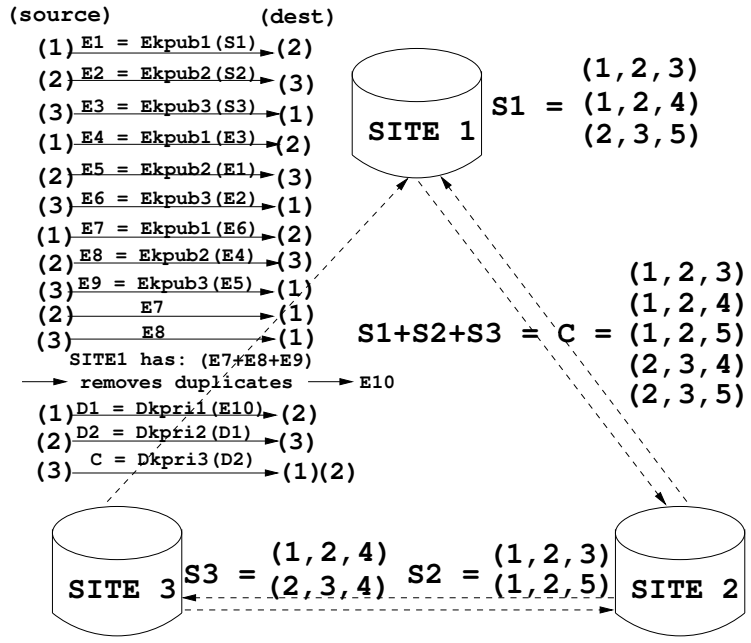

Figura 2.2: Esempio di secure union

sono crittati). Questa è una di quelle informazioni aggiuntive, a cui ci riferivamo prima, che fanno sì che questo algoritmo dal punto di vista della SMC non sia "sicuro". Ma questo è l'unico prezzo da pagare per avere un algoritmo corretto ed efficiente: altre soluzioni sono sicure, sempre secondo il modello SMC, ma non sono efficienti (in termini di tempo di esecuzione).

2.2.3 Secure Size of Set Intersection

Se più parti hanno ognuna un insieme di elementi, tutti appartenenti ad uno stesso dominio, può essere utile calcolare la cardinalità dell'intersezione di questi insiemi. Cioè, dati k partecipanti ognuno avente l'insieme S_i, con i in $[1...k]$, quello che vogliamo calcolare in maniera sicura è $|S_1 \cap .... \cap S_k|$.

Anche in questo caso usiamo la crittografia commutativa, come per la secure union: il protocollo è pressoché identico, eccetto che il sito che riceve l'ultima volta tutti gli elementi crittati, invece di rimuovere i duplicati conta il numero degli elementi il cui numero di duplicati è uguale al numero totale dei partecipanti al protocollo (ciò

significa che quell'elemento è posseduto da tutte le parti).

In sintesi:

1. Ogni sito critta i propri elementi con la propria chiave.

2. Il sito 1 invia i propri elementi crittati al sito 2, che provvederà a crittarli; il sito 2, altresì, invierà i suoi elementi crittati al sito 3, che li critterà; analogamente per il sito 3 e così via.

3. Si ripete il passo precedente utilizzando gli elementi ricevuti dal sito precedente e che vengono crittati ed inviati al successivo dopo aver effettuato una permutazione casuale di tali elementi, sino a che tutti gli elementi di tutti i siti sono stati crittati da tutti i partecipanti

4. Un sito (per esempio il primo) riceve tutti gli elementi crittati da tutti, conta quanti elementi appaiono duplicati in numero uguale al numero delle parti ed invia questo risultato a tutti.

2.2.4 Secure Scalar Product

Supponiamo di avere due siti, ognuno con un vettore booleano di elementi (dove l'i-esima posizione ha valore 0, se l'elemento i non esiste ed 1 altrimenti): questi vettori sono della stessa cardinalità n e ad ogni posizione specifica del vettore corrisponde un elemento specifico del dominio degli elementi. Per sapere il numero esatto degli elementi che i due siti hanno in comune, basta eseguire il prodotto vettoriale dei due vettori: infatti, chiamando A il primo sito e B il secondo, se A possiede il vettore $\vec{X} = (x_1, ..., x_n)$ e B il vettore $\vec{Y} = (y_1, ..., y_n)$, il prodotto vettoriale tra X ed Y, cioè $\sum_{i=1}^{n} x_i * y_i$, darà come risultato il numero esatto degli elementi in comune. Infatti se e solo se l'i-esimo elemento appartiene sia ad A che B il prodotto scalare dell'i-esima componente del vettore darà 1 e quindi la somma di questi valori ci darà il risultato finale. Per ottenere lo stesso risultato, ma in maniera sicura, quello che dobbiamo fare è utilizzare matrici random di dimensioni $n*n$ e vettori random di dimensione n in modo da nascondere i valori reali dei vettori originali. Di seguito l'algoritmo utilizzato (per una trattazione più dettagliata si veda [10]):

1. A e B si mettono d'accordo in comune su una matrice C di cardinalità $n * n$.
2. A genera un vettore random $\vec{R}$ di dimensione n.
3. A calcola $\vec{X'} = C \times \vec{R}$.
4. A calcola $\vec{X''} = \vec{X} + \vec{X'}$.
5. A invia $\vec{X''}$ a B.
6. B calcola $S' = \vec{X''} \times \vec{Y}$.
7. B calcola $\vec{Y'} = C^T \times \vec{Y}$.
8. B genera un vettore random $\vec{R'}$, di dimensione r, con $r < n$.
9. B calcola il vettore $\vec{Y'}$, di dimensione n, dove le prime $\frac{n}{r}$ componenti sono ottenute sommando $\vec{Y'_1}$ a $\vec{R}_1$, le seconde $\frac{n}{r}$ componenti si ottengono sommando $\vec{Y'_2}$ a $\vec{R}_2$ e così via.
10. B invia S' e $\vec{Y'}$ ad A.
11. A conosce $S' = \vec{X''} \times \vec{Y}$ e lo riscrive come $S' = \vec{X} \times \vec{Y} + S'' - S'''$.
12. A calcola $S'' = \vec{R} \times \vec{Y'}$
13. A si calcola un vettore $\vec{R''}$ di cardinalità r, in cui la prima componente è data dalla somma delle prime $\frac{n}{r}$ componenti di $\vec{R}$, la seconda componente è data dalla somma delle seconde $\frac{n}{r}$ componenti di $\vec{R}$ e così via.
14. A invia $\vec{R''}$ e S'' a B.
15. B calcola $S''' = \vec{R'} \times \vec{R''}$.
16. B calcola il risultato finale $\vec{X} \times \vec{Y} = S' - S'' + S'''$.
17. B invia il risultato ad A.

2.3 Applicazioni

A questo punto, avendo le suddette funzioni, è possibile realizzare degli algoritmi di Data Mining distribuito: in particolare, utilizzeremo una versione dell'algoritmo *apriori* distribuito chiamato FDM (Fast Distributed Algorithm) [1].

2.3.1 Fast Distributed Algorithm

Prima diamo una lista di definizioni utili per comprendere l'algoritmo.

- $I = i_1, i_2, ..., i_n$, è un insieme di elementi.
- *DB* è un insieme di transazioni dove ogni transizione T è un insieme di elementi tali che $T \subseteq I$.
- Dato un insieme di elementi $X \subseteq I$, si dice che una transazione T contiene X se e solo se $X \subseteq T$.
- Una *regola associativa* è una implicazione della forma $X \Rightarrow Y$, dove sia X che Y sono contenuti in I (es.: *compra(X, "computer")* $\Rightarrow$ *compra(X, "pacchetto_software_per_ufficio")*).
- La regola $X \Rightarrow Y$ ha *supporto* s in *DB*, se s% delle transazioni presenti in *DB* contengono $X \cup Y$; la regola $X \Rightarrow Y$ ha *confidenza* c se il c% delle transazioni in *DB* che contengono X contengono anche Y (es.: *computer* $\Rightarrow$ *pacchetto_software_per_ufficio [supporto=2%, confidenza=60%]*).
- Un insieme X di k elementi viene detto un $k-itemset$ (es.: $\{I1, I2, I3\}$ e $\{I2, I5, I7\}$ sono $3-itemset$).
- Se ci sono n siti ($S_1 .... S_n$), la base di dati *DB* sarà suddivisa tra gli n siti in modo che se DB_i appartiene al sito i allora $DB = DB_1 \cup DB_2 ... \cup DB_n$.
- L'insieme X ha un *supporto locale* di $X.sup_i$ al sito S_i, se $X.sup_i$ delle transazioni contengono X.

- Il *supporto globale* di X è dato come $X.sup = \sum_{i=1}^{n} X.sup_i$.
- Un insieme di elementi X è supportato globalmente se $X.sup \geq s \times (\sum_{i=1}^{n} |DB_i|)$.
- La *confidenza globale* di una regola $X \Rightarrow Y$ può essere calcolata come $\frac{X \cup Y.sup}{X.sup}$.
- L'insieme dei *large itemset* $L_{(K)}$ consiste in tutti quei k-itemset che sono supportati globalmente.
- L'insieme dei *locally large itemset* $LL_{i(k)}$ consiste di tutti i k-itemset che sono supportati localmente al sito S_i.
- L'insieme dei *globally large k-itemset* supportati localmente al sito S_i è detto $GL_{i(k)}$, che è uguale all'intersezione di $L_{(K)}$ con $LL_{i(k)}$.

Questo, in breve, l'algoritmo (nel caso i dati siano partizionati orizzontalmente):

1. **Candidate Sets Generation**: si generano gli insiemi candidati $CG_{i(k)}$ basandoci su $GL_{i(k-1)}$, ovvero l'nsieme dei globally large itemsets supportati da S_i alla iterazione $(k-1)$, usando l'algoritmo apriori. Ogni sito genera i candidati basandosi sull'intersezione dei globally large $(k-1)$-itemset e dei locally large $(k-1)$-itemset.
2. **Local pruning**: Per ogni X appartenente a $CG_{i(k)}$, il sito S_i conta nel DB_i il numero di occorrenze di X per ottenere $X.sup_i$. Se X è locally large in S_i, viene incluso nell'insieme $LL_{i(k)}$.
3. **Support Count Exchange**: viene fatto un broadcast degli $LL_{i(k)}$, in modo che ogni sito si calcoli il supporto locale degli elementi in $\cup_i LL_{i(k)}$.
4. **Broadcast Mining Results**: ogni sito invia a tutti gli altri il proprio supporto locale per gli insiemi di elementi in $\cup_i LL_{i(k)}$. Da questi, ogni sito può calcolarsi $L_{(k)}$, e si ripete l'algoritmo fino a che ci sono insiemi supportati globalmente.

L'algoritmo può così essere utilizzato aggiungendo le funzioni di privacy preserving per renderlo sicuro (secondo l'accezione data dalla SMC), nei casi in cui i dati siano partizionati orizzontalmente e verticalmente.

2.3.2 Dati partizionati orizzontalmente

In questo scenario, le transazioni sono suddivise tra n siti, ed ogni sito avrà una lista di transazioni complete. Se l'obiettivo è quello di rendere l'algorimo precedente privacy-preserving, nel terzo punto del FDM, cioè nel Support Count Exchange, la funzione di Secure Union può essere utilizzata per ottenere $\cup_i LL_{i(k)}$. E la funzione della Secure Sum può essere utilizzata per ottenere la somma dei support count locali al quarto punto e la somma delle transazioni in *DB* in modo da calcolare il supporto globale. In [6] è descritto nel dettaglio questo procedimento.

2.3.3 Dati partizionati verticalmente

In questo caso ogni sito contiene tutte le transazioni, ma di queste ha solo una parte degli elementi (e non esistono intersezioni di elementi tra i siti), in modo che per ottenere una transazione bisogna fare l'unione delle transazioni presenti su tutti i siti.

Se le transazioni vengono viste come una matrice di valori booleani (0 oppure 1), indicanti l'assenza o presenza di un elemento, si può constatare che per ottenere il support count di un insieme di elementi basta utilizzare la funzione Secure Scalar Product (cioè, in altre parole, eseguire il prodotto vettoriale delle "colonne" rappresentanti l'insieme degli elementi all'interno delle transazioni).

Un altro modo per ottenere la stessa cosa, è quello che ogni sito rappresenti con un insieme S_i solo le transazioni che supportano i sotto-elementi dell'insieme: in questa maniera, la funzione Secure Size of Set Intersection può essere utilizzata per calcolare la cardinalità dell'intersezione di tutti gli S_i, ovvero il support count dell'insieme (infatti, se una transazione contiene quell'insieme l'intersezione darà quell'elemento, in quanto ogni sito supporterà quella regola, viceversa la transazione non sarà considerata poiché almeno un sito non supporterà tale insieme).

2.4 Scelte progettuali

Una delle scelte che abbiamo preso in fase di progettazione, è stata quella di definire la modalità con cui i vari host si potessero identificare come host "0", host "1" e così via,

in quanto le funzioni della libreria spesso fanno riferimento ad un ordinamento tra le varie parti. Abbiamo deciso allora di affidare questo ordinamento in base all'indirizzo IP degli host: così, avendo ogni macchina un indirizzo IP diverso, è possibile decidere chi sia il primo host in questa lista semplicemente controllando chi ha l'indirizzo IP più basso [1]; il secondo della lista, sarà quello il cui indirizzo IP viene immediatamente dopo e così di seguito per tutti gli altri host. Questo impone che a tutti i partecipanti del protocollo sia consegnata, prima di cominciare l'algoritmo, una lista (la stessa per tutti) contenente gli indirizzi IP di tutti gli host che partecipano al protocollo. Ognuno degli host ordinerà questa lista ed avrà così ottenuto la lista ordinata per indirizzo IP degli host che eseguono il protocollo. In questa maniera non esiste nessun sito che in precedenza deve essere nominato "master site": tutti i partecipanti al protocollo utilizzano le funzioni formanti la libreria in maniera identica e saranno solo tali funzioni che decideranno (in base all'indirizzo IP) l'ordinamento degli host all'interno del protocollo (ovvero, non ci sarà nessun sito che invocherà le funzioni in maniera differente dagli altri in quanto "master site").

Per le funzionalità relative alla crittografia commutativa, un algoritmo che ci è sembrato adatto a tale scopo e che inoltre è ben documentato ed affidabile, è l'algoritmo *RSA* (per l'utilizzo di RSA per la crittografia commutativa si veda l'appendice A) che verrà utilizzato per crittare e decrittare gli elementi scambiati tra gli host all'interno del protocollo di comunicazione.

Un'altra scelta che è stata fatta, è stata quella di utilizzare tecnologie per rendere il canale di comunicazione tra gli host sicuro: ovvero, una tecnologia che garantisse la *confidenzialità*, cioè le comunicazioni sono cifrate e quindi intelligibili solo ai legittimi destinatari e l'*affidabilità* delle comunicazioni, ovvero i dati non possono essere alterati; inoltre questa tecnologia deve offire l'*autenticazione* dei messaggi, in modo che sia possibile accertare l'identità degli host partecipanti alla comunicazione.

A questo punto, avendo bisogno di una libreria accessoria che ci permettesse di avere funzionalità di crittografia (per esempio, per la funzione Secure Union), funzioni per generare numeri random (per esempio, per la Secure Sum) e funzioni per creare un

[1] ricordiamo che i byte più importanti di un indirizzo IP sono quelli a sinistra, per cui il confronto viene fatto prima sul byte più a sinistra poi, in caso di uguaglianza sul secondo e così via.

canale sicuro (per quanto detto sopra), abbiamo indirizzato la nostra scelta sulla libreria *OpenSSL*, una libreria robusta, *opensource* (ovvero i cui sorgenti sono liberamente disponibili, anche per eventuali modifiche), ed in grado di offrirci tutte le funzionalità necessarie per la nostra libreria (si consulti l'appendice B per maggiori dettagli su OpenSSL)

Abbiamo inoltre deciso di definire poche funzioni, semplici da utilizzare, rendendo così più pulita la libreria e facendo anche in modo (per quanto detto in precedenza riguardo l'ordinamento degli indirizzi IP) che tutti gli host che eseguono tali funzioni le invochino in maniera identica, facilitandone così ulteriormente l'utilizzo.

Abbiamo inoltre deciso di utilizzare, per la codifica della libreria, il linguaggio C, usando la sintassi ANSI, in modo da permetterne la compilazione sotto qualsiasi sistema che abbia tra i propri strumenti un compilatore conforme allo standard ANSI.

2.5 Architettura del sistema

Come abbiamo detto, il nucleo della libreria è formato dalle quattro funzioni di privacy-preserving. Oltre a queste sono però necessarie anche altre funzioni ausiliarie che ci permettano di:

- Ordinare gli indirizzi IP dei partecipanti ai protocolli: tutte le funzioni della libreria avranno bisogno della lista ordinata degli IP per stabilire l'ordine di invio dei messaggi all'interno dei protocolli di comunicazione.

- Di creare un canale sicuro: per questo ci affideremo ad OpenSSL, che implementa il protocollo *SSL*; inoltre realizzeremo uno script che servirà per generare i certificati e le chiavi private che andranno consegnati ai vari host perché saranno utilizzati da OpenSSL per la gestione del canale SSL.

- Di generare numeri random, necessari per la Secure Sum.

- Di effettuare operazioni matematiche su matrici e vettori, necessarie per la Secure Scalar Product.

- Di permutare l'ordine di stringhe all'interno di un array, necessario in fase di invio per le funzioni di Secure Union e Secure Size of Set Intersection.
- Di rimuovere i duplicati all'interno di un array di stringhe: funzione necessaria alla Secure Union, nella fase finale del protocollo in cui si esegue l'unione degli elementi.
- Di calcolare la cardinalità dell'intersezione di un insieme di stringhe: funzione necessaria alla Secure Intersection, nella fase finale del protocollo in cui si calcola il risultato dell'intersezione.
- Di crittare e decrittare i dati: anche in questo caso OpenSSL ci fornisce delle funzioni che implementano l'algoritmo RSA; questi funzioni andranno però modificate per rendere possibile la crittografia commutativa.

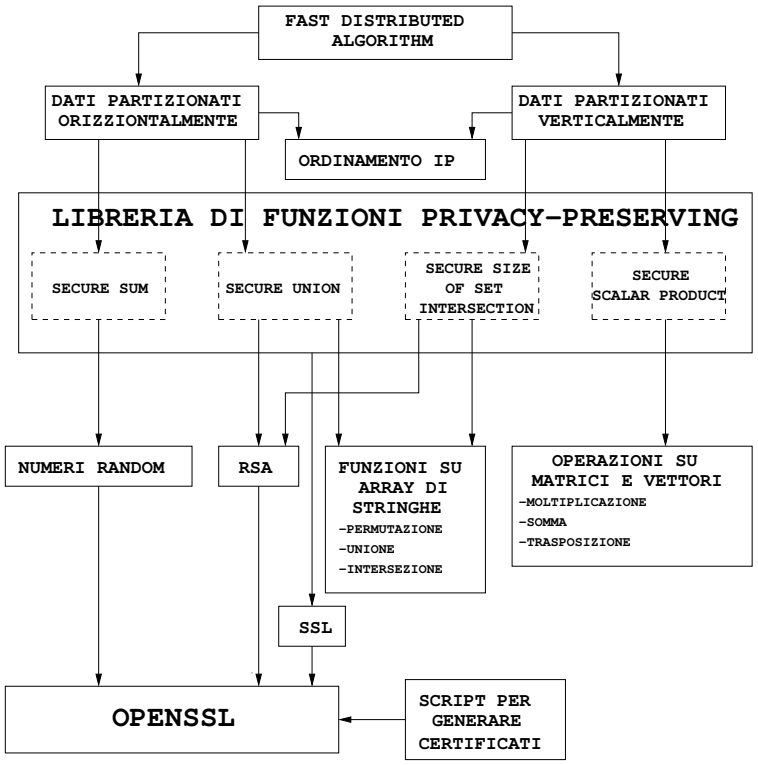

Figura 2.3: Architettura della libreria

Capitolo 3

Realizzazione

A partire dalle scelte progettuali fatte, è quindi cominciata la fase di codifica che si è concentrata sulla definizione dei prototipi delle funzioni principali, sulla scelta ed implementazione delle funzioni accessorie (per esempio, per ordinare una lista di indirizzi IP, per creare le strutture contenenti i parametri TCP/IP degli host), sulla scrittura di funzioni legate ad SSL (come quelle per inizializzare OpenSSL, per settare i parametri di invio e di ascolto per il canale SSL) e RSA (quali quelle per generare chiavi commutative, per leggere e salvare le chiavi pubbliche e private su file), ed infine sulla realizzazione vera e propria delle quattro primitive privacy-preserving e delle funzioni utilizzate da queste per inviare e ricevere i dati durante le fasi del protocollo di comunicazione.

3.1 Tecnologie utilizzate

La fase di codifica è avvenuta utilizzando strumenti di editing tra i quali *Vi* ed *Emacs*: per questa fase e per quella di testing sono state utilizzate esclusivamente macchine *Linux*, facenti parte del cluster presente al CNR. Come compilatore abbiamo utilizzato il programma *gcc* (GNU C Compiler), mentre l'utility *make* è stata utilizzata per ricompilare il codice solo quando necessario, e per definire le regole di compilazione (tramite *Makefile*).

3.2 Dettaglio delle funzioni

Qui di seguito verranno presentate tutte le funzioni implementate durante la fase di realizzazione.

3.2.1 Struttura generale del codice

La struttura del codice (si veda anche la figura 3.1) è formata dai file:

```
rsa.c  secure.c  ssl.c
rsa.h  secure.h  ssl.h
```

dei quali, "secure.c" contiene la libreria vera e propria, "ssl.c" contiene alcune funzioni legate ad OpenSSL, e "rsa.c" ha al suo interno tutte quelle funzioni che permettono di generare chiavi commutative, di crittare e decrittare; gli altri file sono gli header contenenti i prototipi delle medesime funzioni (eccetto alcune funzioni ausiliarie). Inoltre è presente la sottodirectory "certificate" contenente i file:

```
create_cert.sh  ip.txt  openssl.cnf
```

il primo dei quali viene utilizzato per creare i certificati (vedi più avanti) utilizzando il file di configurazione "openssl.cnf"; il file "ip.txt" contiene gli indirizzi IP dei partecipanti al protocollo distribuito (questo file viene utilizzato sia dalle funzioni per la privacy preserving, sia dallo script "create_cert.sh"). È presente anche il file:

```
Makefile
```

che permette la compilazione del codice utilizzando il programma "Make".

3.2.2 Il file ssl.c

Cinque sono le funzioni qui presenti:

```
void print_ssl_error(char *msg);


void ssl_init();
```

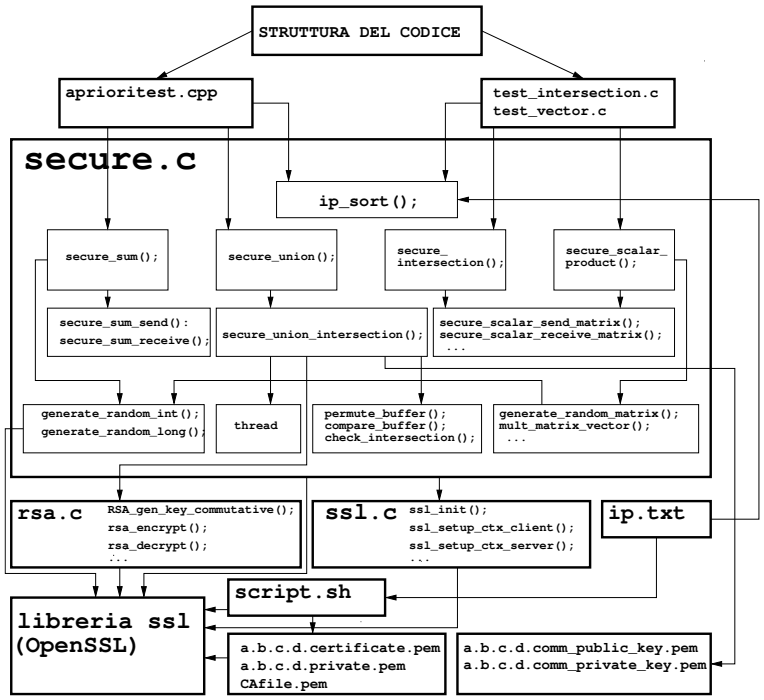

Figura 3.1: Dettaglio del codice

```
void ssl_end();

SSL_CTX* ssl_setup_ctx_client(char *certfile,
            char *certificate, char *privkey);

SSL_CTX *ssl_setup_ctx_server(char *certfile,
            char *certificate, char *privkey);
```

la prima funzione serve per stampare a video messaggi di errore legati ad OpenSSL (connessione non riuscita, certificati RSA non validi, etc); la seconda funzione viene chiamata per inizializzare alcuni parametri di OpenSSL (algoritmi utilizzati, caricamento di stringhe di errore); la terza serve per liberare la memoria occupata da alcune variabili create da OpenSSL; le ultime due funzioni servono per generare un contesto che OpenSSL utilizza per creare un canale SSL da utilizzare per connettersi ad un server (la prima) o altrimenti per rimanere in ascolto di connessioni SSL (si consiglia di vedere l'appendice B per ulteriori chiarimenti): i parametri identificano in ordine,

il file contenente i certificati di tutti gli host, il file contenente il certificato (con la chiave pubblica) dell'host che esegue la chiamata ed il file contenente la chiave privata dell'host. Questi parametri specificano ad OpenSSL quali sono gli host "fidati" (cioè, i cui certificati sono inseriti nell'elenco); inoltre fanno sì che sia lato client che server si richieda sempre l'emissione di un certificato per effettuare l'autenticazione da entrambi i lati.

3.2.3 Il file create_cert.sh

La sottodirectory:

```
certificate
```

viene utilizzata dalla libreria per gestire i certificati. Per creare i certificati, viene utilizzato lo script-file:

```
create_cert.sh
```

che a partire da un file di indirizzi IP genera una serie di chiavi pubbliche/private ed i relativi certificati, creando inoltre un file contenente tutti i certificati (l'insieme delle chiavi pubbliche) degli altri host. Quello che fa questo script (che dovrà essere eseguito su una macchina "fidata" da tutti, come una Certification Authority, alla quale, per esempio, i rappresentanti dei vari host possono delegare la generazione dei certificati, che verranno consegnati loro, per esempio, tramite floppy) è la generazione di una coppia di file per ogni host della forma:

```
A.B.C.D.certificate.pem
```

e:

```
A.B.C.D.private.pem
```

dove A.B.C.D è l'indirizzo IP dell'host, e "certificate" indica che si tratta del certificato (la chiave pubblica ed informazioni relative all'host), mentre "private" indica che si tratta della chiave privata; l'estensione ".pem" è il formato (standard) utilizzato da OpenSSL per salvare i certificati. Inoltre lo script genera un file, uguale per tutti gli host, chiamato:

```
CAfile.pem
```

contenente il certificato di tutti gli host, in questo caso usando un formato che ne facilita la lettura e la comprensione.

Quindi, supponendo di avere un file chiamato

```
ip.txt
```

contenente i seguenti indirizzi IP (non occorre che siano ordinati):

```
192.168.1.1
192.168.1.2
192.168.1.3
192.168.1.4
192.168.1.5
192.168.1.6
192.168.1.7
```

eseguendo da shell il comando:

```
>create_cert.sh ip.txt
```

verranno mostrate a video una serie di domande (che servono per creare i certificati con i parametri corretti, per esempio il nome dell'host ed altri parametri identificativi del client), risposte alle quali si avranno nella directory i seguenti file:

```
192.168.1.1.certificate.pem  192.168.1.4.private.pem
192.168.1.1.private.pem      192.168.1.5.certificate.pem
192.168.1.2.certificate.pem  192.168.1.5.private.pem
192.168.1.2.private.pem      192.168.1.6.certificate.pem
192.168.1.3.certificate.pem  192.168.1.6.private.pem
192.168.1.3.private.pem      192.168.1.7.certificate.pem
CAfile.pem
create_cert.sh*
openssl.cnf
ip.txt
```

rappresentanti le coppie di chiavi da consegnare agli host specifici (in base all'indirizzo IP), ed il file "CAfile.pem" contenente i certificati pubblici di tutti gli host, che andrà consegnato a tutti. Un esempio di come potrebbe apparire il file CAfile.pem dopo l'esecuzione dello script potrebbe essere il seguente (solo la prima parte viene mostrata):

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 0 (0x0)
        Signature Algorithm: md5WithRSAEncryption
        Issuer: C=IT, ST=Italia, L=Pisa, O=Universita' di Pisa,
        OU=Informatica, CN=PC-Laboratorio/emailAddress=sgandurr@cli.di.unipi.it
        Validity
            Not Before: Jun 15 19:15:00 2004 GMT
            Not After : Mar 12 19:15:00 2007 GMT
        Subject: C=IT, ST=Italia, L=Pisa, O=Universita' di Pisa,
        OU=Informatica, CN=PC-Laboratorio/emailAddress=sgandurr@cli.di.unipi.it
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (512 bit)
                Modulus (512 bit):
                    00:e6:34:2c:e9:8a:d9:24:4f:ef:07:99:1e:35:fa:
                    07:85:15:3a:ea:d9:fe:9d:b3:3f:52:ff:3d:73:40:
                    c3:e6:b0:6f:af:07:85:1a:e9:02:cf:41:42:6d:db:
                    7c:72:df:a4:58:32:36:f4:72:72:23:b1:d6:91:a1:
                    6d:6c:2a:d2:71
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Subject Key Identifier:
            25:E8:76:DC:63:82:F2:C0:31:B5:89:F8:95:6E:8E:40:49:BD:F8:38
            X509v3 Authority Key Identifier:
            keyid:25:E8:76:DC:63:82:F2:C0:31:B5:89:F8:95:6E:8E:40:49:BD:F8:38
            DirName:/C=IT/ST=Italia/L=Pisa/O=Universita' di Pisa
            /OU=Informatica/CN=PC-Laboratorio/emailAddress=sgandurr@cli.di.unipi.it
            serial:00

            X509v3 Basic Constraints:
            CA:TRUE
    Signature Algorithm: md5WithRSAEncryption
        1a:05:3d:00:30:2d:43:e0:8e:e2:fe:f6:52:0f:d8:b2:da:98:
        00:67:e3:89:5b:4e:3c:a4:ed:fd:6d:2f:dc:c4:47:54:d7:b2:
        c5:94:72:f8:7e:37:37:10:e8:78:59:a7:c5:c5:de:b5:6a:4f:
        10:7c:c9:d7:d5:f0:d6:d4:53:81
-----BEGIN CERTIFICATE-----
MIIDPzCCAumgAwIBAgIBADANBgkqhkiG9w0BAQQFADCBozELMAkGA1UEBhMCSVQx
DzANBgNVBAgTBkl0YWxpYTENMAsGA1UEBxMEUGlzYTEcMBoGA1UEChMTVW5pdmVy
c2l0YScgZGkgUGlzYTEUMBIGA1UECxMLSW5mb3JtYXRpY2ExFzAVBgNVBAMTDlBD
LUxhYm9yYXRvcmlvMScwJQYJKoZIhvcNAQkBFhhzZ2FuZHVyckBjbGkuZGkudW5p
cGkuaXQwHhcNMDQwNjE1MTkxNTAwWhcNMDcwMzEyMTkxNTAwWjCBozELMAkGA1UE
BhMCSVQxDzANBgNVBAgTBkl0YWxpYTENMAsGA1UEBxMEUGlzYTEcMBoGA1UEChMT
VW5pdmVyc2l0YScgZGkgUGlzYTEUMBIGA1UECxMLSW5mb3JtYXRpY2ExFzAVBgNV
BAMTDlBDLUxhYm9yYXRvcmlvMScwJQYJKoZIhvcNAQkBFhhzZ2FuZHVyckBjbGku
ZGkudW5pcGkuaXQwXDANBgkqhkiG9w0BAQEFAANLADBIAkEA5jQs6YrZJE/vB5ke
NfoHhRU66tn+nbM/Uv89c0DD5rBvrweFGukCz0FCbdt8ct+kWDI29HJyI7HWkaFt
bCrScQIDAQABo4IBBDCCAQAwHQYDVR0OBBYEFCXodtxjgvLAMbWJ+JVujkBJvfg4
MIHQBgNVHSMEgcgwgcWAFCXodtxjgvLAMbWJ+JVujkBJvfg4oYGppIGmMIGjMQsw
CQYDVQQGEwJJVDEPMA0GA1UECBMGSXRhbGlhMQ0wCwYDVQQHEwRQaXNhMRwwGgYD
VQQKExNVbml2ZXJzaXRhJyBkaSBQaXNhMRQwEgYDVQQLEwtJbmZvcmlhdGljYTEX
MBUGA1UEAxMOUEMtTGFib3JhdG9yaW8xJzAlBgkqhkiG9w0BCQEWGHNnYW5kdXJy
QGNsaS5kaS51bmlwaS5pdIIBADAMBgNVHRMEBTADAQH/MA0GCSqGSIb3DQEBBAUA
A0EAGgU9ADAtQ+CO4v72Ug/YstqYAGfjiVtOPKTt/W0v3MRHVNeyxZRy+H43NxDo
eFmnxcXetWpPEHzJ19Xw1tR
```

3.2.4 Il file rsa.c

Le funzioni crittografiche legate al protocollo RSA, sono contenute nel file:

`rsa.c`

Le funzioni per generare chiavi RSA

Per generare una chiave RSA, utilizzando i valori di default, definiti come:

```
#define PUBLIC_EXPONENT 3
#define MODULUS_SIZE 1024
```

che sono i valori dell'esponente pubblico e la dimensione del modulo in bit, si usa la funzione

```
RSA *RSA_gen_key();
```

che ritorna una struttura "RSA", contenente tutte le informazioni relative alla coppia di chiavi (queste chiavi sono generate chiamando funzioni della liberia OpenSSL). Per generare una chiave commutativa, a partire da una chiave RSA, si utilizza la funzione:

```
RSA *RSA_gen_key_commutative(RSA *key);
```

che a sua volta richiama la funzione:

```
RSA *RSA_generate_key_commutative(int bits, BIGNUM *p_value,
                                            BIGNUM *q_value);
```

tramite la quale, con i valori di "p_value" e "q_value", presenti nella struttura "RSA" chiamata "key", si generano il modulo "N" ed i valori di "E" e di "D", rispettivamente gli esponenti pubblici e privati (che nel caso della crittografia commutativa devono essere entrambi tenuti segreti), e si ritorna una struttura "RSA" contenente tutte le informazioni necessarie. Questa funzione segue lo schema classico della generazione delle chiavi RSA, che riassumiamo qui brevemente (per maggiori dettagli vedi [4]):

- Si generano due numeri primi molto grandi, P e Q (nel caso della crittografia commutativa devono essere gli stessi per tutte la parti): il prodotto di questi due numeri dà come risultato il modulo N (e proprio dalla difficoltà computazionale di fattorizzare N rende RSA, per il momento, sicuro).

- Si calcola la funzione $\phi(N)$, che identifica il numero di interi che sono minori di N e relativamente primi con esso: nel caso di N dato dal prodotto di due numeri primi, la funzione è uguale a $(P-1)(Q-1)$.
- Si genera un intero E minore di $(P-1)(Q-1)$ e primo con esso.
- Si calcola D tramite l'equazione $ED = 1\ mod(\phi(N))$ (cioè, l'inverso di E).

Per salvare le chiavi pubbliche e private su file, in formato "PEM", si usa la funzione:

```
int write_rsa_keys(char *pubname, char *privname, RSA *rsa,
                   char *password);
```

che prende come parametri i due nomi dei file "pubname" e "privname" rispettivamente per la chiave pubblica e quella privata, la struttura "rsa" contenente le informazioni relative alle chiavi, e utilizza la parola segreta "password" per salvare il file contenente la chiave privata in modo da renderlo leggibile solo a chi è a conoscenza di quella parola. La funzione:

```
int calculate_size(char *pub_key_file, int length);
```

ritorna un intero indicante la dimensione dei blocchi di cifratura utilizzati da RSA, a partire dalla chiave salvata su "pub_key_file" e conoscendo la dimensione "length" del messaggio da cifrare. Questa funzione è utile poiché in caso di utilizzo dell'opzione *RSA_NO_PADDING*, durante la fase di crittature tramite RSA, OpenSSL richiede che la dimensione del messaggio sia multiplo esatto della dimensione del blocco (usando la chiave corrrente): la scelta che abbiamo preso, è stata allora quella di riempire con una serie di "0" i byte rimanenti al raggiungimento del multiplo esatto (era infatti indispensabile riempire la parte rimanente con la stessa sequenza di byte per riuscire ad identificare messaggi identici dopo la crittatura[1]).

[1] la scelta di non mettere byte casuali di padding alla fine del messaggio è una pratica poco sicura perché espone il crittogramma ad una serie di analisi e di attacchi per forzarne la decifrazione: questo però è l'unico metodo che consente di verificare se due messaggi crittati siano il risultato della crittazione delle stesso messaggio.

Le funzioni per crittare

Le funzioni per crittare sono due, di cui la prima:

```
unsigned char *rsa_encrypt1(char *pub_key_file,
                        char *from, int length);
```

viene utilizzate per crittare "from" di dimensione "length" utilizzando la chiave salvata su "pub_key_file" e ritorna un array di unsigned char. Questa funzione viene utilizzata la prima volta dal protocollo della Secure Union e della Secure Intersection da tutte le parti per crittare le stringhe in chiaro, riempiendo il padding di carattere "0". La seconda funzione:

```
unsigned char *rsa_encrypt2(char *pub_key_file,
               unsigned char *from, int length);
```

viene invece adoperata per crittare messaggi già crittati, con i parametri identici alla precedente funzione; in questo caso non c'è però bisogno di preoccuparsi del padding con la differenza che nella seconda funzione i due messaggi, prima e dopo, avranno la stessa dimensione, mentre nel primo caso (a meno di un multiplo esatto), il messaggio in chiaro e quello crittato avranno lunghezza differente.

La funzione rsa_decrypt

L'ultima funzione di questo file è la:

```
int rsa_decrypt(char *priv_key_file, unsigned char *from,
             unsigned char *to, int size, char *password);
```

che viene utilizzata per decrittare il messaggio presente in "from", tramite la chiave privata salvata sul file "priv_key_file" con la parola segreta "password", scrivendo il risultato nell'array "to". A differenza delle funzioni di crittatura, dunque, in questo caso l'array di destinazione sarà stato allocato precedentemente alla chiamata di questa funzione (questo array avrà cioè la stessa dimensione "size" dell'array "from", anche nel caso in cui la decifratura sia quella finale, che restituirà in tale array il messaggio, seguito dal carattere '\0' e da una serie di caratteri di padding).

3.2.5 Il file secure.c

Definizioni

Alcune "#define" hanno bisogno di spiegazioni, in quanto sono fondamentali per la comprensione del funzionamento della libreria:

```
#define SUM_PORT_NUMBER 8815
#define UNION_PORT_NUMBER 8816
#define INTER_PORT_NUMBER 8817
#define SCALAR_PORT_NUMBER 8818
```

Queste definiscono le porte di ascolto (il protocollo utilizzato è ovviamente il TCP, in quanto l'unico supportato da SSL) utilizzate dalle varie funzioni, rispettivamente per la Secure Sum, la Secure Union, la Secure Size of Set Intersection e il Secure Scalar Product: ad ogni protocollo è stata assegnata cioè una porta, in modo che sia anche possibile, laddove necessario, utilizzare queste funzioni in parallelo sulla stessa macchina.

Queste altre:

```
#define SUM_CONNECT_TIMEOUT 20000
#define UNION_CONNECT_TIMEOUT 2000
#define SCALAR_CONNECT_TIMEOUT 3000
```

si riferiscono al timeout (espresso in secondi) in cui le funzioni, rispettivamente di Secure Sum, Secure Union/Secure Intersection e Secure Scalar Product, rimangono in attesa di ricevere una connessione da un altro host prima di chiudere la connessione, o provano a connettersi ad un host prima di dichiararlo irraggiungible e chiudere la connessione (i timeout sono settati alti in quanto spesso i tempi di crittatura/decrittazione fanno sì che gli host rimangano in ascolto per molto tempo prima di ricevere dati). Altre "#define" saranno spiegate nelle sezioni relative alle funzioni che le utilizzano. Adesso possiamo analizzare le funzioni presenti nel file, delle quali alcune sono di utilità generale, ed altre sono specifiche per i protocolli di privacy-preserving.

La funzione ip_sort

La prima funzione generale, utile per tutte le quattro funzioni che formano il nucleo della libreria è la:

```
char **ip_sort(char *ip_file, int *hosts);
```

funzione che a partire dal nome di un file "ip_file" contenente una serie di righe, ognuna rappresentante un indirizzo IP (nella forma canonica di numeri decimali separati da punti , cioè A.B.C.D), crea e ritorna un array di indirizzi IP ordinato in ordine crescente (vedi pag.15), scrivendo la sua dimensione nella variabile "hosts".

La funzione create_sockaddresses

C'era anche bisogno di una funzione che creasse delle strutture *sockaddr_in* per permettere di ricevere/effettuare connessioni da/verso altre macchine:

```
psockaddr_in *create_sockaddresses(char **ip_addresses,
   int hosts_num, char *my_name, char *my_ip, int *pos);
```

Questa funzione crea un array di strutture *sockaddr_in*, ritornate come *psockaddr_in* tramite la:

```
typedef struct sockaddr_in *psockaddr_in;
```

a partire dalla lista ordinata di IP "ip_addresses" contenente "hosts_num" indirizzi IP, per cui in precedenza sarà stata chiamata la funzione "ip_sort" (descritta in precedenza) per ottenere questi due parametri; alla fine la funzione scrive in "my_ip" l'indirizzo IP di "my_name" (ovvero, il nome della macchina su cui viene invocata la funzione) ed in "pos" la posizione dell'host all'interno dalla lista degli indirizzi IP (0 se è il primo della lista, 1 se è il secondo e così via). Solitamente questa funzione viene preceduta da una chiamata a:

```
gethostname(my_name, NAME_LENGTH);
```

funzione della libreria C che ritorna il nome dell'host, dove "NAME_LENGTH" è definito come

```
#define NAME_LENGTH 50
```

La funzione connect_timeout

L'ultima funzione di cui ci occupiamo, utilizzata da tutte le altre quattro funzioni privacy-preserving, è la:

```
int connect_timeout(int sockfd, struct sockaddr *sock_address,
                    int nsec);
```

funzione utilizzata per eseguire la funzione di libreria *connect*, passandole come primi due parametri "socfkd" e "sock_address". Questa funzione prova a connettersi verso l'host per "nsec" prima di dichiararlo non raggiungible: per fare questo la "connect_timeout" chiama "alarm" che, se il tempo scade senza che sia stato possibile connettersi, ritorna al chiamante della funzione.

La funzione secure_sum

La prima funzione, tra le quattro descritte in [2], che analizziamo è la *secure sum*. Essa ha come prototipo:

```
unsigned long secure_sum(char **ip_addresses,
          int hosts_num, unsigned long value);
```

che deve essere chiamata utilizzando come parametri "ip_addresses", un array di indirizzi IP ordinati. Prima di chiamare questa funzione è quindi stata eseguita la chiamata a:

```
ip_sort(certificate/ip.txt, &hosts_num);
```

che ritorna un array di indirizzi ordinati a partire da quelli presenti nel file "ip.txt" (vedi pag.29); le altrue due variabili sono "hosts_num" che identifica il numero di client partecipanti al protocollo e "value" che è il valore che il client possiede e che vuole sommare a quello degli altri; la funzione ritorna la somma totale in maniera sicura.

Il comportamento della funzione è il seguente: viene innanzitutto inizializzato OpenSSL, tramite la chiamata a

```
ssl_init();
```

poi crea le strutture “sockaddr_in”, utilizzate per comunicare con gli altri client, tramite la chiamata a:

```
create_sockaddresses();
```

ed inoltre crea i contesti *SSL_CTX*, sia client che server, che verranno adoperati dalla chiamate della libreria di OpensSSL ed il socket di ascolto, mettendosi in “listen” su di esso. Eseguita questa prima fase di inizializzazione comincia il protocollo vero e proprio: il client “0” (quello che cioè ha l’indirizzo IP più basso tra i partecipanti dell’algoritmo) è quello che comincia per primo ad inviare i dati: gli altri lo seguono subito dopo, ovvero il client “1” riceverà il valore dal client “0”, eseguirà i calcoli specificati dall’algoritmo (vedi pag.7) ed invierà il risultato al client “2”, e così via, fino a che l’ultimo host invierà il risultato al client “0” che provvederà ad eseguire la parte finale del protocollo ed inviare il risultato a tutti gli altri client. Quindi, i client hanno un comportamento diverso a seconda del valore della variabile *pos* che indica la posizione all’interno dell’array di indirizzi, tramite la condizione:

```
if(pos)
{
...
}
else
{
...
}
```

che fa sì che tutti i client che hanno posizione maggiore di “0” eseguiranno questi tre passi:

1. Rimangono in ascolto in attesa del valore dal client precedente (che ha la posizione *(pos-1)*).

2. Dopo aver eseguito i calcoli dell’algoritmo, inviano il risultato parziale al client successivo (che ha posizione tra gli indirizzi IP *next_pos*).

3. Rimangono in ascolto del risultato finale inviatogli dal client “0”.

ed il client “0” eseguirà invece questi tre passi:

1. Invia il risultato della somma tra il suo valore ed il numero random al client “1”.
2. Rimane in ascolto in attesa che l’ultimo client gli invii il risultato finale.
3. Esegue la parte finale del protocollo, ed invia il risultato a tutti.

Le funzioni richiamate per eseguire questi passi sono la:

```
int secure_sum_send(SSL_CTX* ctx_client,
    struct sockaddr *sock_address, unsigned long value);
```

che permette di inviare il valore “value” al client i cui dati sono presenti nella struttura “sock_address” passata per riferimento, utilizzando il contesto SSL “ctx_client” (creato all’interno della *secure_sum*), e la funzione:

```
int secure_sum_receive(int rcv_sockfd, SSL* ssl,
    struct sockaddr *sock_address, unsigned long *value);
```

che fa sì che sia possibile ricevere un unsigned long dal precedente client, salvandolo in “value”, rimanendo in attesa di una connessione dal socket “rcv_sockfd” utilizzando la struttura “sock_address” (che serve per la chiamata alla funzione di libreria *accept*) e la struttura “ssl”, creata in precedenza tramite la chiamata a:

```
if((ssl_server = SSL_new(ctx_server)) == NULL)
{
     print_ssl_error("SSL_new");
     exit(1);
}
```

dove “ssl_server” è il secondo parametro passato alla *secure_sum_receive*, e “ctx_server” è il contesto *SSL_CTX* creato in fase di inizializzazione. Inoltre vengono utilizzate le funzioni:

```
generate_random_long();
```

utilizzata dal client “0” per generare il numero random utilizzato dal protocollo della Secure Sum per nascondere il valore reale della somma, e:

```
unsigned long calculate_value(unsigned long tot_value,
                                    unsigned long value);
```

che viene utilizzata da tutti i client per calcolare il risultato della somma modulare tra il proprio valore "value" e quello totale "tot_value" dal client precedente, dove il modulo viene definito tramite la':

```
#define MOD ULONG_MAX
```

dove *ULONG_MAX* (definito nella libreria del C) indica il numero più grande che può essere creato tramite un unsigned long.

La funzione secure_union_intersection

Per quanto riguarda le funzioni *Secure Union* e *Secure Size of Set Intersection*, essendo molto simili i due algoritmi che le compongono, abbiamo realizzato un'unica funzione che esegue tutti e due i protocolli. Questa funzione è la:

```
unsigned char **secure_union_intersection(char **ip_addresses,
int hosts_num, char **sets, unsigned long *total, int protocol);
```

i cui parametri specificano, nell'ordine, l'array di indirizzi IP ed il numero di host che eseguono il protocollo (analogamente alla Secure Sum), l'insieme degli elementi, passati come array di stringhe, dei quali si vuole fare l'unione o l'intersezione, il puntatore ad una variabile in cui verrà scritto il risulato finale (la cardinalità dell'unione/intersezione) ed il protocollo, ovvero Secure Union o Secure Intersection, per le quali esistono due "#define":

```
#define SECURE_UNION 0
#define SECURE_INTERSECTION 1
```

Per richiamare questa funzione esistono altre due funzioni, una per la Secure Union ed una per la Secure Size of Set Intersection, i cui prototipi sono:

```
unsigned char **secure_union(char **ip_addresses,
int hosts_num, char **sets, unsigned long *total);
```

e:

```
unsigned long secure_intersection(char **ip_addresses,
                                  int hosts_num, char **sets);
```

la prima delle quali ritorna l'array di stringhe risultato dell'unione (i parametri della funzione vengono passati alla *secure_union_intersection*), mentre la seconda ritorna il risultato delle cardinalità dell'intersezione degli insiemi di stringhe (anche in questo caso i parametri vengono passati alla funzione generale). Prima di scendere nei dettagli dell'implementazione delle due primitive, analizziamo brevemente il funzionamento generale della *secure_union_intersection*:

- Il client "0" invia i valori "P" e "Q" (vedi pag.25) a tutti gli altri client, e si genera la sua coppia di chiavi commutative.
- Tutti i client, dall'uno all'ultimo, alla ricezione dei due valori, si generano la loro coppia di chiavi commmutative.
- Il client "0" critta le proprie stringhe con la sua chiave pubblica commutativa, ed invia quindi le stringhe crittate al client "1".
- Tutti i client rimangono in ascolto di ricevere le stringhe crittate del client precedente: alla ricezione di queste, le critteranno a loro volta, tramite la loro chiave pubblica, permuteranno l'ordine con la quale sono state ricevute, e le invieranno all'host successivo (se il numero totale di crittature effettutate su quelle stringhe è ancora minore del numero totale degli host), oppure al client "0" (se sono state crittate da tutti). La prima volta (ad eccezione del client "0") dopo aver ricevuto la chiave RSA commutativa, invieranno anche le loro stringhe crittate all'host successivo.
- Il client "0" rimane in attesa che ogni client gli invii le stringhe crittate da tutti, ovvero le stringhe di un client che dopo essere state crittate anche da tutti gli altri client, devono essere inviate a lui per fare l'unione o l'intersezione.
- Se il protocollo è la Secure Intersection, il client "0" calcola la cardinalità dell'intersezione delle stringhe, invia il risultato a tutti, ed il protocollo termina.

- Se il protocollo è la Secure Union, il client "0" esegue l'unione delle stringhe (rimuovendone i duplicati), poi comincia a decrittare tutte le stringhe con la propria chiave privata, e le invia al client "1".
- Il client "1" e tutti i successivi fino al penultimo, ricevono le stringhe, le decrittano con la loro chiave privata e le inviano al successivo.
- L'ultimo client, riceve le stringhe, effettua l'ultima decrittazione, ed invia l'unione delle stringe in chiaro a tutti gli altri client, ed il protocollo termina.

La funzione *secure_union_intersection* utilizza al suo interno *thread*, che vengono impiegati per crittare, decrittare, ricevere le stringhe, inviarle, ricevere i valori della chiave RSA commutativa. Quello che viene fatto, in generale, è di rimanere in ascolto sulla porta della Secure Union o della Secure Intersection (questo dipende dal protocollo scelto all'atto dell'invocazione della funzione), ricevere una richiesta e, a seconda del tipo di richiesta, lanciare un thread apposito che gestisca tale richiesta. Prima di vedere però come vengono gestite queste richieste, analizziamo la fase di inizializzazione della funzione (in parte simile alla Secure Sum): a seconda del protocollo, la porta di ascolto sarà diversa, ed infatti viene eseguito questo controllo:

```
if(protocol == SECURE_UNION)
{
     RCV_PORT_NUMBER = UNION_PORT_NUMBER;
}
else
{
     RCV_PORT_NUMBER = INTER_PORT_NUMBER;
}
```

in modo da mettersi in "listen" e connettersi alla corretta porta. Poi si inizializza OpenSSL, si creano le strutture *sockaddr_in*, si conta il numero di stringhe passate (si ricorda che l'array passato deve essere terminato da NULL), si calcola la lunghezza della stringa presente nella prima posizione, che servirà per calcolare la dimensione dei blocchi di cifratura: per fare questo, le stringhe dovranno avere tutte la lunghezza tale che la divisione di essa per la dimensione della chiave dia lo stesso risultato (per

esempio, se la chiave è di 128 byte, le stringhe dovranno essere tutte comprese tra 0 e 127 byte, o tra 128 e 255 byte, e così via), sia per il problema del padding descritto in precedenza che per ottimizzare il protocollo in fase di invio e come strutture dati (la lunghezza delle stringhe dopo la crittatura è sempre la medesima). Poi, come per la Secure Sum, si creano i contesti *SSL_CTX* sia lato client che lato server e si preparano le stringhe utilizzate per salvare le chiavi commutative, che avranno i nomi:

```
A.B.C.D.comm_public_key.pem
A.B.C.D.comm_private_key.pem
```

indicanti, rispettivamente, il nome del file su cui salvare la chiave commutativa pubblica e quella privata, dove A.B.C.D è l'indirizzo IP del client. La directory in cui salvare i due file è la:

```
certificate
```

Infine si crea il socket d'ascolto e si chiama la funzione *listen* su di esso. Adesso comincia il protocollo vero e proprio: il client "0" genera una chiave RSA tramite la

```
RSA_gen_key();
```

i cui valori P e Q verranno inviati a tutti gli altri client per la generazione delle chiavi commutative. Per fare questo, il client "0" genera (*hosts_num* - 1) thread che invocano la funzione:

```
void *thread_secure_union_intersection_send_rsa(void *ptr);
```

passandogli, come parametro, un puntatore alla struttura definita come:

```
typedef struct params_rsa_s
{
     SSL_CTX* ctx_client;
     struct sockaddr *sock_address;
     BIGNUM *p;
     BIGNUM *q;
}param_rsa_s;
```

contenente, nell'ordine, il contesto utilizzato da OpenSSL per creare il canale SSL, il puntatore alla struttura *sockaddr* contenente i parametri del client a cui vanno inviati i valori di RSA (indirizzo IP, porta di ascolto), e i due valori "P" e "Q" utilizzati per generare la coppia di chiavi commutative. Questa funzione invierà per prima la richiesta di invio della chiave RSA, poi convertirà "P" e "Q" da *BIGNUM* alla notazione decimale, e li invierà sotto forma di stringa al client. Come nel caso di questa funzione, per tutte le funzioni che devono inviare dati il primo invio che effettuano viene utilizzato per identificare il tipo di richiesta che viene fatta: ad ogni richiesta viene assegnata un intero, e all'atto della ricezione di questo intero, i client sono in grado di capire che tipo di invio dati viene fatto e sanno di conseguenza quale funzione invocare per soddisfare tale richiesta. Queste vengono definite tramite delle "#define":

```
#define ENCRYPT 1
#define DECRYPT 2
#define SEND_CRYPT 3
#define SEND_CLEAR 4
#define SEND_RSA 5
#define SEND_INTERSECTION 6
```

che hanno il seguente significato:

- ENCRYPT: richiesta di inviare le stringhe sulle quali va effettuata la crittatura.
- DECRYPT: richiesta di inviare le stringhe sulle quali va effettuata la decrittazione.
- SEND_CRYPT: richiesta di inviare le stringhe crittate da tutti (quelle che vanno inviate all'host "0" per effetture l'unione o l'intersezione).
- SEND_CLEAR: richiesta di inviare le stringhe in chiaro risultato finale della Secure Union.
- SEND_RSA: richiesta di inviare i parametri per creare una chiave RSA commutativa.
- SEND_INTERSECTION: richiesta di inviare il valore finale risultato della Secure Intersection.

A questo punto, dopo aver inviato i parametri RSA per le chiavi commutative, il client "0" provvede a lanciare un altro thread che si occuperà di crittare le stringhe in chiaro (passate come parametri della *secure_union_intersection*) tramite la funzione:

```
void *thread_secure_union_intersection_c(void *ptr);
```

che ha come parametro un puntatore alla struttura definita come:

```
typedef struct params_union_intersection_c
{
     int other_sockfd;
     SSL *ssl_server;
     int len;
     SSL_CTX *ctx_client;
     struct sockaddr *sock_address;
     struct sockaddr *server_sock_address;
     char *comm_public_key;
     char **clear;
     unsigned long n_clear;
     int max;
     struct params_union_intersection_receive_c *par_rec_c;
}param_union_intersection_c;
```

le cui variabili indicano:

- *other_sockfd*: indica il socket da cui ricevere le stringhe; se viene settato a "0" non ci sarà ricezione ma solo invio (nel caso del primo invio in cui si crittano le proprie stringhe e si inviano).
- *ssl_server*: la struttura utilizzata da OpenSSL per creare il canale SSL di ascolto (uguale a NULL se ci sono solo stringhe da inviare).
- *len*: lunghezza delle stringhe in ricezione ed invio (ovvero, un multiplo della dimensione del blocco di cifratura).
- *ctx_client*: il contesto utilizzato da OpenSSL per creare una struttura "SSL" che servirà per creare un canale per inviare le stringhe al client successivo.

- *sock_address*: la struttura sockaddr con i parametri TCP/IP del client successivo a cui inviare le stringhe crittate.
- *server_sock_address*: la struttura sockaddr per inviare i dati al client "0" (nel caso in cui le stringhe siano state crittate da tutti i client).
- *comm_public_key*: il nome del file contentente la chiave pubblica commutativa, utilizzate per crittare le stringhe.
- *clear*: usata per contenere le stringhe in chiaro che, crittate, vengono inviate al client successivo (questa variabile viene utilizzate solo in questo caso).
- *n_clear*: contiene il numero dei messaggi in chiaro che devono essere crittari.
- *max*: contiene il numero degli host che devono crittare il messaggio.
- *par_rec_c*: puntatore ad una struttura che serve al client "0" per contenere le stringhe crittate da tutti (vedi dopo).

A questo punto, il client "0" ha due possibilità dettate dal protocollo: se questo è la Secure Union, lancerà un altro thread che invocherà la funzione:

```
void *thread_secure_union_send_critt1(void *ptr);
```

che utilizza i parametri presenti in "ptr" che è un puntatore ad una struttura:

```
typedef struct params_critt_s
{
     int len;
     SSL_CTX *ctx_client;
     struct sockaddr *sock_address;
     char *comm_private_key;
     int max;
     char *password;
     struct params_union_intersection_receive_c *par_rec_c;
}param_critt_s;
```

i cui parametri indicano:

- *len*: lunghezza delle stringhe in ricezione ed invio.
- *ctx_client*: contesto utilizzato per creare una struttura *SSL* da usare per inviare i dati.
- *sock_address*: la struttura con i parametri del client a cui inviare le stringhe.
- *comm_private_key*: il nome del file con la chiave privata utilizzate per decrittare le stringhe.
- *max*: numero indicante il numero di host che devono decrittare il messaggio.
- *password*: stringa che indica con quale parola segreta è stata salvata la chiave privata (occorre infatti per leggere il file e ricavare la chiave privata).
- *par_rec_c*: struttura contenente le stringhe crittate da tutti.

e questo thread rimarrà in attesa di aver ricevuto tutte le stringe crittate da tutti, ed invierà, successivamente, le stringhe risultato dell'unione al client successivo, dopo averle decrittate. Se, invece, il protocollo è la Secure Intersection, il client "0" lancerà un thread che chiamerà la funzione:

```
void *thread_secure_intersection_send_result(void *ptr);
```

che ha come parametri un puntatore alla struttura definita come:

```
typedef struct params_intersection_s
{
     int len;
     SSL_CTX *ctx_client;
     struct sockaddr **sock_addresses;
     int max;
     unsigned long *n_inter;
     struct params_union_intersection_receive_c *par_rec_c;
}param_intersection_s;
```

le cui variabili sono identiche alla struttura precedente, eccetto l'aggiunta della variabile "n_inter" che conterrà il risultato della Secure Intersection ed anche in questo caso il thread rimarrà in attesa che tutte le stringhe crittate da tutti i client gli siano arrivate, per farne l'intersezione ed inviare il risultato finale a tutti. Per fare sì che questi thread (sia nel caso della Secure Union che in quello della Secure Intersection) sappiano quando possono inviare le stringhe, il client "0" deve aver bisogno di una struttura:

```
typedef struct params_union_intersection_receive_c
{
     SSL *ssl_server;
     int tot;
     int max;
     unsigned long n_critt;
     unsigned char ***critt;
     int len;
}param_union_intersection_receive_c;
```

che è una variabile col nome *par_rec_c* i cui parametri indicano:

- *ssl_server*: la struttura usata per accettare connessioni SSL.
- *tot*: indica il numero dei client che hanno inviate le stringhe.
- *max*: indica il numero dei client che devono inviare le stringhe.
- *n_critt*: numero totale di stringhe crittate ricevute.
- *critt*: puntatore agli array di stringhe ricevute dai client.
- *len*: lunghezza delle stringhe crittate.

che viene utilizzata sia da questi thread che da altri (vedi sotto) per contenere tutte le stringhe crittate da tutti i client. I thread, ovviamente, hanno bisogno di sincronizzarsi su questa struttura, nel caso in cui:

1. All'interno della *thread_secure_union_intersection_c*, il client "0" abbia crittato delle stringhe e verifichi che queste sono state crittate da tutti gli altri client, e quindi non c'è più necessità di inviarle agli altri host, per cui se le salva nella struttura.

2. All'interno della *thread_secure_union_intersection_receive_critt*, il client "0" riceverà da tutti gli altri host le stringhe che sono state crittate da tutti, e li salverà in quella struttura.

3. All'interno della *thread_secure_union_send_critt1* dove il client "0" rimarrà in attesa di aver ricevuto tutte le stringhe crittate da tutti per poi effetturare l'unione.

4. All'interno della *thread_secure_intersection_send_result* dove, come nel caso precedente, il client "0" deve aspettare che le stringhe crittate da tutti gli siano state inviate da tutti i client, prima di effettuare l'intersezione tra di esse.

e per sincronizzarsi utilizzano due variabili globali:

```
pthread_mutex_t mutex_critt_params = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_critt_params = PTHREAD_COND_INITIALIZER;
```

utilizzate la prima come *mutex*, cioè per avere accesso esclusivo alla variabile, la seconda come *condition*, ovvero il thread si blocca e rimane in attesa di essere "svegliato" al verificarsi di una certa condizione. In questa maniera, i vari thread possono ricevere le stringhe, salvarle, fare l'unione ed inviarle, senza fare accessi contemporanei alla struttura e senza perdere cicli di clock in attesa attiva (nel caso in cui si aspetta che tutti i client abbiano inviato le stringhe crittate da tutti, per eseguire l'unione o l'intersezione).

Gli altri client, invece, a questo punto sono entrati nel ciclo principale, che prevede che essi rimangano in ascolto di richieste (invio parametri RSA, invio stringhe da crittare, invio stringhe da decrittare, etc) fino a che il risultato della Secure Union o Intersection sia stato inviato loro. Quello che fanno è quindi di rimanere in "listen" sulla porta TCP del protocollo, ricevere una richiesta (vedi pag.37), e lanciare un thread specifico che si occupi di tale richiesta. Dentro il "loop" principale, è presente

lo statement *switch* per gestire tali richieste, che, a seconda dell'intero ricevuto, farà sì che i client lancino i thread che eseguiranno queste funzioni:

- Caso "ENCRYPT": viene lanciato un thread che esegue la funzione:

  ```
  thread_secure_union_intersection_c
  ```

  per il quale vedi pag.38.

- Caso "DECRYPT" (solo per la Secure Union): il thread che viene lanciato invoca la funzione:

  ```
  thread_secure_union_d
  ```

  che ha come parametro un puntatore alla struttura definita come:

  ```
  typedef struct params_union_d
  {
       SSL *ssl_server;
       int len;
       SSL_CTX *ctx_client;
       struct sockaddr *sock_address;
       struct sockaddr **server_sock_address;
       char *comm_private_key;
       int max;
       char *password;
       unsigned char ***clear;
       unsigned long *n_clear;
  }param_union_d
  ```

  le cui variabili indicano

  - *ssl_server*: la struttura *SSL* usata per creare il canale di ascolto SSL.
  - *len*: la lunghezza delle stringhe di ricezione ed invio.
  - *ctx_client*: il contesto utilizzato per creare un canale SSL lato client.

- *sock_address*: la struttura contenente i parametri del client successivo.
- *server_sock_address*: le strutture con i parametri di tutti i client, utilizzate dall'ultimo client per inviare le stringhe in chiaro risultato finale dell'unione.
- *comm_private_key*: il nome del file contenente la chiave privata commutativa utilizzata per decifrare.
- *max*: numero totale di decrittazioni che devono essere effettuate sulle stringhe.
- *password*: la password usata per salvare il file contenente la chiave privata del client.
- *clear*: usata dall'ultimo client per salvarsi le stringhe in chiaro finali.
- *n_clear*: usata dall'ultimo client per salvarsi il numero totale delle stringhe finali.

Questo funzione semplicemente rimane in ascolto di ricevere le stringhe dal client precedente, le decritta utilizzando la chiave privata commutativa, e le invia al client successivo. L'ultimo client, a differenza degli altri, effettuando l'ultima decrittazione otterrà le stringhe in chiaro, e quindi si farà una copia (nella struttura esiste un puntatore grazie al quale le stringhe possono essere ritornate), ed invierà a tutti gli altri client le stringhe in chiaro.

- Caso "SEND_RSA": viene lanciato un thread che esegue la funzione:

  ```
  thread_secure_union_intersection_receive_rsa
  ```

  che utilizza la struttura (passata con un puntatore):

  ```
  typedef struct params_rsa_r
  {
       SSL *ssl_server;
       BIGNUM **p;
       BIGNUM **q;
  }param_rsa_r
  ```

le cui variabili sono il server SSL da cui ricevere i dati, e due puntatori a “p” e “q” utilizzati per generare la chiave commutativa. Inoltre i client, la prima volta che ricevono una richiesta di questo genere, lanciano un altro thread utilizzato per crittare le proprie stringhe in chiaro ma questa volta questo thread, grazie al fatto che la variabile “other_sockfd” della struttura passata con un puntatore viene settata a zero, si occuperà solamente di crittare ed inviare le stringhe in chiaro passate come parametri della *secure_union_intersection* senza cioè mettersi in ascolto (questo è infatti necessario perché la prima volta i client devono solamente crittare le loro stringhe senza riceverle).

- Caso “SEND_CRYPT”: solo il client “0” può ricevere questo tipo di richieste, indicanti l’invio delle stringhe crittate da tutti. Il thread che viene lanciato esegue la funzione:

  `thread_secure_union_intersection_receive_critt`

  a cui viene passata tramite puntatore la variabile “par_rec_c” (vedi pag.41) in cui salvare le stringhe ricevute (dopodiché il thread lanciato all’avvio dal client “0”, deputato a fare l’unione o intersezione delle stringhe, quando tutti i client avranno inviate le stringhe verrà svegliato tramite una chiamata alla funzione della libreria pthread *pthread_cond_broadcast*).

- Caso “SEND_CLEAR”: tutti i client, eccetto il client “0” ricevono questa richiesta, e lanciano un thread che esegue la funzione:

  `thread_secure_union_receive_clear`

  a cui viene passata, tramite puntatore, una struttura:

```
typedef struct params_clear_r
{
     SSL *ssl_server;
     int len;
     unsigned char ***clear;
```

```
        unsigned long *n_clear;
    }param_clear_r;
```

le cui variabili sono, nell'ordine, il server SSL da cui ricevere le stringhe, la lunghezza delle stringhe, il puntatore alla lista in cui salvare le stringhe e il puntatore alla variabile che conterrà il numero totale di stringhe ricevute.

- Caso "SEND_INTERSECTION": solo per la Secure Intersection; in questo caso il thread esegue la funzione:

```
thread_secure_intersection_receive_result
```

che ha come parametro un puntatore ad una struttura definita come:

```
typedef struct params_intersection_r
{
     SSL *ssl_server;
     unsigned long *n_inter;
}param_intersection_r;
```

le cui variabili sono il puntatore al server SSL utilizzato per ricevere il risultato finale, ed il puntatore alla variabile in cui salvare tale valore.

Per uscire da questo ciclo (settando la variabile "end" ad "1") i casi sono:

1. Nel ramo dello switch "ENCRYPT": se il client è il numero 0, ed il protocollo è la secure intersection, e tutti i client hanno inviato le stringhe crittate da tutti, il client "0" esce poiché all'avvio ha già lanciato un thread che si occuperà di fare l'unione od intersezione delle stringhe, e non ha inoltre bisogno di ricevere altri dati.

2. Nel ramo dello switch "DECRYPT": l'ultimo client esce, in quanto a questo punto ha già lanciato il thread che si occuperà di inviare le stringhe in chiaro (risultato finale del protocollo) a tutti gli altri client e provvederà inoltre a salvare nelle variabili del client stesso le stringhe ed il numero di esse.

3. Nel ramo dello switch “SEND_CRYPT”: analogo al primo caso (dipende dal fatto che i thread chiamati nei due rami possono entrambi ricevere l’ultima serie di stringhe crittate da tutti, o dal client precedente il client “0” nel primo caso, o da uno qualsiasi degli altri client nel secondo caso: essendo sincronizzate le variabili a cui accedono i thread e non deterministico l’accesso ad esse, entrambi i casi vanno contemplati).

4. Nel ramo dello switch “SEND_CLEAR”: tutti i client, ad eccezione dell’ultimo, escono in quanto hanno ricevuto il risultato finale della Secure Union.

5. Nel ramo dello switch “SEND_INTERSECTION”: tutti i client, ad eccezione dell’ultimo, escono in quanto hanno ricevuto il risultato finale della Secure Intersection.

Vengono inoltre utilizzate le funzioni:

```
void permute_buffer(unsigned char **buffs,
                        unsigned long max);

unsigned char **compare_buffer(unsigned char **buffs,
                         unsigned long len, int size);

unsigned long check_intersection(unsigned char **buffs,
            unsigned long len, int size, int hosts_num);
```

La prima viene utilizzata sia nel protocollo della Secure Union che in quello della Secure Size of Set Intersection per permutare l’ordine dei “max” elementi presenti in “buffs” (questa funzione viene utilizzata dopo aver crittato gli elementi e prima di inviarli all’host successivo); la seconda funzione serve per rimuovere gli elementi duplicati presenti in “buffs” di numero “len” e tutti di dimensione “size” all’interno della Secure Union, ritornando un vettore contenente l’unione di tale elementi senza duplicati; infine la terza funzione viene utilizzata all’interno del protocollo della Secure Intersection per contare il numero degli elementi presenti in “buffs” di numero “len” e tutti di dimensione “size” posseduti da tutti gli “host_num” host (cioè, la cardinalità dell’intersezione degli elementi).

La funzione secure_scalar_product

L'ultima primitiva di cui ci occupiamo è la Secure Scalar Product, implementata dalla funzione il cui prototipo è:

```
unsigned long secure_scalar_product(char **ip_addresses,
                                    int hosts_num, char *set);
```

in cui le variabili "ip_addresses" e "hosts_num" hanno lo stesso significato di quello già visto per le due funzioni "secure_sum" (a pag.30) e "secure_union_intersection" (a pag.33), dove in questo caso però "hosts_num" deve essere uguale a due (il protocollo può essere implementato infatti tra solo due client), mentre "set" è la rappresentazione della colonna (oppure un valore booleano indicante la presenza/assenza del sottoinsieme nella riga) del client che chiama la funzione, tramite una serie di "0" o di "1", indicanti la presenza o l'assenza dell'elemento della transazione nella colonna considerata (se invece sono più di uno gli elementi da considerare, si può eseguire l'AND tra gli elementi formanti lo stesso sottoinsieme, espressi anche in questo caso con i valori booleani "0" od "1", ottenendo quindi un vettore che può essere passato alla funzione). Sono presenti alcune funzioni accessorie, utilizzate per eseguire i calcoli tra i vettori e le matrici:

```
unsigned long *generate_random_vector(int n);

unsigned long **generate_random_matrix(int n, int m);

unsigned long *generate_empty_vector(int n);

unsigned long **generate_empty_matrix(int n, int m);

unsigned long **traspose_matrix(unsigned long **ma,
                                int n, int m);

unsigned long *mult_matrix_vector(unsigned long **ma,
                    unsigned long *v, int n, int m);
```

```
unsigned long mult_vector_vector(unsigned long *v1,
                          unsigned long *v2, int n);

unsigned long *sum_vector_vector(unsigned long *v1,
                          unsigned long *v2, int n);
```

che, nell'ordine, permettono:

- Di generare un vettore di dimensione "n" di elementi random unsigned long.
- Di generare una matrice di rango "n" per "m" di elementi random unsigned long.
- Di generare un vettore di dimensione "n" con gli tutti gli elementi inizializzati a zero.
- Di creare una matrice di rango "n" per "m" i cui elementi vengono inizializzati tutti a zero.
- Di creare una matrice trasposta delle matrice "ma" di rango "n" per "m".
- Di effetturare la moltiplicazione tra la matrice "ma" di rango "n" per "m" ed il vettore "v" di dimensione "m" e ritornare il vettore così ottenuto.
- Di effetturare il prodotto tra i due vettori "v1" e "v2" di dimensione "n" e di ritornare il risultato.
- Di fare la somma tra i due vettori "v1" e "v2" di dimensione "n" ritornando il valore così ottenuto.

A seconda che i due client "A" e "B" abbiano bisogno, seguendo i passi descritti dall'algoritmo del Secure Scalar Product (vedi a pag.11), di inviare o ricevere dati, ci saranno delle funzioni che permetteranno di inviare matrici, vettori e scalari, o viceversa di ricevere matrici, vettori e scalari. Al primo gruppo appartengono le funzioni:

```
int secure_scalar_send_matrix(SSL_CTX *ctx_client,
                    struct sockaddr *sock_address,
```

```
                unsigned long **matrix, int m, int n);


int secure_scalar_send_vector(SSL_CTX *ctx_client,
                    struct sockaddr *sock_address,
                     unsigned long *vector, int n);

int secure_scalar_send_scalar(SSL_CTX *ctx_client,
                    struct sockaddr *sock_address,
                            unsigned long scalar);
```

i cui parametri indicano:

- *ctx_client*: il contesto utilizzato per creare il canale SSL per inviare i dati.
- *sock_address*: la struttura contenente i parametri TCP/IP dell'altro client.
- *matrix*, *vector*, *scalar*: la matrice, il vettore o lo scalare i cui valori devono essere inviati.
- *m*, *n*: indicano la dimensione delle matrici e dei vettori da inviare.

Le funzioni utilizzate per ricevere i valori, sono le seguenti:

```
int secure_scalar_receive_matrix(int rcv_sockfd, SSL* ssl,
                            struct sockaddr *sock_address,
                    unsigned long **matrix, int m, int n);

int secure_scalar_receive_vector(int rcv_sockfd, SSL* ssl,
                            struct sockaddr *sock_address,
                            unsigned long *vector, int n);

int secure_scalar_receive_scalar(int rcv_sockfd, SSL* ssl,
                            struct sockaddr *sock_address,
                                   unsigned long *scalar);
```

i cui parametri indicano:

- *rcv_sockfd*: il socket su cui stare in ascolto in attesa di ricevere una connessione.
- *ssl*: la struttura utilizzata per ricevere i dati utilizzando il protocollo SSL.
- *sock_address*: la struttura utilizzata dalla funzione di libreria "accept".
- *matrix*, *vector*, *scalar*: puntatori alla matrice, vettore e scalare in cui salvare i valori ricevuti.
- *m*, *n*: indicano la dimensione delle matrici e dei vettori che vengono ricevuti

Anche nella *secure_scalar_product*, esiste una prima fase di inizializzazione in cui i due client inizializzano SSL, preparano le stringhe con i nomi dei file su cui sono presenti le loro chiavi RSA, preparano i contesti utilizzati per creare canali SSL di invio e ricezione, creano il socket da cui riceveranno le richieste di invio dati mettendosi in "listen" su di esso. A seconda che il client sia il primo od il secondo tra gli indirizzi IP presenti nel vettore di indirizzi, il client eseguirà il protocollo essendo il client "A", oppure essendo il client "B" come descritto nei passi dell'algoritmo. Per generare la matrice random "C" utilizzata da entrambi gli host, "A" genera la metà superiore, e la invia a "B", che invece genera la metà inferiore e la invia ad "A", e da questi valori entrambi sono in grado di costruirsi la matrice "C". Dopo aver concordato questa matrice, "A" e "B" eseguono i passi del protocollo, utilizzando le funzioni descritte in precedenza, ritornando alla fine del protocollo il risultato finale indicante il numero di elementi che sia "A" che "B" possiedono.

3.3 Usabilità

Uno degli obiettivi che erano stati prefissati durate la fase di progettazione, era quello di rendere l'utilizzo di queste funzioni il più semplice possibile. Si è così cercato di mantenere i prototipi delle primitive simili, per esempio come parametri da passare durante la loro invocazione, e di nasconderne i dettagli implementativi in modo da presentare a chi utilizza tali funzioni solo le quattro primitive ed una funzione accessoria (quella per fare l'ordinamento degli indirizzi IP). Così facendo, quello che interesserà conoscere agli utilizzatori di tali funzioni sarà solo il prototipo delle primitive da

invocare, gli indirizzi IP degli host che partecipano ai protocolli implementati da tali funzioni e di essere in possesso dei certificati pubblici e privati generati per il loro indirizzo IP e del file contenente il certificato pubblico di tutti gli altri host.

Per semplicità riassumiamo brevemente i passi che devono essere seguiti per l'utilizzo di tali funzioni in ambito distribuito (per esempio all'interno di un programma che abbia bisogno di effettuare operazioni sicure di somma, unione, intersezione e prodotto scalare tra i vari host):

1. Tutti gli host devono essere in possesso di un file contenente gli indirizzi IP degli host che partecipano al protocollo distribuito. Supponiamo che il file sia "ip.txt" e che stia nella sottodirectory "certificate" all'interno della directory con i sorgenti e gli eseguibili.

2. Questo file viene consegnato ad un host esterno, fidato da tutti, che utilizzerà lo script "create_cert.sh" con parametro il nome di tale file e creerà i file dei certificati pubblici e privati per tutti gli host ed il file contenente tutti i certificati pubblici (vedi par. 3.2.3). Questi darà a tutti gli host la loro coppia di file di certificati pubblici/privati ed il file "CAfile.pem": questi file andranno nella sottodirectory "certificate" di ogni host.

3. A questo punto gli host possono utilizzare la funzioni di privacy-preserving. Prima di chiamare però tali funzioni deve essere invocata una volta (per esempio all'inizio del programma, o prima di invocare una di tali funzioni per la prima volta) la:

   ```
   char **ip_sort(char *ip_file, int *hosts);
   ```

   in cui il primo parametro è proprio il nome del file con gli indirizzi IP (in questo caso "ip.txt") ed il secondo una variabile che conterrà il numero degli host. Questa funzione ritorna un array di stringhe ordinato contenente gli indirizzi IP e che può essere utilizzato per invocare le primitive di privacy-preserving.

4. Adesso è possibile invocare le quattro primitive:

```
unsigned long secure_sum(char **ip_addresses,
          int hosts_num, unsigned long value);

unsigned char **secure_union(char **ip_addresses,
 int hosts_num, char **sets, unsigned long *total);

unsigned long secure_intersection(char **ip_addresses,
                           int hosts_num, char **sets);

unsigned long secure_scalar_product(char **ip_addresses,
                             int hosts_num, char *sets);
```

in cui i primi due parametri sono gli stessi per tutte le quattro funzioni e sono i valori ritornati dalla funzione "ip_sort". Gli altri valori sono:

- Per la "secure_sum": "value" è il valore di cui tale host vuole fare la somma. La funzione ritorna la somma finale di tutti i valori.
- Per la "secure_union": "sets" è l'array di stringhe (array terminato dal carattere "NULL") che rappresentano gli elementi di cui si vuole fare l'unione. La funzione ritorna un array di stringhe rappresentanti l'unione degli elementi e scrive in "total" il numero totale degli elementi.
- Per la "secure_intersection": è l'array di stringhe (array terminato dal carattere "NULL") che rappresentano gli elementi di cui si vuole fare l'intersezione. La funzione ritorna il numero indicante la cardinalità dell'intersezione di tutte le stringhe.
- Per la "secure_scalar_product": "set" indica la colonna di valori booleani (tale colonna deve essere espressa tramite i caratteri "0" ed "1") di cui si vuole fare il prodotto scalare. La funzione ritorna il risultato del prodotto scalare tra le due colonne dei due host.

Inoltre ricordiamo che per poter compilare correttamente tutti i file, tramite il comando:

```
make
```

deve essere presente su tutti gli host il pacchetto OpenSSL (quello che occorre è la libreria “ssl” e gli eseguibili forniti da OpenSSL per l’host che genera i certificati).

Nella figura 3.2 viene visualizzata la struttura della libreria di privacy-preserving generata dal processo di compilazione, inserita nel contesto del Data Mining distribuito per metterne in evidenza la sua relazione con l’algoritmo *apriori* e con la libreria OpenSSL.

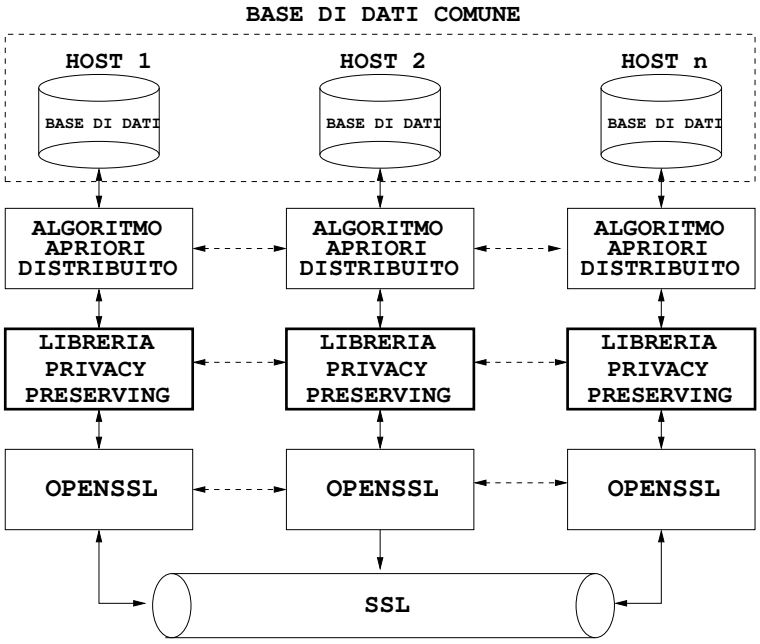

Figura 3.2: Utilizzo della libreria

Capitolo 4

Test

Per effettuare i test, nel caso di dati partizionati orizzontalmente (vedi pag.14), abbiamo utilizzato una versione dell'algoritmo *apriori*, il cui codice (scritto in C++) è disponibile all'url

`http://db.cs.helsinki.fi/~goethals/cgi-bin/apriori.tgz`

Questa versione permette di leggere un file contenente una serie di transazioni, formate da numeri separati da spazi e terminate dal carattere di invio, di specificare un numero indicante il supporto minino che gli itemset devono avere e scrive il risultato, ovvero tutti gli itemsets il cui supporto è maggiore di quello indicato, su un file di output.

Sono state effettuate delle modifiche al codice per permettere di utilizzarlo in ambito distribuito, ovvero in modo che tutte le transazioni presenti sul file fossere suddivise tra più host. Per fare questo, sono state aggiunte le seguenti parti al codice:

- Il supporto viene specificato tramite un numero compreso tra 0 e 100 indicante la percentuale che deve essere superata affinché l'itemset sia considerato supportato globalmente.
- Viene eseguito un conteggio del numero di transazioni e a partire da questo si effettua la Secure Sum per ottenere il numero totale di transazioni, in modo da calcolarsi il supporto locale e quello globale in termini di numero di transazioni.

- L'algoritmo procede normalmente, eccetto che nel momento in cui si calcola il supporto di ogni itemset, se si verifica che un itemset non ha il supporto minimo locale, questi non viene cancellato.

- Per ogni itemset che ha un supporto maggiore di 0 si crea un oggetto "ItemSet" che contiene la rappresentazione di tale itemset ed il suo supporto.

- Si crea un array di stringhe contenenti la rappresentazione degli itemsets che hanno il supporto maggiore od uguale del support locale, e si effettua la Secure Union.

- La Secure Union ritorna un array di stringhe, che identificano l'unione dei locally large itemsets: per ogni itemset, ogni host invia, tramite la Secure Sum, il supporto di tale itemset (0 nel caso non abbia nessuno supporto) per ottenere il supporto globale di tale itemset. Gli host vengono così a conoscenza dei globally large itemsets, e possono adesso eliminare tutti gli altri itemsets.

- A partire dai globally large itemsets, si generano i candidate itemsets, e l'algoritmo prosegue come in precedenza, fino a che non ci sono più globally large itemsets.

Il codice è suddiviso nei seguenti file (le modifiche effettuate nei codici sorgenti sono messe in evidenza tra i tag "/*NEW*/" e "/*END NEW*/"):

```
AprioriSets.cpp  Data.cpp     Item.cpp
AprioriSets.h    Data.h       ItemSet.h
aprioritest.cpp  ItemSet.cpp  Item.h
```

4.1 Prove eseguite

Per effettuare le prove sono stati utilizzati file contenenti molteplici transazioni: questi file fanno parte del repository del FIMI'03 (International Workshop on Frequent Itemset Mining Implementations). In particolare sono stati utilizzati per le prove i file:

```
retail.dat
mushroom.dat
T40I10D100K.dat
```

contenenti rispettivamente 88162, 8124, 100000 transazioni.

Questi file sono stati poi suddivisi tra i diversi host partecipanti all'algoritmo, in modo che ciascuno host possedesse una parte delle transazioni (senza transazioni in comune), e l'unione delle parti formasse il file originale. L'algoritmo è stato dapprima eseguito su un singolo host e, successivamente, in maniera distribuita, controllando che i risultati finali fossero corretti. L'output dell'algoritmo indica, ad ogni passo, il tempo trascorso, il numero di locally large itemsets, il numero di itemsets risultanti dopo la Secure Union, ed il numero di globally large itemsets; gli itemsets frequenti vengono scritti su un file di output, uno per riga, riportando anche il supporto di quell'itemsets. Essendoci una serie di permutazioni all'interno della Secure Union, è necessario, per verificare che il file output dell'algoritmo eseguito su un singolo host e quello generato quando l'algoritmo viene eseguito tra più host, fare un ordinamento dei file (per esempio, tramite l'utility *sort*), ed in seguito effettuare il controllo di uguaglianza (per esempio, tramite l'utility *diff*). Le macchine che sono state utilizzate per fare i test sono quelle che appartengono al cluster del CNR, denominate *cannonau*, *n01*, *n02*,*n03*, *n04*, *n05*, *n06*, *n07*. Nella tabella 4.1 sono visualizzate in forma compatta le prove eseguite. La prima prova è stata effettuata per verificare i tempi di esecuzione nel caso in cui il numero di host partecipanti al protocollo aumentasse. Nel grafico 4.1 sono riportati i tempi parziali per iterazione nel caso di 2, 3, 4, 5, 6, 7, 8 host, utilizzando le transazioni presente nel file *retail.dat*, con un supporto dello 0.2%: vengono anche identificati il numero di locally large itemsets e globally large itemsets generati ad ogni iterazione in quanto i primi sono gli elementi che vengono crittati da tutti gli host ed i secondi sono gli elementi che vengono decrittati da tutti. Questi incidono quindi sui tempi di esecuzione in maniera diretta, in quanto i tempi di esecuzione crescono al crescere degli itemsets generati essendoci più crittazioni/decrittazioni da effettuare per ogni host. La chiave RSA utilizzata per questa prova era di dimensione 512 bits (si considera "sicura" una chiave di 512 o, meglio, di 1024 bits). Come si vede dal grafico, le prime iterazioni, quelle che hanno il numero maggiore di locally e globally

N. di Host	Supporto	Dimensione chiave RSA	Tempo di esecuzione	File
3	1%	1024 bits	62 sec	retail.dat
3	0,2%	1024 bits	959 sec	retail.dat
2	0,2%	512 bits	244 sec	retail.dat
3	0,2%	512 bits	344 sec	retail.dat
4	0,2%	512 bits	506 sec	retail.dat
5	0,2%	512 bits	713 sec	retail.dat
6	0,2%	512 bits	971 sec	retail.dat
7	0,2%	512 bits	1149 sec	retail.dat
8	0,2%	512 bits	1432 sec	retail.dat
3	50%	1024 bits	49 sec	mushroom.dat
3	50%	512 bits	18 sec	mushroom.dat
3	33%	1024 bits	431 sec	mushroom.dat
3	33%	512 bits	152 sec	mushroom.dat
3	30%	1024 bits	673 sec	mushroom.dat
3	30%	512 bits	237 sec	mushroom.dat
3	27%	1024 bits	1002 sec	mushroom.dat
3	27%	512 bits	352 sec	mushroom.dat
3	2%	1024 bits	610 sec	T40I10D100K.dat
3	1,5%	1024 bits	1750 sec	T40I10D100K.dat
3	1,4%	1024 bits	2212 sec	T40I10D100K.dat
3	1,3%	1024 bits	2786 sec	T40I10D100K.dat

Tabella 4.1: Prove effettuate per dati partizionati orizzontalmente

large itemesets, sono quelle più lente; inoltre, all'aumentare del numero di host e di conseguenza, all'aumentare del numero di locally large itemsets e quindi del numero di crittazioni/decrittazioni da effettuare (ricordiamo che ogni host deve crittare tutti gli itemsets degli altri host), i tempi di esecuzione crescono in maniera proporzionale. Il grafico della funzione è pressochè identico in tutti e quattro i casi, eccetto per una differenza proporzionale al numero di locally large itemset ed al numero di host.

Nel grafico 4.2 viene mostrato il tempo finale delle prove eseguite con un numero di host differente. Come si vede i tempi aumentano in maniera sopra-lineare, e questo per due fattori: aumentano il numero di locally large itemsets (che vanno crittati ed inviati), ed aumentano il numero di host che devono effettuare le fasi di crittazione/decrittazione e quindi moltiplicando i due fattori i tempi crescono in maniera non proporzionale rispetto al numero di host (ma proporzionale rispetto al numero totale di locally large e globally large itemsets).

La prova successiva, eseguita con una chiave RSA di 1024 bits, è servita appunto per mettere in evidenza la relazione tra tempo di esecuzione e numero di itemsets da

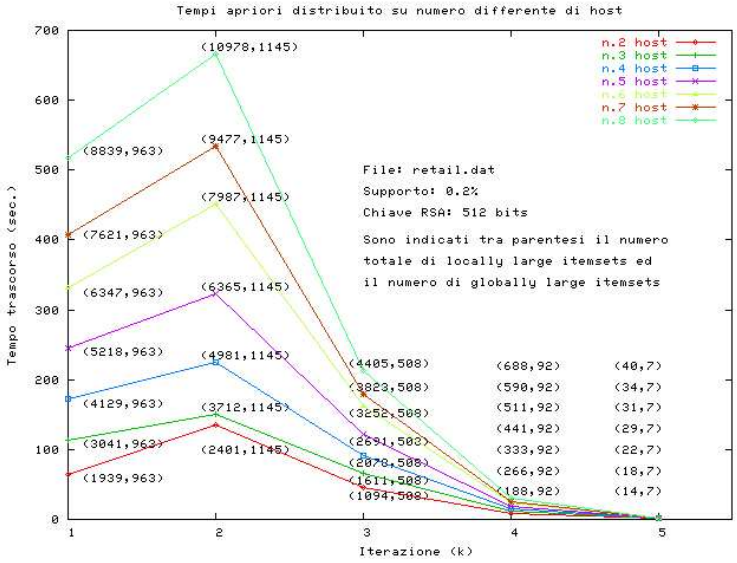

Figura 4.1: Tempi parziali con numero di host differente

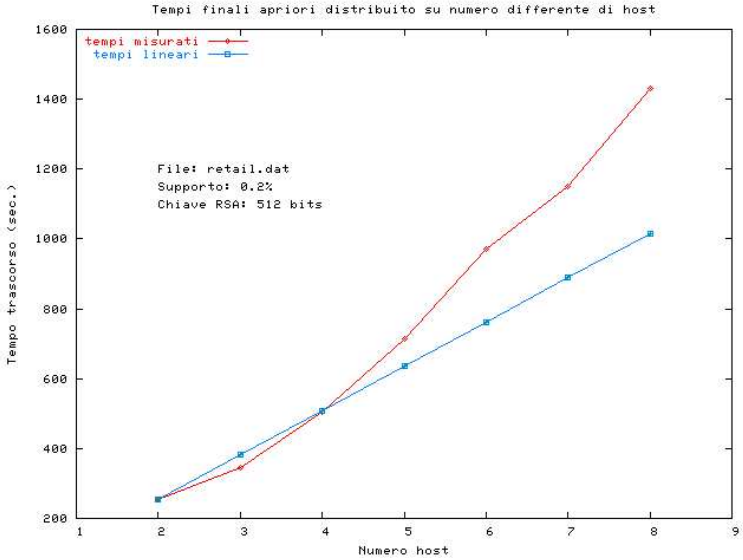

Figura 4.2: Tempi finali con numero di host differente

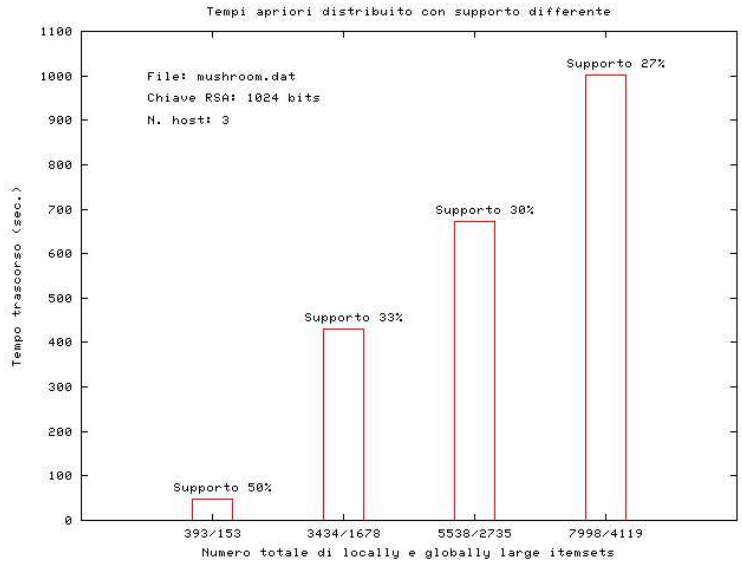

Figura 4.3: Tempi finali con supporto differente (1)

crittare: per aumentare questi è stato via via diminuito il supporto (espresso in termini di percentuale) che ogni itemset deve verificare, ottenendo così un numero maggiore di itemsets (ed un numero maggiore di iterazioni da effettuare). Il file utilizzato è *mushroom.dat* (questo file ha dei supporti molto alti), ogni volta con un supporto differente, rispettivamente di 50%, 33%, 30% e 27% (si veda il grafico 4.3); il numero di host coinvolti nel protocollo era di 3. Si vede che il tempo di esecuzione è proporzionale al numero di locally e globally large itemsets totali creati ed è quasi totalmente determinato da essi: infatti la crittatura/decrittatura di tale insiemi è molto pesante dal punto di vista di calcoli effettuati. Infatti, l'RSA viene utilizzato normalmente con esponente pubblico molto piccolo proprio per effettuare velocemente la fase di crittatura e viene usato per scambiare solo la chiave di sessione, per esempio all'interno del protocollo SSL, mentre per quanto detto in precedenza a proposito della crittografia commutativa l'esponente pubblico in questo caso è molto grande e quindi i tempi di crittatura aumentano in maniera considerevole.

La stessa prova è stata effettuata sul file *T40I10D100K.dat*: anche in questo caso il numero di host era 3, la dimensione della chiave RSA era di 1024 bits, ma il supporto era di 2%, 1,5%, 1,4% e 1,3% (vedere la figura 4.4). Anche in questo caso, il numero

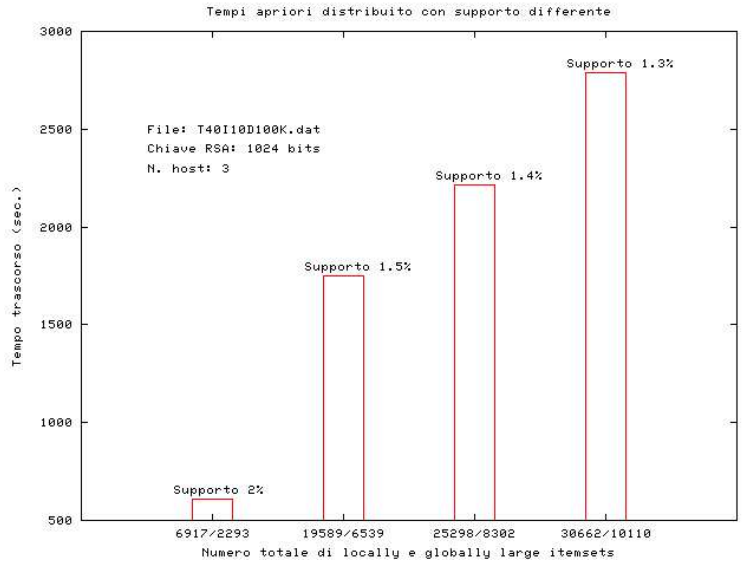

Figura 4.4: Tempi finali con supporto differente (2)

di globally (e locally) large itemsets aumenta ad ogni diminuzione della percentuale del supporto (e gli insiemi sono più numerosi rispetto alla prova precedente): anche qui si nota la relazione diretta tra essi ed il tempo di esecuzione, il che conferma la preminenza nel tempo di esecuzione delle fasi di crittazione/decrittazione degli elementi. Il grafico 4.5 riporta l'unione dei risultati delle ultime due prove in maniera compatta, in modo da rendere visibile lo stretto rapporto di proporzionalità esistente tra tempi di esecuzione e numero di elementi da crittare/decrittare, per cui si può dire che i tempi di esecuzione sono quasi unicamente derivati da queste due fasi, per cui ottimizzando queste, si ottengono tempi migliori.

Quanto detto in precedenza viene reso evidente se gli stessi test vengono effettuati con dimensioni di chiavi RSA differenti. La prova successiva, i cui risultati sono visibili nel grafico 4.6, è stata effettuata sul file *mushroom.dat*, con 3 host e con supporto del 27%, e con chiave prima di 1024 bits e poi di 512 bits. Dal grafico si evince che i tempi sono nettamente inferiori nel caso della chiave di 512 bits, grazie al fatto che le fasi di crittatura e decrittatura (che come detto giocano un ruolo preminente all'interno della Secure Union) sono rese più veloci grazie alla minor dimensione della chiave

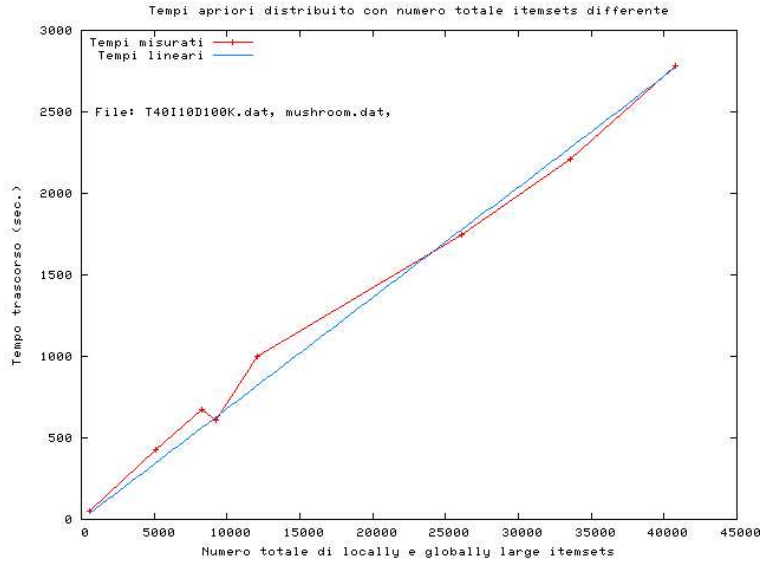

Figura 4.5: Tempi finali con numero di locally/globally large itemsets differente

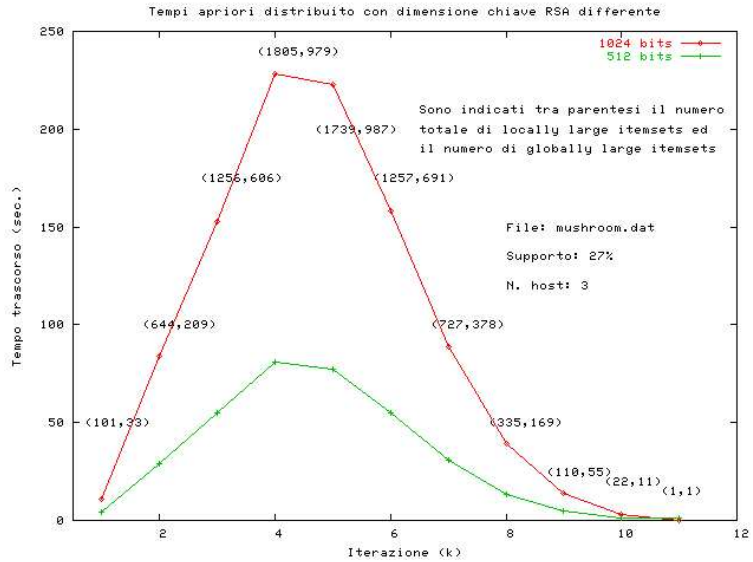

Figura 4.6: Tempi parziali con dimensione chiave RSA differente (1)

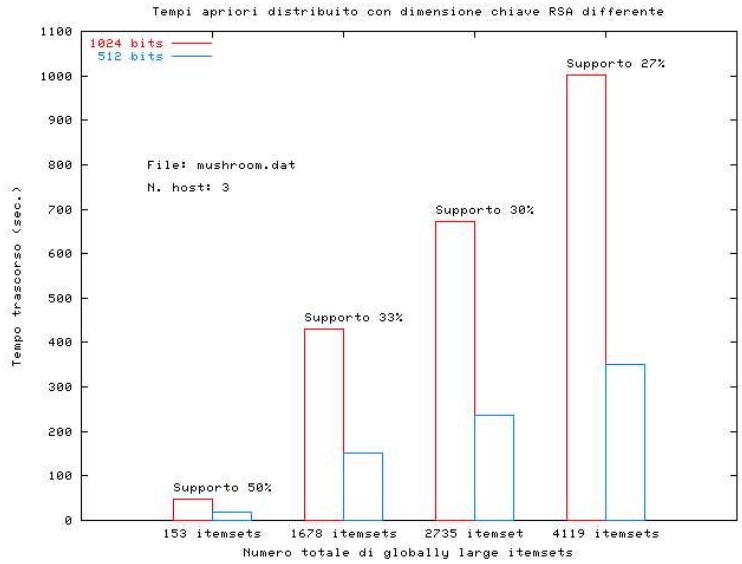

Figura 4.7: Tempi finale con dimensione chiave RSA differente (2)

Nel grafico 4.7 sono riportati i tempi finali utilizzando gli stessi parametri della prova precedente, anche con il supporto del 50%, 33% e 30%: questi tempi non fanno che verificare quanto detto in precedenza, ovvero che i tempi di esecuzione sono notevolmente ridotti (di circa due terzi) al diminuire della chiave da 512 a 1024 bits.

I test fin qui effettuati ci hanno permesso di verificare sia la correttezza che di stimare l'efficienza delle primitive Secure Sum e Secure Union all'interno dell'algoritmo apriori distribuito nel caso di dati partizionati orizzontalmente; per i dati partizionati verticalmente ci siano trovati di fronte ad una considerazione durante la fase di progettazione delle prove: infatti, se si applicasse l'algoritmo specificato nella pubblicazione [2], dovremmo, per ogni riga che contiene il sottoinsieme, creare un oggetto (che rappresenta la riga contenente tale sottoinsieme), crittarlo ed inviarlo e questo per tutti i sottoinsiemi e per tutte le righe che contengono tale sottoinsieme. Ma questo sarebbe molto oneroso dal punto di vista di calcoli da effetturare e quindi il tempo impiegato sarebbe molto alto: per fare un esempio, alla prima iterazione, magari con 100 sottoinsiemi da controllare e 100000 righe (transazioni) bisognerebbe inviare e crittare al piu' $100 * 100000$ elementi (sebbene nel caso pessimo) e questo per tutti gli host. Ora, visti i tempi dei test effettuati con la Secure Union, ci è sembrato opportuno svolgere altri

N. test	Dimensione “n”	Tempo di esecuzione
1	100	1 sec
2	200	1 sec
3	300	2 sec
4	400	2 sec
5	500	5 sec
6	600	6 sec
7	700	10 sec
8	800	12 sec
9	900	15 sec
10	1000	18 sec
11	1100	23 sec
12	1200	26 sec
13	1300	32 sec
14	1400	36 sec
15	1500	48 sec
16	1600	47 sec
17	1700	53 sec
18	1800	60 sec
19	1900	72 sec
20	2000	74 sec

Tabella 4.2: Prove effettuate per la Secure Scalar Product

tipi di test che consentissero ugualmente di verificare la correttezza di tali funzioni e calcolarne l’efficienza, senza però eseguire l’algoritmo proposto su quell’articolo: abbiamo quindi testato le funzioni senza inserirle all’interno di un algoritmo distribuito (come fatto per i dati partizionati orizzontalmente), ma le abbiamo testate singolarmente, ovvero le funzioni sono state invocate più volte con parametri sempre differenti e sono stati successivamente analizzati i risultati ottenuti.

Per effettuare i test sulla primitiva Secure Scalar Product, i file utilizzati sono:

```
test_vector.c
check_vector.c
```

il primo dei quali effettua una serie di chiamate alla primitiva della Secure Scalar Product, generando ogni volta dei vettori random di elementi uguali a “0” oppure “1”, e salvando la loro rappresentazione su file (il cui nome viene passato come parametro).

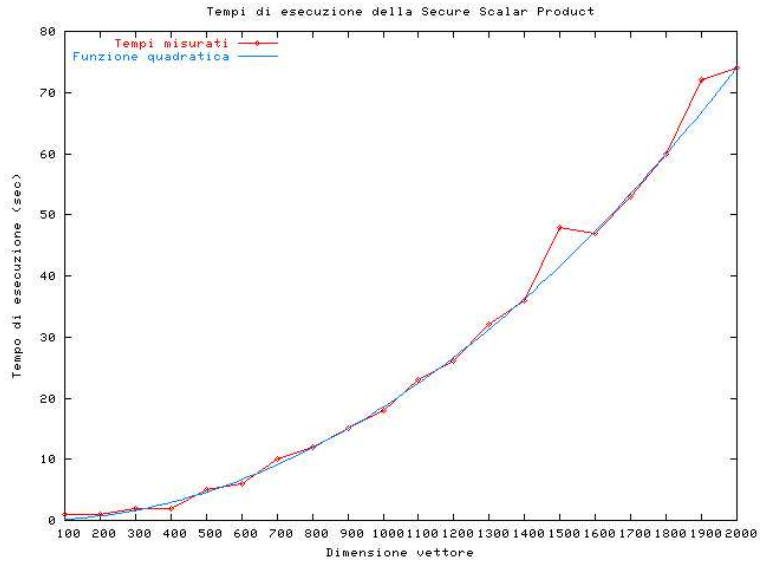

Figura 4.8: Tempi finali della Secure Scalar Product

Viene salvato anche su file il risultato della Secure Scalar Product, e vengono visualizzati a video i tempi ed i risultati dell'algoritmo. Il secondo file serve per controllare l'esito delle prove: esamina i due file generati dagli host, conta il numero di occorrenze di elementi in cui entrambi i vettori sono uguali ad "1", e verifica che questo numero sia uguale a quello calcolato tramite il protocollo della Secure Scalar Product.

Le prove effettuate (vedi la tab. 4.2 per il prospetto) sono state fatte variando la dimensione dei vettori dei due host, tra 100 e 2000 elementi. Nel grafico 4.8 sono riportati i tempi delle prove ed anche il grafico di una funzione quadratica: si vede chiaramente che i tempi sono simili al grafico di questo funzione. Infatti, i tempi di esecuzione sono quadratici perché la funzione prevede la generazione della matrice quadrata (la matrice di elementi random comune accordo) ed i due host si devono inviare i valori di questa matrice e questa fase del protocollo è quella che richiede il tempo di esecuzione maggiore (infatti vengono inviati solamente vettori o scalari durante le fasi successive del protocollo). Quindi, per ridurre i tempi di esecuzione si potrebbe fare sì che i due host prima dell'esecuzione del protocollo siano già in possesso di tali matrici di valori random (magari salvandole su file), in modo da ridurre notevolmente i tempi di esecuzione grazie al fatto che questi valori non vengono spediti

N. test	Numero totale di insiemi	Tempo di esecuzione
1	513	1 sec
2	1017	31 sec
3	1532	39 sec
4	1982	39 sec
5	2277	51 sec
6	2536	51 sec
7	3620	99 sec
8	4074	83 sec
9	4521	99 sec
10	6202	122 sec
11	6836	177 sec
12	8762	234 sec
13	10661	216 sec
14	15692	318 sec
15	21989	675 sec
16	49292	1109 sec

Tabella 4.3: Prove effettuate per la Secure Size of Set Intersection

(in tutto $n * n$ valori, dove n è la dimensione del vettore).

Infine, per effettuare i test sulla primitiva Secure Intersection, il file utilizzato è stato:

`test_intersection.c`

In questo caso le prove avevano solo lo scopo di verificare la correttezza dei risultati ritornati dalla funzione: infatti, essendo la funzione simile alla Secure Union (eccetto che per la parte finale del protocollo), quanto detto in precedenza nel caso dei test effettuati con l'algoritmo *apriori distribuito* riguardo l'efficienza della funzione rimane valido anche per questa funzione, ovvero l'incidenza molto alta sui tempi di esecuzione delle fasi di crittatura (in questo caso non esista la fase di decrittatura degli elementi). Per eseguire i test di questa funzione sono state effettuate una serie di chiamate alla Secure Intersection ogni volta utilizzando come parametro un vettore di stringhe (che rappresentano gli insiemi) creato in maniera random (vengono creati insiemi di tre elementi tramite tre cicli annidati) e sono quindi stati misurati i tempi finali e messi in relazione al numero totale iniziale di insiemi. Il prospetto delle prove è rappresentato nella tabella 4.3, mentre nella figura 4.9 sono visualizzati graficamente i risultati delle prove. Come si vede, i tempi sono strettamente legati al numero iniziale di insiemi:

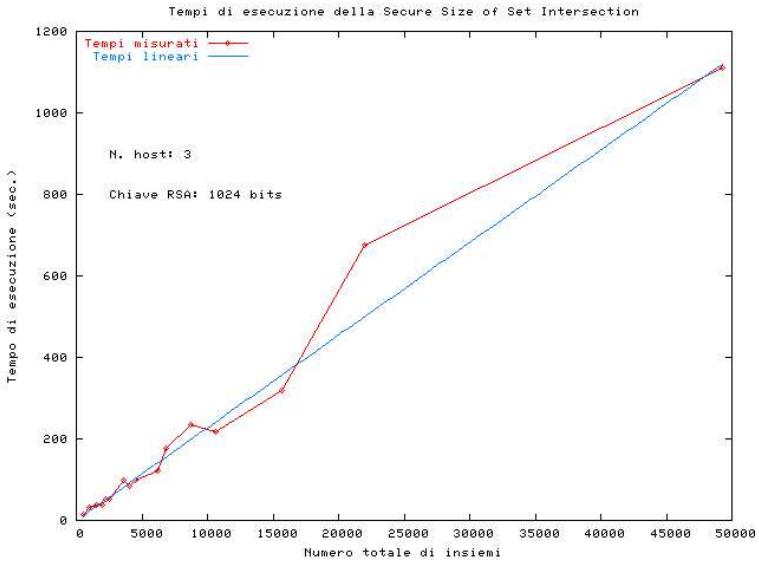

Figura 4.9: Tempi finali della Secure Size of Set Intersection

inoltre questa dipendenza è maggiore rispetto alla Secure Union in quanto in questo caso non esiste la fase di decrittazione finale degli elementi (i globally large itemsets), per cui la cardinalità dell'unione degli insiemi appartenenti a tutti gli host è il fattore principale a determinare il tempo di esecuzione.

Capitolo 5

Conclusioni

In quest'ultima parte della relazione verranno espresse alcune considerazioni relative agli argomenti che ho affrontato durante il periodo di stage ed alle conoscenze che ho acquisite al termine di esso e verrà analizzato il progetto finale cercando di capire se gli obiettivi iniziali sono stati raggiunti

5.1 Conoscenze acquisite durante il tirocinio

Durante il tirocinio sono venuto a contatto con le problematiche del Data Mining, soprattutto nella fasi iniziali, per cui ho dovuto studiare alcuni aspetti di questa disciplina. Dopo aver acquisito conoscenze sufficienti su tale argomento ed aver cominciato la fase di progettazione delle funzioni, ho dovuto prendere confidenza con la libreria OpenSSL sia per utilizzare le funzioni che fanno parte di tale libreria (per es. quelle che che generano byte casuali, quelle per creare un canale SSL, quelle per scrivere e leggere su tale canale etc) sia a livello di modifica del codice sorgente (andavano create le funzioni relative alla crittografia commutativa utilizzando le strutture dati presenti nella libreria). Inoltre, era necessario studiare il funzionamento dell'algoritmo *RSA* e analizzare gli aspetti della crittografia commutativa ed anche il protocollo *SSL*. Per la codifica della libreria in C ho avuto bisogno di apprendere la programmazione di rete in ambito Unix.

Quindi, alla fine del tirocinio le conoscenze che ho acquisito sono state:

- Conoscenze sul Data Mining, in particolare sull'algoritmo *apriori*.

- Conoscenze sull'utilizzo e modifica di una libreria *opensource* composta da molte funzioni e strutture dati.
- Conoscenze sulla crittografia, soprattutto sull'algoritmo *RSA* e sulla crittografia commutativa e sul protocollo *SSL*.
- Conoscenze di programmazione di rete in C.
- Conoscenze relative alla *Secure Multiparty Computation* e sulle possibili soluzioni per garantire la riservatezza dei dati nell'ambito degli Algoritmi di Data Mining distribuito.
- Conoscenze sulla codifica di file di script eseguibili da shell in ambito Linux.

5.2 Considerazioni sul progetto

Gli obiettivi iniziali del progetto erano quelli di creare una libreria di primitive che potessero essere impiegate nell'ambito di algoritmi di Data Mining distribuito, di verificarne la loro correttezza e stimarne l'efficienza tramite una serie di test.

I test effettuati ci permettono di concludere che nel caso dell'algoritmo *apriori* distribuito con file contenenti transazioni che generano molti locally large itemsets (nell'ordine delle migliaia per host), la libreria ha tempi di esecuzione abbastanza lunghi specialmente nel caso di chiave RSA di 1024 bits: con una chiave di 512 bits i tempi diminuiscono in maniera abbastanza evidente. Questo dipende dal fatto che la crittatura e decrittazione attuata tramite RSA è molto lenta, ed è per questo che tale algoritmo viene solitamente impiegato unicamente per crittare la cosidetta chiave di sessione (di solito una chiave di un algoritmo a cifratura simmetrica, come il *3DES*) e si lascia ad altri algoritmi più veloci il compito di crittare/decrittare i dati. Una possibile modifica futura a questa libreria dovrà essere effettuata proprio all'interno di queste fasi di crittazione e decrittazione dei dati, cercando algoritmi commutativi più efficienti dell'RSA (in termini di tempo di esecuzione).

Si è anche visto che aumentando il numero di host i tempi di esecuzione crescono in maniera proporzionale se si considerano il numero totale di elementi da crittare/decrittare ma in maniera sopra-lineare se si considerano il numero di host, per cui l'algoritmo può essere eseguito anche su un numero maggiore di host di quello utilizzato per i test (8 host nel caso massimo) tuttavia i tempi diventano abbastanza lunghi (molto dipenderà anche da come sono distribuite le transazioni tra gli host).

Per quanto riguarda la Secure Scalar Product, abbiamo già osservato che per aumentare notevolmente l'efficienza bisogna fare sì che i due partecipanti del protocollo siano già in possesso di una serie di valori random (scelti di comune accordo) tramite i quali creare la matrice con tali elementi, prima dell'esecuzione dell'algoritmo.

Infine, le prove effettuate sulla Secure Intersection ci hanno permesso ancora di mettere in evidenza la stretta relazione tra tempi di esecuzione e le fasi del protocollo in cui si crittano gli elementi, quindi se si riesce a diminuire i tempi di queste fasi l'algoritmo diventa più efficiente. C'è inoltre anche da ricordare che un piccolo overhead sui tempi di esecuzione di tutte le quattro primitive è dovuto all'utilizzo di un canale SSL, per cui tutti i dati che transitano sulle connessioni di rete vengono crittati ulteriormente.

Per quanto riguarda la correttezza delle funzioni, tutti i test hanno dato risultati positivi, ovvero i file di output del protocollo distribuito contenevano gli stessi insiemi di quelli generati dall'*apriori* su un singolo host, i risultati delle Secure Scalar Product sono sempre stati quelli corretti ed il risultato ritornato dalla Secure Intersection è sempre stato verificato essere esatto. Per questo la libreria risponde ai requisiti di correttezza e di riservatezza dei dati che ci eravamo prefissati come obiettivi da raggiungere.

L'interfaccia della libreria è stata mantenuta il più semplice ed omogenea possibile: le funzioni da chiamare sono solamente le quattro formanti il nucleo della libreria (inoltre queste funzioni hanno parametri simili) più la funzione per ordinare il file di indirizzi IP ed in aggiunta tutti gli host utilizzano tale libreria nella stessa maniera. Per conculdere, la generazione di certificati è facilitata grazie all'utilizzo di uno script-file.

Appendice A

Crittografia commutativa

Un algoritmo crittografico è commutativo se le due equazioni seguenti sono verificate, per ogni chiave possibile $K_1, ..., K_n \in K$, per ogni messaggio $m \in M$, e per qualsiasi permutazione di i, j:

1. $E_{k_{i1}}(...E_{K_{in}}(M)...) = E_{k_{j1}}(...E_{K_{jn}}(M)...)$
2. $\forall M_1, M_2 \in M$, tali che $M_1 \neq M_2$ per ogni k dato, $\varepsilon < \frac{1}{2^k}$,
   $Pr(E_{k_{i1}}(...E_{K_{in}}(M_1)...) = E_{k_{j1}}(...E_{K_{jn}}(M_2)...)) < \varepsilon$

Queste due proprietà permettono, per esempio, di verificare se due messaggi sono uguali, senza però conoscere il contenuto dei messaggi (utile per risolvere il già citato problema del milionario, vedi pag.3): supponendo che due parti, A e B, abbiano due messaggi M_A e M_B che vogliono confrontare, senza però svelare il contenuto dei loro messaggi (cioè, l'unica informazione che vogliono sapere è se i due messaggi sono identici oppure no), la crittografia commutativa può essere utilizzata in questa maniera:

- A critta il suo messaggio M_A con la sua chiave k_A e lo invia a B.
- B critta il suo messaggio M_B con la sua chiave k_B e lo invia a A.
- A critta il messaggio che B gli ha inviato con la sua chiave k_A.
- B critta il messaggio che A gli ha inviato con la sua chiave k_B.
- A questo punto se l'equazione $E_{k_A}((E_{k_B}(M_B)) = E_{k_B}((E_{k_A}(M_A))$ è vera, i due messaggi A e B sono identici.

Alla fine del protocollo, sia A che B hanno visto il messaggio dell'altro solamente in forma crittata.

Il metodo di crittografia a chiave pubblica RSA ([9]), è un algoritmo che risponde ad i suddetti requisiti: ma, invece di utilizzare RSA nella maniera abituale, cioè quella per cui si rende pubblica la chiave e e privata la fattorizzazione di n, bisogna alterare leggermente la procedura ([5]). Le due parti A e B concordano insieme su p e q, e quindi si calcolano $n = pq$; A si genera quindi la coppia di chiavi $((e_A,n),(d_A,n))$ e B altrettanto genera la sua coppia $((e_B,n),(d_B,n))$, entrambi usando lo stesso n. Sia A che B tengono segrete la loro coppia di chiavi, sia quella "pubblica" che quella "privata". Solitamente, usando RSA, si rende pubblica la chiave (e,n), e viene ritenuto computazionalmente difficile ricavare le altre due componenti d e la fattorizzazione di n; in questo caso invece rendiamo pubblico il modulo n e la sua fattorizzazione, ma private e e d.

L'unica restrizione che si pone usando l'RSA in questa maniera, è l'impossibilità di aggiungere come "padding" a fine messaggio una sequenza di bit casuali: così facendo si otterrebbero infatti dopo la crittatura due messaggi, con probabilità molto alta, diversi anche se i due messaggi fossero in realtà uguali (infatti, i bit di "riempimento" sarebbero diversi). Questo rende possibile tentare di forzare la cifratura così ottenuta, con alcuni attacchi noti (vedi pag.126 [4]). Inoltre, un'altra restrizione deriva dal fatto che scegliendo come chiave e un numero primo molto alto (mentre con l'RSA si sceglie di solito un valore basso, per esempio $e = 3$) il processo di crittatura è notevolmente rallentato.

Appendice B

OpenSSL

Il progetto OpenSSL, raggiungibile tramite l'url http://www.openssl.org/, è un *toolkit* Open Source che implementa i protocolli SSL (Secure Sockets Layer) e TLS (Transport Layer Security) e dispone inoltre di una libreria crittografica. In questa appendice verrano mostrate brevemente le funzioni e le strutture dati più comuni della libreria *ssl*, che sono state utilizzate per realizzare le primitive privacy-preserving.

Il primo comando che deve venir utilizzato quando si vuole utilizzare la libreria *ssl* è:

```
int SSL_library_init(void);
```

che serve appunto per inizializzare tale libreria (registrando i cifrari ed i message digest disponibili), ed inoltre

```
SSL_load_error_strings();
```

che servirà, nel caso di errori, per stampare stringhe testuali sullo schermo (questa funzione cioè carica le stringhe che verranno poi stampate in caso di errori della libreria). Altre funzioni che devono essere chiamate in fase di inizializzazione sono le:

```
OpenSSL_add_all_algorithms();
OpenSSL_add_all_ciphers();
OpenSSL_add_all_digests();
```

che servono per caricare tutti gli algoritmi disponibili all'interno di una tabella interna (per un accesso più rapido).

In seguito deve essere creato un oggetto *SSL_CTX*, che permette di stabilire una serie di parametri che verranno utilizzati per creare il canale SSL. Per creare un *contesto* da utilizzare lato client, si esegue la seguente chiamata:

```
ctx = SSL_CTX_new(SSLv3_client_method());
```

se invece il contesto verrà utilizzato lato server la chiamata sarà:

```
ctx = SSL_CTX_new(SSLv3_server_method());
```

La chiamata a tale funzione ritorna un oggetto *SSL_CTX* che viene utilizzato per il canale SSL. Su questo oggetto vengono invocate le funzioni che servono per settare i parametri della connessione SSL, come per esempio l'utilizzo di un certificato, la richiesta di certificati, tramite le funzioni:

```
SSL_CTX_load_verify_locations(SSL_CTX *ctx, const char* CAfile,
                              const char* CApath);
```

che permette di specificare il file contenente i certificati pubblici degli host "fidati", e:

```
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER|
                   SSL_VERIFY_FAIL_IF_NO_PEER_CERT, NULL);
```

che specifica che per tale connessione si richiede al *peer* l'emissione di un certificato, ed in mancanza di tale certificato la connessione SSL verrà chiusa. Inoltre le due funzioni:

```
SSL_CTX_use_certificate_file(SSL_CTX *ctx, const char* file,
                             int type);
SSL_CTX_use_PrivateKey_file(SSL_CTX *ctx, const char* file,
                            int type);
```

permettono di specificare i file contenenti la chiave pubblica e privata che vengono utilizzate durante la fase di *handshaking* del protocollo SSL.

Una volta creata una connessione di rete (per esempio tramite la creazione di un socket), questa può venir associata ad un oggetto *SSL*: prima bisogna creare un oggetto *SSL* tramite la:

```
SSL *SSL_new(SSL_CTX *ctx);
```

funzione che accetta come parametro un oggetto *SSL_CTX*, e ritorna il puntatore ad una struttura "SSL" che eredita tutti i parametri settati all'interno del contesto. Una volta creato questo oggetto, è possibile utilizzare la:

```
int SSL_set_fd(SSL *ssl, int fd);
```

per associare a tale struttura "ssl" il file descriptor "fd": in questa maniera è possibile utilizzare il file descriptor (che sarà verosimilmente un socket descriptor) per operazioni di input/output tramite la struttura SSL che si occuperà di rendere la comunicazione sicura.

Per effettuare le operazioni di *handshake* del protocollo SSL, si possono usare le funzioni:

```
int SSL_accept(SSL *ssl);
```

che aspetta che un client inizi la connessione verso tale canale SSL, e la:

```
int SSL_connect(SSL *ssl);
```

che comincia la fase di handshake con un server.

Quando infine il canale è stato creato, si utilizzano le funzioni:

```
int SSL_read(SSL *ssl, void *buff, int num);
```

per leggere al massimo "num" bytes dal canale "ssl" nel buffer "buf", e:

```
int SSL_read(SSL *ssl, const void *buff, int num);
```

per scrivere "num" bytes dal buffer "buff" nella connessione "ssl".

Bibliografia

[1] D. W. Cheung, J. Han, V. T. Ng, A. W. Fu e Y. Fu. A fast distributed algorithm for mining association rules. In *Proceedings of the fourth international conference on on Parallel and distributed information systems*, pp. 31–43. IEEE Computer Society, 1996. ISBN 0-8186-7475-X.

[2] C. Clifton, M. Kantarcioglu, J. Vaidya, X. Lin, e M. Zhu. Tools for Privacy Preserving Distributed Data Mining. *ACM SIGKDD Explorations*, vol. 4(2), December 2002.

[3] R. Cramer. *Introduction To Secure Computation*. Disponibile su http://www.brics.dk/ cramer/papers/CRAMER_revised.ps, 2000.

[4] P. Ferragina e F. Luccio. *Crittografia: Principi, Algoritmi, Applicazioni*. Bollati Boringhieri, 2001.

[5] M. J. Fisher. *Cryptography and Computer Security: lecture 32*. Yale University, Department of Computer Science, 2003.

[6] M. Kantarcioglu e C. Clifton. Privacy-preserving Distributed Mining of Association Rules on Horizontally Partitioned Data. In *ACM SIGMOD Workshop on Research Issues on DMKD'02*. June 2002.

[7] B. Pinkas. Cryptographic techniques for privacy-preserving data mining. *SIGKDD Explor. Newsl.*, vol. 4(2):pp. 12–19, 2002.

[8] M. R. Fagin e P. Winkler. Comparing information without leaking it. *Communications of the ACM*, vol. 39(5):pp. 77–85, May 1996.

[9] R. L. Rivest, A. Shamir e L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, vol. 26(1):pp. 96–99, 1983.

[10] J. Vaidya e C. Clifton. Privacy Preserving Association Rule Mining in Vertically Partitioned Data. In *The Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Edmonton, Canada, July 2002.

[11] A. C.-C. Yao. How to generate and exchange secrets. *27th Annual Symposium on Foundations of Computer Science*, pp. 162–167, 1986.