

SAPIENZA
UNIVERSITÀ DI ROMA

Introduzione a SPDZ

Laureando
Lorenzo Magliocco

Relatore
Riccardo Lazzeretti

Indice

1. Introduzione
    1.1 Contesto dell'applicazione
    1.2 Contributi
    1.3 Struttura della tesi

2. SPDZ
    2.1 Concetti di base
        2.1.1 Secret Sharing
        2.1.2 MAC
        2.1.3 Fasi di SPDZ
    2.2 Operazioni elementari senza MAC
        2.2.1 Ottenere gli shares di un input
        2.2.2 Somma
        2.2.3 Moltiplicazione
    2.3 Operazioni elementari con MAC
        2.3.1 Ottenere gli shares del MAC di un input
        2.3.2 Somma
        2.3.3 Moltiplicazione
    2.4 Controllo dei MAC
        2.4.1 Associare un MAC agli open values
        2.4.2 MAC check
    2.5 Caso d'esame: Hamming Distance pesata
    2.6 Esempi numerici
        2.6.1 XOR
        2.6.2 XOR con aritmetica modulare

3. Implementazione
    3.1 Tecnologie
    3.2 Struttura generale
    3.3 Rappresentazione dei dati
        3.3.1 Strutture dati
        3.3.2 Iride
    3.4 How to run
    3.5 Traccia di esecuzione

4. Documentazione
    4.1 Root folder
        4.1.1 Defines.h
        4.1.2 Communication.c
        4.1.3 Debug.c
        4.1.4 Common.c
        4.1.5 SPDZ.c
        4.1.6 Converter.c
    4.2 ./structures
        4.2.1 %Array.c
        4.2.2 OpenValArray.c
        4.2.3 RandArray.c
        4.2.4 TripleArray.c
        4.2.5 Iris.c
        4.2.6 HammingDist.c
    4.3 Print pattern

5. Conclusioni

A1. Operatori logici come funzioni

Risorse

Bibliografia

Abstract. In questa tesi presentiamo SPDZ con *Trusted Dealer (TD)*, una soluzione per Secure Multi-Party Computation. In particolare, ne descriviamo le idee ed i meccanismi di base. Ci poniamo in un contesto di two-party computation in ambito biometrico e consideriamo il calcolo dell'Hamming distance pesata di due iridi in SPDZ.
Ne forniamo una implementazione in C, con documentazione completa del codice.
Questa tesi si propone come supporto al lettore che approccia per la prima volta MPC e SPDZ, oltre che come base sulla quale costruire un sistema di autenticazione biometrico sicuro anche in un ambiente in cui tutti i partecipanti fuorché uno deviano dal protocollo.

---

1. Introduzione

Un topic centrale della crittografia è quello della Multi-Party Computation (MPC). Si tratta di un problema in cui n partecipanti possiedono ciascuno un input x_i e vogliono calcolare congiuntamente una funzione $f(x_1, \ldots, x_n)$.

Uno degli esempi più noti è il problema dei milionari, in cui due milionari vogliono sapere chi dei due è più ricco senza però rivelare all'altro quanto possiedono.
Ci sono diverse declinazioni di questo problema che portano a dei risultati pratici: in particolare, laddove sia necessario effettuare dei confronti senza però rivelare gli input dei partecipanti. Si pensi ad esempio ad un sistema elettronico di aste.

Nell'ambito della MPC, il singolo utente non può assumere l'ambiente come affidabile. Di fatti potrebbero esserci dei partecipanti che hanno come unico scopo quello di ottenere informazioni sensibili o di modificare l'esecuzione del protocollo stesso.

Esiste una distinzione teorica tra i vari partecipanti, che si possono definire:

- *honest* se seguono il protocollo;
- *semi-honest* se seguono il protocollo ma tentano di acquisire delle informazioni aggiuntive (ad esempio, l'input di altri partecipanti) semplicemente analizzando il flusso di informazioni a cui hanno accesso;
- *malicious* se deviano dal protocollo arbitrariamente per acquisire informazioni o per corrompere il risultato.

Partecipanti *semi-honest* o *malicious* si dicono *corrupt.*
Possiamo modellare un'entità che sicuramente non è *corrupt* mediante il concetto di *trusted*.

I diversi protocolli possono offrire un certo grado di sicurezza a seconda dell'ambiente di interesse.
Se ad esempio tutti i partecipanti fossero *honest*, non avremmo bisogno di alcuno strumento per garantire l'integrità della computazione o la sicurezza degli input.

In contesti reali è ragionevole pensare di trovarsi in presenza di partecipanti *semi-honest* o *malicious*. Se almeno metà dei partecipanti è *corrupt*, definiamo l'ambiente *dishonest majority*. In questo caso, non esiste alcun protocollo perfettamente sicuro (nel senso di information-theoretic security).
Inoltre i protocolli sicuri per i *malicious* coprono anche i *semi-honest*, ma sono ben più complessi computazionalmente rispetto a quelli realizzati per i soli *semi-honest*.

Considereremo SPDZ [1, 2], un protocollo efficiente in un caso estremo di *dishonest majority* in cui dati n partecipanti, $n - 1$ possono essere *corrupt*. L'ambiente ci impedisce di avere information-theoretic security: dobbiamo considerare l'ipotesi di attaccanti limitati computazionalmente, ossia limitati in termini di potenza di calcolo o di tempo.
Il protocollo è statisticamente sicuro. Tutte le operazioni sono in aritmetica modulare modulo p e la probabilità di successo di un attaccante è inversamente proporzionale a tale valore (spesso 2^{32}, 2^{64}, 2^{128}).

1.1 Contesto dell'applicazione

Sono sempre più i servizi che contemplano la necessità per un utente di doversi autenticare. Spaziano da quelli accessibili a tutti, come i social networks o l'home banking, fino a quelli molto specifici e riservati, come l'accesso ad una sala macchine.
In ogni caso, l'utente deve fornire un input che possa consentirne il riconoscimento attraverso un sistema di autenticazione.

Nei sistemi ad autenticazione mediante password, il server conosce la nostra password, eventualmente come hash. L'autenticazione ha successo solo se forniamo esattamente la nostra password.
Nei sistemi ad autenticazione biometrica invece, il server conosce una istanza della nostra biometria di interesse. Quando vogliamo autenticarci, dobbiamo effettuare una nuova rilevazione della stessa biometria.
Le due rilevazioni però possono differire a causa delle diverse misurazioni: ad esempio, il rilevatore potrebbe essere sporco ed introdurre del rumore, oppure l'utente potrebbe non posizionare la biometria nello stesso modo.
Dobbiamo introdurre quindi una soglia di errore, al di sotto della quale il sistema garantisce comunque l'accesso.

Nonostante la complessità maggiore, esistono molti casi reali in cui è necessaria una autenticazione biometrica. Questo implica, per un utente, l'immettere una rilevazione di una propria biometria in un sistema il quale, a valle di una certa computazione, determinerà se l'accesso è consentito o meno.

La sottrazione di una biometria che non varia in maniera significativa nel tempo (per esempio l'iride o l'impronta digitale) comporta un grave problema di sicurezza. L'attacco potrebbe avvenire per via fisica, sebbene più difficile da eseguire: si pensi al recuperare un'impronta digitale lasciata su un dispositivo o su una superficie per poi replicarla. Potrebbe altresì essere condotto in via digitale, nel caso di un sistema poco sicuro oppure appositamente pensato per acquisire più informazioni del dovuto.

Nel nostro modello il *Server* ha a disposizione un database contenente le features estratte dalle biometrie degli utenti durante una fase di enrollment, mentre il *Client* fornisce una nuova istanza della biometria di un utente. Alla fine della computazione, *Client* e Server devono solo ottenere il permesso di accesso ai servizi (o il diniego) ma il *Client* non deve ottenere le biometrie nel *Server* e il *Server* non deve ottenere né la nuova biometria sottomessa nè l'identificativo del client. Inoltre vogliamo integrità nella computazione, intercettando eventuali deviazioni dal protocollo da parte di qualunque delle due entità.

Come base per tale implementazione, tratteremo il calcolo dell'Hamming distance pesata utilizzando SPDZ. L'Hamming distance pesata tra due iridi è infatti un indicatore del loro grado di similarità. Potremmo ottenere un sistema di autenticazione biometrica unimodale implementando il confronto in SPDZ (non lo vediamo qui), impostando una certa soglia e sfruttando l'Hamming distance che abbiamo calcolato come indicatore per determinare la similarità tra due iridi.

1.2 Contributi

1. Illustriamo SPDZ con *Trusted Dealer (TD)* a scopo didattico, analizzando il protocollo dal punto di vista dei singoli partecipanti ed evidenziando passaggi che potrebbero risultare controintuitivi;
2. Proponiamo una nuova implementazione di SPDZ con *TD*, di cui consideriamo la struttura, le operazioni elementari ed il calcolo dell'Hamming distance pesata di due iridi. Tale implementazione si propone come base per SEMBA [3], un sistema di autenticazione biometrica multimodale che contempla anche l'iride;
3. Presentiamo una documentazione formale completa della suddetta implementazione.

1.3 Struttura della tesi

Nel Capitolo 2 introdurremo SPDZ. Presenteremo poi l'implementazione del nostro caso d'esame: un sistema in SPDZ con *TD* per il calcolo dell'Hamming distance pesata di due iridi tra un *Client* e un *Server*, prima con una descrizione ad alto livello

nel Capitolo 3, poi nel dettaglio mediante una documentazione formale del codice nel Capitolo 4. Infine, nel Capitolo 5 tiriamo le conclusioni, offrendo spunti per espandere il contenuto della tesi verso casi più pratici.

Il codice ed una versione live di questo documento possono essere trovate in Risorse.

2. SPDZ

Introduciamo i concetti base di SPDZ. Dopodiché passiamo alle operazioni elementari prima considerando il caso di sicurezza contro i soli avversari passivi (*semi-honest*), poi anche attivi (*malicious*). Descriviamo in seguito il calcolo dell'Hamming distance pesata, caso d'esame della successiva implementazione. Trattiamo infine la procedura che determina l'integrità di una computazione una volta arrivata a termine.

2.1 Concetti di base

2.1.1 Secret Sharing

Supponiamo di essere il dirigente di una banca con n dipendenti e di avere un codice di lunghezza n necessario per l'apertura di una cassaforte. Vogliamo consentire ai dipendenti di accedervi per poter completare alcune particolari operazioni. Analizziamo diversi scenari, considerando però anche la presenza di dipendenti malintenzionati (li diremo *malicious*, in linea con la terminologia della MPC) che vogliono sottrarre il contenuto della cassaforte.

Scenario 1. Forniamo il codice completo a tutti.
Così facendo però basterebbe un solo *malicious* per compromettere la sicurezza della cassaforte.
Facciamo allora in modo che ognuno degli n dipendenti possegga soltanto un'informazione non completa sul codice, che definiamo **share** del valore segreto in questione. Un tale meccanismo si dice **Secret Sharing.**

Scenario 2. Distribuiamo una cifra del codice per ogni dipendente.
Questa soluzione è sicuramente migliore ma dà ancora luogo a problemi: ogni dipendente conosce una cifra del codice e la relativa posizione. Al crescere del numero di *malicious* una porzione sempre maggiore del codice sarebbe compromessa. Potrebbero infatti collaborare scambiandosi le cifre conosciute, procedendo poi per tentativi nell'indovinare le rimanenti.
Ci interessa dunque minimizzare anche l'informazione contenuta nel singolo share sul valore segreto complessivo.

Scenario 3. Generiamo dei valori random che, sommati, restituiscono il codice e ne distribuiamo uno ad ogni dipendente.
In questo modo per accedere alla cassaforte c'è bisogno necessariamente di tutti gli n dipendenti, e non si ha alcuna informazione sul codice anche se $n - 1$ sono *malicious*. Tuttavia, se anche un solo dipendente non dovesse fornire il proprio share nessuno potrebbe aprire la cassaforte.

Esistono altri metodi per ovviare a questo inconveniente che, nella praticità del nostro esempio, potrebbe ragionevolmente verificarsi (un dipendente dimentica il proprio codice, non vuole fornirlo, ...). Tra di essi citiamo il Secret Sharing di Shamir, un algoritmo con il quale si può ricostruire il codice a partire da un sottoinsieme qualsiasi di k shares degli n totali. In questo modo abbiamo sicurezza fintanto che al più $k - 1$ dipendenti sono *malicious*. Otteniamo quindi una certa ridondanza in cambio di ipotesi più restrittive sulla tolleranza di *malicious* nell'ambiente.

Nella nostra implementazione tutto ciò che genera *TD* nel protocollo (specificheremo cosa in questa sezione) segue l'esempio relativo ai valori random. Non possiamo utilizzare il Secret Sharing di Shamir poiché ci siamo posti in un ambiente di *dishonest majority*. In realtà non ne avremmo nemmeno bisogno, poiché non c'è correlazione tra due esecuzioni distinte del protocollo e non abbiamo bisogno di memorizzare le informazioni generate dal *TD* a lungo termine.

In un ambiente di MPC, i partecipanti avranno bisogno di condividere i propri input per effettuare una computazione congiuntamente. Sfrutteremo uno schema di Secret Sharing lineare. Un partecipante che volesse condividere un proprio input x in un ambiente con n partecipanti dovrà generare degli shares x_i di tale input, condividendo l'i-mo share con l'i-mo partecipante, in maniera tale che:

$$x = \sum_{i=1}^{n} x_i$$

Per generare gli shares utilizzeremo di nuovo valori random, in maniera similare a quanto detto per il *TD*. L'unica differenza è che ogni partecipante dovrà generare uno share anche per sé ed utilizzare quello nelle computazioni successive piuttosto che il valore completo. Il partecipante i-mo dunque dovrà utilizzare x_i invece di x.

2.1.2 MAC

Nel protocollo avremo bisogno di associare ad ogni valore un ulteriore valore che possa fungere da indicatore di integrità: un MAC (Message Authentication Code).
Per **valore esteso** o **rappresentazione estesa** intendiamo una coppia (valore, MAC).
Per generare un MAC di un valore abbiamo bisogno di una **MAC key**.
Sia α una MAC key. Definiamo **MAC** di un valore x la seguente funzione:

$$\gamma(x) := \alpha x$$

Ne sfrutteremo le seguenti proprietà di linearità:

$$\gamma(x + y) = \alpha(x + y) = ax + ay = \gamma(x) + \gamma(y)$$
$$\gamma(ax) = \alpha ax = a\alpha x = a\gamma(x)$$

Avremo bisogno di una ed una sola MAC key.

In un contesto di MPC vogliamo però impedire che i partecipanti possano generare indipendentemente dagli altri dei valori estesi, altrimenti un *malicious* potrebbe deviare dal protocollo modificando arbitrariamente (ed in maniera coerente) anche il MAC, che perderebbe quindi di significato.
Risolviamo il problema assegnando ad ogni partecipante soltanto uno share della MAC key. Nel nostro caso tale chiave è generata (quindi conosciuta interamente) da *TD*, che la ripartisce tra i partecipanti.
Sarà inoltre necessario un seed comune a tutti i partecipanti, a partire dal quale poter generare degli stessi valori random. Ne avremo bisogno per la verifica dell'integrità della computazione, che vedremo in (2.4.2). Anche questo, nell'implementazione, viene generato e comunicato da *TD*.

2.1.3 Fasi di SPDZ

SPDZ è un protocollo a due fasi, che possono susseguirsi ciclicamente:

- *offline phase* (o *preprocessing*), operativa nel momento in cui non ci sono computazioni in corso;
- *online phase* (o *processing*), operativa quando i partecipanti vogliono effettuare una computazione.

Il protocollo permane nettamente per più tempo nell'*offline phase*, dove i partecipanti non richiedono alcun requisito di reattività. In tale fase viene effettuata l'operazione computazionalmente onerosa di generazione del materiale di preprocessing, per migliorare le prestazioni della fase successiva.
Nell'*online phase*, invece, minimizzare il tempo di risposta diventa una priorità. Qui i partecipanti sfruttano il materiale di preprocessing in virtù di questo requisito ed effettuano le computazioni di interesse, comunicando tra di loro.

In questa tesi ci concentreremo solo su somme e moltiplicazioni. Come materiale di preprocessing introdurremo dunque solo la **tripla moltiplicativa**: una tripla (a, b, c) tale che $c = ab$. Ogni partecipante avrà soltanto uno share di ogni tripla.

Come vedremo, i partecipanti sono in grado di effettuare le somme localmente, mentre per ogni moltiplicazione è necessaria comunicazione ed una di queste triple, che viene consumata.

Nel nostro caso, le triple vengono generate e ripartite tra i partecipanti con Secret Sharing una volta per tutte prima dell'inizio delle computazioni dal *TD*, nell'*offline phase*. Anche gli shares della MAC key ed il seed vengono generati e condivisi durante questa fase.

2.2 Operazioni elementari senza MAC

Di seguito ci riferiremo a two-party computation, con partecipanti P_1, P_2, poiché è il contesto che consideriamo nella successiva implementazione. Si possono fare analoghi ragionamenti considerando n partecipanti.

2.2.1 Ottenere gli shares di un input

Supponiamo di essere P_1 e di voler condividere un input x con P_2.
Possiamo generare un valore random r e definire gli shares di x nel seguente modo:

$x_1 = x - r$	$x_2 = r$

È sufficiente comunicare x_2 a P_2.

2.2.2 Somma

P_1, P_2 posseggono rispettivamente gli input x, y e sono interessati a calcolarne la somma $s = x + y$ congiuntamente.
Entrambi generano i due shares e ne comunicano uno solo all'altro partecipante.

$x_1 + x_2 = x$	$y_1 + y_2 = y$

Localmente, avranno a disposizione[1]:

P_1: (x_1, y_1)	P_2: (x_2, y_2)

Possono semplicemente sommare localmente le componenti degli input per ottenere gli shares di s. Sommare s_1, s_2 effettivamente ci restituisce la somma s:

[1] Ovviamente ogni partecipante conosce interamente il suo input, quindi bisognerebbe inserire anche l'altro share tra i valori disponibili localmente. Non viene indicato poiché non ha valenza ai fini del protocollo. Questo vale anche per le successive occorrenze.

P_1: $s_1 = x_1 + y_1$
P_2: $s_2 = x_2 + y_2$

$s = s_1 + s_2 = x_1 + y_1 + x_2 + y_2 \Rightarrow s = x + y$

2.2.3 Moltiplicazione

P_1, P_2 posseggono rispettivamente gli input x, y e vogliono calcolarne il prodotto $z = xy$.
È necessaria una tripla per ogni moltiplicazione, che diciamo essere $c = ab$.

Supponiamo siano stati generati mediante valori random i seguenti shares:

$$x_1 + x_2 = x \quad y_1 + y_2 = y \quad a_1 + a_2 = a \quad b_1 + b_2 = b \quad c_1 + c_2 = c$$

Siano inoltre $\epsilon = x - a$ e $\delta = y - b$.
Ogni partecipante è in grado di calcolarne localmente i propri shares; vogliamo inoltre che queste quantità siano note a tutti. Per farlo è sufficiente, per ogni partecipante, inviare i propri shares a tutti gli altri.
Definiamo i valori così trovati **open values**. Ogni moltiplicazione ne genera due; li useremo successivamente per il protocollo del MAC check.

P_1, P_2 hanno localmente a disposizione i seguenti valori:

P_1: $(x_1, y_1, a_1, b_1, c_1, \epsilon, \delta)$ $\qquad$ P_2: $(x_2, y_2, a_2, b_2, c_2, \epsilon, \delta)$

Per ottenere z consideriamo la seguente funzione lineare:

$$z = c + \epsilon b + \delta a + \epsilon\delta$$

Possiamo ottenerne degli shares z_1, z_2 effettuando localmente le seguenti somme:

P_1: $z_1 = c_1 + \epsilon b_1 + \delta a_1 + \epsilon\delta$
P_2: $z_2 = c_2 + \epsilon b_2 + \delta a_2$

Si noti che la quantità costante $\epsilon\delta$ sarà sommata da un solo partecipante, scelto arbitrariamente prima dell'inizio della computazione.

Dimostriamo che gli shares così ottenuti siano effettivamente shares del prodotto. Riscriviamo ϵ, δ e sostituiamoli all'interno delle operazioni dei singoli partecipanti:

$$\epsilon = x - a = (x_1 + x_2) - (a_1 + a_2)$$
$$\delta = y - b = (y_1 + y_2) - (b_1 + b_2)$$

$$P_1\colon z_1 = c_1 + x_1b_1 + x_2b_1 - a_1b_1 - a_2b_1 + y_1a_1 + y_2a_1 - b_1a_1 - b_2a_1 + \epsilon\delta$$
$$P_2\colon z_2 = c_2 + x_1b_2 + x_2b_2 - a_1b_2 - a_2b_2 + y_1a_2 + y_2a_2 - b_1a_2 - b_2a_2$$

Sommiamo infine gli shares z_1, z_2 per riottenere z:

$$z = z_1 + z_2 = c_1 + c_2 + x_1(b_1 + b_2) + x_2(b_1 + b_2) - a_1(b_1 + b_2) - a_2(b_1 + b_2) + y$$
$$+ y_2(a_1 + a_2) - b_1(a_1 + a_2) - b_2(a_1 + a_2) + \epsilon\delta$$
$$z = c + (x_1 + x_2)b - (a_1 + a_2)b + (y_1 + y_2)a - (b_1 + b_2)a + \epsilon\delta$$
$$z = c + xb - ab + ya - ab + \epsilon\delta$$
$$z = ab + xb - ab + ya - ab + xy - xb - ya + ab \Rightarrow z = xy$$

2.3 Operazioni elementari con MAC

Il protocollo finora descritto garantisce sicurezza contro partecipanti *semi-honest*. Infatti, gli shares degli input ricevuti sono dei valori random e a seguito di operazioni elementari si ottengono ancora valori random.
Vogliamo ora rendere il protocollo sicuro anche contro partecipanti *malicious*. Lo facciamo introducendo i MAC ed estendendo coerentemente le operazioni elementari.

2.3.1 Ottenere gli shares del MAC di un input

Supponiamo di essere P_1 e di voler condividere un input x con P_2, calcolandone il MAC. P_1, P_2 posseggono rispettivamente gli shares della MAC key α_1, α_2.
Dopo aver generato e condiviso opportunamente gli shares x_1, x_2, ogni partecipante avrà localmente a disposizione:

P_1: (x_1, α_1)	P_2: (x_2, α_2)

Possiamo quindi utilizzare la moltiplicazione[2] in SPDZ per calcolare uno share per ogni partecipante di $\gamma(x) = \alpha x$.

2.3.2 Somma

P_1, P_2 posseggono rispettivamente gli input x, y e gli shares della MAC key α_1, α_2.
Sono interessati a calcolare la somma $s = x + y$ congiuntamente, ottenendo anche uno share dei MAC $\gamma(s)$.
La somma dei valori è stata già descritta in (2.2.2). Consideriamo solo i MAC.
Possiamo applicare la procedura definita in (2.3.1) per ottenere uno share del MAC sia di x che di y. Localmente i partecipanti avranno a disposizione:

P_1: $(\gamma(x)_1, \gamma(y)_1)$	P_2: $(\gamma(x)_2, \gamma(y)_2)$

Dalla somma dei MAC locali ricaviamo il MAC complessivo:

$$\gamma(s) = \gamma(x)_1 + \gamma(x)_2 + \gamma(y)_1 + \gamma(y)_2 = \gamma(x) + \gamma(y) = \gamma(x + y)$$

Introdurre una moltiplicazione comporta una complessità maggiore, vista la necessità di comunicazione. Nell'implementazione, il controllo dei MAC viene fatto soltanto al termine della computazione e non viene calcolato il MAC delle singole addizioni, bensì degli open values. (cf. 2.4.2)

2.3.3 Moltiplicazione

P_1, P_2 posseggono rispettivamente gli input x, y, gli shares di una tripla di riferimento $c = ab$ e gli shares della MAC key α_1, α_2.
Sono interessati a calcolare il prodotto $z = xy$ congiuntamente, ottenendo anche uno share del MAC $\gamma(z)$.

Anzitutto bisogna specificare che le triple seguono la rappresentazione estesa. Consideriamo i MAC di tali triple. *TD* conosce α. Contestualmente alla generazione

[2] Questo è un passaggio cruciale. Nell'implementazione, gli open values dati da questa moltiplicazione NON vengono salvati. (cf. `spdz_genIrisMACValues`)

delle triple, genera anche degli shares di $\gamma(a)$, $\gamma(b)$, $\gamma(c)$ arbitrariamente[3] e li invia a P_1, P_2.
Riprendiamo la funzione lineare vista per il calcolo delle moltiplicazioni:

$$z = c + \epsilon b + \delta a + \epsilon\delta$$

Vogliamo calcolare $\gamma(z)$.
Sappiamo che ϵ, δ sono valori costanti, poiché noti ad entrambi i partecipanti.

$$\gamma(z) = \alpha(c + \epsilon b + \delta a + \epsilon\delta) = \gamma(c) + \epsilon\gamma(b) + \delta\gamma(a) + \alpha\epsilon\delta$$

Possiamo ottenerne degli shares $\gamma(z)_1, \gamma(z)_2$ effettuando localmente le seguenti somme:

$$P_1\colon \gamma(z)_1 = \gamma(c)_1 + \epsilon\gamma(b)_1 + \delta\gamma(a)_1 + \alpha_1\epsilon\delta$$
$$P_2\colon \gamma(z)_2 = \gamma(c)_2 + \epsilon\gamma(b)_2 + \delta\gamma(a)_2 + \alpha_2\epsilon\delta$$

2.4 Controllo dei MAC

Abbiamo visto che ottenere un MAC dopo ogni singola addizione comporta comunicazione aggiuntiva.
Possiamo migliorare nettamente le prestazioni del protocollo considerando gli open values che otteniamo dalle moltiplicazioni.
P_1, P_2, rispetto ad un open value ϵ, conosceranno rispettivamente:

$$P_1\colon (\epsilon, \gamma(\epsilon)_1) \qquad\qquad P_2\colon (\epsilon, \gamma(\epsilon)_2)$$

Rispetto ad un open value, ogni partecipante conosce interamente il valore ma possiede solo uno share del MAC. Questo fenomeno si descrive come **partial opening** di un valore esteso.
Anzitutto vediamo come ricavare $\gamma(\epsilon)_1$, $\gamma(\epsilon)_2$ e poi descriviamo la procedura che ci consente di verificare se qualcuno ha barato nella computazione.

2.4.1 Associare un MAC agli open values

Usiamo la rappresentazione estesa degli input e delle triple moltiplicative.

[3] Implementato mediante valori random.

Possiamo utilizzare la procedura in (2.3.1) per calcolare uno share per ogni partecipante di $\gamma(x) = \alpha x$ e $\gamma(y) = \alpha y$.
TD ha già generato e comunicato a P_1, P_2 degli shares di $\gamma(a)$, $\gamma(b)$, $\gamma(c)$.

Analizziamo nuovamente la situazione locale dei partecipanti rispetto ai MAC:

P_1: $(\gamma(x)_1, \gamma(y)_1, \gamma(a)_1, \gamma(b)_1, \gamma(c)_1)$
P_2: $(\gamma(x)_2, \gamma(y)_2, \gamma(a)_2, \gamma(b)_2, \gamma(c)_2)$

Sappiamo che $\gamma(x) = \alpha x$ e che $\gamma(a) = \alpha a$ (risp. y,b), e che $\epsilon = x - a$, $\delta = y - b$.

$$\gamma(\epsilon) = \gamma(x - a) = \alpha(x - a) = \alpha x - \alpha a = \gamma(x) - \gamma(a)$$

Inoltre, $\gamma(x)=\gamma(x)_1 + \gamma(x)_2$ e $\gamma(a)=\gamma(a)_1 + \gamma(a)_2$ (risp. y,b).

$$\gamma(\epsilon) = \gamma(x) - \gamma(a) = \gamma(x)_1 + \gamma(x)_2 - \gamma(a)_1 + \gamma(a)_2 = (\gamma(x)_1 - \gamma(a)_1) + (\gamma(x)_2$$

Il partecipante i avrà quindi uno share del MAC di ϵ (risp. δ):

$$\gamma(\epsilon)_i = \gamma(x)_i - \gamma(a)_i$$

2.4.2 MAC check

Siamo arrivati al termine della computazione. Ogni partecipante avrà un insieme di n open values O_v e dei relativi shares dei MAC O_γ, così come uno share della MAC key α_i.

Vogliamo verificare che nessuno abbia barato.
TD ha inviato in fase di preprocessing un seed a tutti i partecipanti, che viene utilizzato localmente per generare un array di dimensione n di valori random R.
Il partecipante i avrà dunque localmente le seguenti informazioni:

$$O_v = \{o_1, o_2, \ldots, o_n\} \quad O_\gamma = \{\gamma(o_1)_i, \gamma(o_2)_i, \ldots, \gamma(o_n)_i\} \quad R = \{r_1, r_2, \ldots, r_n\} \quad \alpha_i$$

Si sottolinea che O_v ed R sono comuni ad entrambe le parti.

Ogni partecipante calcola un valore, a, prodotto scalare dei vettori R e O_v:

$$a = \sum_{j=1}^{n} r_j o_j$$

Questo valore sarà lo stesso per entrambi i partecipanti: deriva dal prodotto di valori pubblici[4].
Ogni partecipante calcola inoltre un valore, γ_i, prodotto scalare dei vettori R e O_γ:

$$\gamma_i = \sum_{j=1}^{n} r_j \gamma(o_j)_i$$

A questo punto ogni partecipante può calcolare un valore σ_i come differenza di γ_i e $a\alpha_i$:

$$\sigma_i = \gamma_i - a\alpha_i$$

Questo valore, per costruzione, deve essere proprio uno share di 0.
Ogni partecipante committa[5] il proprio σ_i e quindi riceve il σ_j dell'altro.
Il protocollo rileva correttezza se è verificata la seguente condizione:

$$\sigma_1 + \sigma_2 = 0$$

Dimostriamo ora che σ_i sia effettivamente uno share di 0.
Possiamo semplicemente attuare una serie di sostituzioni:

$$\sigma_1 + \sigma_2 = \gamma_1 - a\alpha_1 + \gamma_2 - a\alpha_2 = \sum_{j=1}^{n} r_j \gamma(o_j)_1 - \alpha_1 \sum_{j=1}^{n} r_j o_j + \sum_{j=1}^{n} r_j \gamma(o_j)_2 - \alpha_2 \sum_{j=1}^{n} r_j o_j =$$

$$= \sum_{j=1}^{n} r_j (\gamma(o_j)_1 + (\gamma(o_j)_2) - \alpha \sum_{j=1}^{n} r_j o_j = \sum_{j=1}^{n} r_j \gamma(o_j) - \sum_{j=1}^{n} r_j \alpha o_j = \sum_{j=1}^{n} r_j (\gamma(o_j) - \gamma(o_j))$$

[4] Noti ad entrambi i partecipanti.
[5] Vedremo nell'implementazione più approfonditamente cosa intendiamo. Intuitivamente, vogliamo che nessuno dei due partecipanti abbia un vantaggio conoscendo prima dell'altro lo share mancante.

2.5 Caso d'esame: Hamming Distance pesata

Nella nostra applicazione, avremo bisogno di calcolare in SPDZ la Hamming distance pesata tra due iridi, rappresentate come specificato in (3.3.2).
Definiamo F_1, F_2 i vettori di iriscode; M_1, M_2 i vettori delle maschere.
Un vettore di iriscode contiene N bit che rappresentano features: $F = \{f_1, f_2, \ldots, f_N\}$.
Un vettore di maschera contiene N bit per che rappresentano la maschera::
$M = \{m_1, m_2, \ldots, m_N\}$.
Possiamo calcolare l'Hamming distance pesata nel seguente modo:

$$HD = \frac{\sum_{j=1}^{N} [(f_{1,j} \oplus f_{2,j}) \wedge (\overline{m}_{1,j} \wedge \overline{m}_{2,j})]}{N - \sum_{j=1}^{N} (m_{1,j} \vee m_{2,j})}$$

In pratica, si tratta di un rapporto tra tutte le evenienze tali per cui la feature j-ma è diversa nelle due iridi ed il numero di features complessivo. Vengono tuttavia ignorate, sia al numeratore che al denominatore, le features per cui è stato rivelato almeno in un'iride un bit di rumore nella maschera, ossia un 1.
Avere due iridi uguali comporta una Hamming distance pari a 0.
Applicando De Morgan, otteniamo $\overline{m}_{1,i} \wedge \overline{m}_{2,i} = \overline{m_{1,i} \vee m_{2,i}}$.
Possiamo a questo punto sfruttare la tabella fornita in (A1) per riscrivere l'Hamming distance pesata come sola combinazione di somme e prodotti. Otteniamo:

$$HD = \frac{\sum_{j=1}^{N} [(f_{1,j} + f_{2,j} - 2f_{1,j}f_{2,j})\,(1 - (m_{1,j} \vee m_{2,j})]}{N - \sum_{j=1}^{N} (m_{1,j} \vee m_{2,j})}$$

con $(m_{1,j} \vee m_{2,j}) = m_{1,j} + m_{2,j} - m_{1,j}m_{2,j}$.
Vista la complessità derivante dalla moltiplicazione in SPDZ, è importante calcolare una sola volta l'OR.

2.6 Esempi numerici

Siano P_1, P_2 due partecipanti.

Supponiamo che abbiano rispettivamente un bit x ed un bit y, e che ne vogliano calcolare congiuntamente lo XOR.
Dobbiamo quindi valutare la funzione $x + y - 2xy$. (cf. A1)
Ogni partecipante genera due shares del proprio input, (x_1, x_2) e (y_1, y_2), e ne invia uno (ad esempio, il secondo).
Dalla fase di preprocessing, sono state generate delle triple moltiplicative. Ne consideriamo una sola, $c = ab$, in quanto lo XOR come funzione prevede una sola moltiplicazione. P_1 e P_2 hanno ottenuto il proprio share di tale tripla; rispettivamente, (a_1, b_1, c_1) e (a_2, b_2, c_2).
A questo punto, P_1, P_2 aprono $\epsilon = x - a$ e $\delta = y - b$, ossia calcolano $\epsilon_i = x_i - a_i$ (risp. δ) ed inviano il proprio share all'altro partecipante. Dunque entrambi conoscono ϵ, δ.
P_1, P_2 possono ora calcolare il proprio share della moltiplicazione seguendo la funzione lineare vista in (2.2.3).
Entrambi i partecipanti hanno il proprio share di ogni operando della funzione XOR. Possono quindi sommarli localmente per ottenere uno share dello XOR.
La somma di tali shares è proprio lo XOR.

Per il calcolo e la verifica dei MAC, si parte col sacrificare due triple moltiplicative per ottenere degli shares di $\alpha x, \alpha y$. Si genera un vettore random di 2 elementi (abbiamo fatto una sola moltiplicazione dove salviamo gli open values) ed applichiamo la procedura descritta in (2.4.2) per verificare il risultato.

Seguono due esempi numerici, in cui vedremo solo la parte relativa al calcolo del risultato. Non consideriamo il calcolo e la verifica dei MAC, che lasciamo solo descrittivamente.
Nel secondo esempio si tocca il problema della rappresentazione finita numerica.

2.6.1 XOR

Poniamo $x = 1, y = 0$. Ci aspettiamo dunque che $x \oplus y = 1$.

$$x = 1 \;\; -> \;\; (x_1, x_2) = (1, 0)$$
$$y = 0 \;\; -> \;\; (y_1, y_2) = (2, \; -2)$$

$$a = 7 \;\; -> \;\; (a_1, a_2) = (7, 0)$$
$$b = 3 \;\; -> \;\; (b_1, b_2) = (-2, 5)$$
$$c = 21 \;\; -> \;\; (c_1, c_2) = (11, 10)$$

$P_1 : (x_1, y_1, a_1, b_1, c_1) = (1, 2, 7, -2, 11)$
$P_2 : (x_2, y_2, a_2, b_2, c_2) = (0, -2, 0, 5, 10)$

$P_1 : \epsilon_1 = x_1 - a_1 = 1 - 7 = -6$
$P_2 : \epsilon_2 = x_2 - a_2 = 0 - 0 = 0$
$P_1 : \delta_1 = y_1 - b_1 = 2 + 2 = 4$
$P_2 : \delta_2 = y_2 - b_2 = -2 - 5 = -7$

$\epsilon = \epsilon_1 + \epsilon_2 = -6 + 0 = -6$
$\delta = \delta_1 + \delta_2 = 4 - 7 = -3$
$P_1 : z_1 = c_1 + \epsilon b_1 + \delta a_1 + \epsilon\delta = 11 + 12 - 21 + 18 = 20$
$P_2 : z_2 = c_2 + \epsilon b_2 + \delta a_2 = 10 - 30 + 0 = -20$

$XOR_1 = x_1 + y_1 - 2z_1 = 1 + 2 - 40 = -37$
$XOR_2 = x_2 + y_2 - 2z_2 = 0 - 2 + 40 = 38$
$XOR = XOR_1 + XOR_2 = 1$

2.6.2 XOR con aritmetica modulare

Supponiamo operazioni modulo $2^4 = 16$.
Usiamo gli stessi input e la stessa tripla dell'esempio precedente.
Non ricalcoliamo esplicitamente gli shares di ϵ,δ.

$x = 1 \ -> \ (x_1, x_2) = (10, 7)$
$y = 0 \ -> \ (y_1, y_2) = (3, 13)$

$a = 7 \ -> \ (a_1, a_2) = (7, 0)$
$b = 3 \ -> \ (b_1, b_2) = (12, 7)$
$c = 5 \ -> \ (c_1, c_2) = (3, 2)$

$P_1 : (x_1, y_1, a_1, b_1, c_1) = (10, 3, 7, 12, 3)$
$P_2 : (x_2, y_2, a_2, b_2, c_2) = (7, 13, 0, 7, 2)$

$\epsilon = x - a = 1 - 7 = -6 = 10$
$\delta = y - b = 0 - 3 = -3 = 13$

$P_1 : z_1 = c_1 + \epsilon b_1 + \delta a_1 + \epsilon\delta = 3 + 8 + 11 + 2 = 24 = 8$
$P_2 : z_2 = c_2 + \epsilon b_2 + \delta a_2 = 2 + 6 + 0 = 8$

$XOR_1 = x_1 + y_1 - 2z_1 = 1 + 2 - 0 = 3$
$XOR_2 = x_2 + y_2 - 2z_2 = 0 - 2 + 0 = 14$
$XOR = XOR_1 + XOR_2 = 17 = 1$

3. Implementazione

Descriviamo sommariamente l'applicazione, denotando le tecnologie applicate. Forniamo poi una panoramica sulla struttura del codice e delle strutture dati utilizzate.
Segue un how to run ed una traccia d'esecuzione.

3.1 Tecnologie

Ambiente di programmazione: **Ubuntu 18.04**.
Linguaggio di programmazione: **C**.
La scelta di C deriva dalla necessità di lavorare a basso livello visto il requisito intrinseco di prestazioni di SPDZ.

3.2 Struttura generale

Il codice comprende tre entità distinte: *Trusted Dealer (TD)*, *Client* e *Server*.
Il *TD* va eseguito per primo. Rappresenta la fase offline di SPDZ, così come la componente di commit-and-open: si occupa di generare gli shares del materiale di preprocessing che saranno poi comunicati rispettivamente al *Client* e al *Server* e di verificare l'integrità della computazione una volta arrivata al termine.
Il *Server* rappresenta una banca dati. Nell'implementazione, possiede una sola iride.
Il *Client* rappresenta il soggetto che vuole calcolare l'Hamming distance tra la propria iride e quella del server.
Se il *Server* avesse ad esempio più iridi, il *Client* potrebbe voler ottenere l'Hamming distance minore tra la propria iride e tutte quelle del *Server*. Di fatto si può implementare un meccanismo di autenticazione.

3.3 Rappresentazione dei dati

Implementiamo la **rappresentazione estesa** come una coppia (`int` **`value,`** `int` **`MAC`**).
Per tutte le strutture dati che seguono si utilizza internamente `int`.
Le operazioni saranno quindi svolte in aritmetica modulare in un intervallo $(-\ 2^{31},\ 2^{31}-1)$.

3.3.1 Strutture dati

Nell'implementazione utilizziamo degli array per rappresentare le strutture dati che contengono open values, valori random e le triple. Viene attuato un trade-off tra flessibilità e prestazioni. La pre-allocazione di tutto l'array infatti comporta un'unica allocazione di memoria a monte, migliorando le prestazioni (minore overhead in comunicazioni e in accesso).

3.3.2 Iride

Possiamo rappresentare un'iride come una sequenza di bit.
Un'iride consta di un **iriscode**, che contiene una rappresentazione dell'iride, e di una **maschera**, che modella i rumori presenti all'atto della rilevazione.
Ogni bit dell'iriscode è detto **feature**.
Definiamo **irisfile** un file testuale che contiene esclusivamente caratteri '0' e '1', che rappresentano rispettivamente i bit 0 e 1 dell'iride.
In particolare: sia n il numero di caratteri contenuti in un irisfile. I primi $n/2$ caratteri rappresentano l'*iriscode*, gli altri la *maschera*.
Definiamo **irisfile valido** un irisfile che deve avere un determinato numero di *features*. Non approfondiamo qui la generazione dell'*iriscode* e della *maschera* di un'iride a partire dalla rilevazione. Infatti, ci vengono forniti dei file di input con un numero di *features* ben definito a monte.

3.4 How to run

`make` genererà tre eseguibili in una cartella bin, che andranno lanciati nel seguente ordine:

```
./dealer
./server <irisfile>
./client <irisfile>
```

Se `DEBUG = 1`, `irisfile` non è necessario.
Bisognerà attendere che il dealer finisca di generare il materiale di preprocessing prima di lanciare server e client.

3.5 Traccia di esecuzione

TD genera le triple, la MAC key ed i relativi shares. Genera inoltre un seed comune ai due partecipanti, necessario per generare localmente un vettore di valori casuali comune.

Server si connette a *TD* e ottiene i suoi shares. *Client* si connette a *TD*, ottiene i suoi shares, e si connette a *Server*.
Client e *Server* leggono la propria iride (contenente *iriscode* e *maschera*) da un file. Qualora l'irisfile non fosse valido, si tenta una correzione per renderlo valido, rimuovendo tutti i caratteri diversi da '0' ed '1'. Se la correzione fallisce, il programma termina.
Altrimenti il *Client* (risp. *Server*) genera due shares della propria iride (in maniera random) e ne invia uno al *Server* (risp. *Client*).
Nota: il controllo è presente soltanto nel caso in cui `DEBUG = 0`.

Ci sarà comunicazione per ogni moltiplicazione, mentre le addizioni si possono eseguire localmente.
Client e *Server* calcolano uno share del MAC di ognuna delle due iridi (mediante una moltiplicazione), che servirà per calcolare tutti i successivi MAC degli open values.
A questo punto *Client* e *Server* calcoleranno congiuntamente l'Hamming distance (localmente ne otterranno uno share).

Client e *Server* calcolano un valore sigma_i locale che per costruzione deve essere uno share di 0 e comunicano i propri share di sigma_i e dell'Hamming distance al *TD*. Una volta ricevuti, *TD* comunica lo share di sigma_i mancante al rispettivo partecipante: *TD* qui ha funzionalità di commit-and-open.
Client e *Server* verificano che gli share sommino effettivamente 0; in tal caso, la funzione è stata presumibilmente calcolata correttamente e inviano un acknowledgement positivo al *TD*.
Se *TD* riceve due acknowledgement positivi, apre gli shares dell'Hamming distance mancante al rispettivo partecipante. *TD, Server, Client* terminano, con *Server* e *Client* che stampano l'Hamming distance calcolata.
In caso contrario, almeno un partecipante non ha seguito il protocollo. *TD* dunque termina senza aprire gli shares dell'Hamming distance.

4. Documentazione

La documentazione dell'applicazione copre tutti i moduli e tutte le funzioni implementate.
Sono presenti indicazioni sul contenuto e sulle finalità di ogni modulo, nonché una descrizione formale di ogni funzione.
Il codice è valgrind-proof, e sono presenti diverse verbosità per una migliore comprensione.

4.1 Root folder

Client.c, **Server.c** e **Dealer.c** corrispondono alle tre entità della nostra applicazione, e seguono la traccia d'esecuzione descritta in (3.5).

4.1.1 Defines.h

Qui si trovano tutti i defines.
I `MAXVAL_*` consentono un migliore controllo dell'esecuzione. Può essere d'aiuto utilizzare valori bassi nel caso in cui si voglia accedere a delle print di debug per osservare i valori numerici.
Si suggerisce di non modificare quelli in grassetto.

`SERVER_ADDR: indirizzo IP del Server.`
`SERVER_PORT: porta del Server.`
`DEALER_ADDR: indirizzo IP del Dealer.`
`DEALER_PORT: porta del Dealer.`
`BUFFER_SIZE``: dimensione in bytes del buffer di comunicazione.`

`MULT_SAVEOPEN``: valore dell'opzione SAVEOPEN di` *`spdz_mult`*`.`

`MAXVAL_TRIPLE: modulo dei rand() utilizzati nella generazione di triple (`*`tripleArray_populate`*`).`
`MAXVAL_RANDARR: modulo dei rand() utilizzati nella generazione di valori random (`*`randArray_populate`*`).`
`MAXVAL_MACKEY: modulo dei rand() utilizzati nella generazione delle chiavi MAC (usato dal TD).`
`MAXVAL_IRISCODE: modulo dei rand() utilizzati nella generazione degli shares dell'iriscode e del MAC di un iride.`

`VERBOSE: impostazione che consente di gestire la verbosità delle stampe. Valori ammissibili:`
`0 -> nessuna verbosità;`
`1 -> un certo grado di verbosità, settare DEBUG_PRINTELEMS con criterio;`
`2 -> stampe molto verbose, da usare con cautela.`
`DEBUG: impostazione booleana che attiva o meno un'esecuzione`

di prova, considerando DEBUG_IRIS_SERVER e DEBUG_IRIS_CLIENT. **DEBUG_IRIS_SERVER:** l'irisfile che utilizza il server nell'esecuzione di prova. **DEBUG_IRIS_CLIENT:** l'irisfile che utilizza il client nell'esecuzione di prova. DEBUG_PRINTELEMS: cf. print pattern.
MAX_TRIPLES: numero di triple che vengono generate ed inviate dal TD e ricevute dai partecipanti. Settare un numero sufficientemente alto per garantire di avere abbastanza triple per portare a termine le computazioni.
IO_BUFFER_SIZE: dimensione in bytes del buffer utilizzato dal convertitore.
FILENAME_MAXCONVLEN: lunghezza massima del filename di output. FILENAME_MAXLEN: lunghezza massima del filename di input. FILENAME_CONVSTR: stringa a cui viene appeso il nome di un input file se viene convertito.
ID_SERVER: determina l'ID del Server. Usato in *Party*. **ID_CLIENT:** determina l'ID del Client. Usato in *Party*.

4.1.2 Communication.c

In questo modulo sono definite le funzionalità base per la comunicazione tra le varie entità: principalmente wrapper di system call. Inoltre, invio e ricezione di tipi primitivi di interesse per l'applicazione.

```
int net_bind(int port);
```

Wrapper di **bind**(2).
Crea un nuovo socket, tenta di farne il bind su **port** e mette tale socket in listen.
Se ha successo, restituisce un descriptor del socket.
Altrimenti termina il programma con un messaggio di errore.

```
int net_accept(int socket_desc);
```

Wrapper di **accept**(2).
Tenta di fare accept su **socket_desc**.

Se ha successo, restituisce un descriptor relativo al socket creato rispetto alla connessione entrante.
Altrimenti termina il programma con un messaggio di errore.

```
int net_connect(const char* IP, int port);
```

Wrapper di **connect**(2).
Tenta di fare connect verso l'ip **IP** e la porta **port**, in modalità TCP.

Se ha successo, restituisce un descriptor relativo al socket usato per la connessione.
Altrimenti termina il programma con un messaggio di errore.

```
int net_send(int to, void* data, int size, int flags);
```

Wrapper di **send**(2).
Tenta di scrivere **size** bytes di **data** sul descriptor **to**.
flags viene gestito da **send**(2). Su interruzione **EINTR** continua l'invio.
Viene invocato **send**(2) 1+(**size**/NET_BUFFER_SIZE) volte. Tale costante è definita in *Defines.h*.

Se ha successo, restituisce il numero di bytes inviati.
Altrimenti termina il programma con un messaggio di errore.

int **net_recv**(int **from**, void* **buf**, int **size**, int **flags**);
Wrapper di **recv**(2). Tenta di leggere **size** bytes dal descriptor **from** nel buffer **buf**. **flags** viene gestito da **recv**(2). Su interruzione **EINTR** continua la ricezione. Viene invocato **recv**(2) 1+(**size**/NET_BUFFER_SIZE) volte. Tale costante è definita in *Defines.h*. Se ha successo, restituisce il numero di bytes ricevuti. Altrimenti termina il programma con un messaggio di errore.

void **net_sendInt**(int **val**, int **to**);
Invia l'intero **val** al descriptor **to**. Invoca **net_recv**.

int **net_recvInt**(int **from**);
Restituisce un intero ricevuto dal descriptor **from**.

4.1.3 Debug.c

Funzioni di debug, utilizzate per il controllo della consistenza del risultato tra una valutazione di funzione in SPDZ e in chiaro.

`HammingDistance* debug_hd(Iris* iris1, Iris* iris2);`
Utilizzata quando le due iridi sono in chiaro. Restituisce l'Hamming distance tra **iris1** e **iris2**.

4.1.4 Common.c

Funzioni generiche.

`void handle_error(const char* msg);`
Stampa un messaggio d'errore con **perror**(2), con argomento **msg**. Termina il programma.

`int check_features(int n);`
Controlla che **n** sia un numero di features congruo. I valori ammissibili sono definiti in un array **valid_features**, presente nello stesso file. Restituisce 0 se il numero di features è valido, -1 altrimenti.

4.1.5 SPDZ.c

Ogni entità che vuole accedere al calcolo congiunto in SPDZ dovrà creare un oggetto *Party*. Nella nostra implementazione stiamo considerando two-party computation.
Ogni entità deve avere un identificatore `ID`. Abbiamo definito (in `Defines.h`):

- `ID_SERVER 0`
- `ID_CLIENT 1`

Questa informazione è fondamentale: nel protocollo si sceglie una sola entità che dovrà aggiungere valori costanti. Di default, è `ID_SERVER`.
Ricordiamo che il vettore dei valori random delle due entità deve essere lo stesso, in quanto generato dallo stesso seed, fornito dal TD.

struct Party
L'oggetto *Party*. *ID*: identificatore dell'entità; *MACKey*: share della MAC key; *Peer*: descriptor dell'altro partecipante al protocollo; *TripleArray*: puntatore al vettore degli shares delle triple moltiplicative; *OpenValArray*: puntatore al vettore degli open values; *RandArray*: puntatore al vettore dei valori random.

`Party* party_create(int ID, int MACkey, int peer, TripleArray* tripleArray, RandArray* randArray, OpenValArray* openValArray);`
Costruttore dell'oggetto *Party*. Inizializza tutti i campi con gli argomenti della funzione. *%Array* devono essere pre-esistenti.

`void party_destroy(Party* party);`
Distruttore dell'oggetto *Party*. Invoca inoltre i distruttori degli oggetti *%Array*.

int* **spdz_mult**(Party* **party**, int **x**, int **y**, int **MAC_x**, int **MAC_y**, int **mode**);
Implementazione del protocollo della moltiplicazione in SPDZ. Si consuma una tripla da **party**->*tripleArray*, si calcolano gli shares di ϵ, δ e avviene comunicazione per ricostruire ϵ, δ localmente. Se il **party** ha come *ID* ID_SERVER, aggiunge le componenti costanti. **mode** è una mask che mette a disposizione le seguenti opzioni: 0 - nessuna opzione 1 (MULT_SAVEOPEN) - salva gli open values ϵ, δ ottenuti dalla moltiplicazione in **party**->*openValArray*. Restituisce una struttura allocata dinamicamente int[2] **res** tale che: **res**[0] contiene uno share del risultato della moltiplicazione **xy**; **res**[1] contiene uno share del MAC del risultato della moltiplicazione **xy**. In questo calcolo si usano **MAC_x** e **MAC_y**.

void **spdz_genIrisMACShares**(Party* **party**, Iris* **iris**);
Riempie i campi relativi al MAC dell'iride **iris** rispetto a *iriscode* e *mask* con uno share del MAC. Questa inizializzazione è fondamentale, in quanto si ottiene uno share del MAC della nostra iride a partire dal quale potremo riempire il vettore degli open values. Per farlo sfrutta **spdz_mult** senza salvare gli open values[6]. Questo perché si genererebbe un meccanismo ricorsivo senza scopo (MAC di un MAC).

HammingDistance* **spdz_hd**(Party* **party**, Iris* **iris1**, Iris* **iris2**);
Le due iridi rappresentano degli shares. Restituisce uno share dell'Hamming distance tra **iris1** e **iris2**. Inoltre utilizza **spdz_mult**, che fa side-effect su **party**.

[6] Non c'è perdita di sicurezza nel protocollo: se un partecipante decidesse di barare qui dovrebbe comunque verificare la correttezza dei sigma_i descritta in `spdz_MACCheck` con dei valori arbitrari.

HammingDistance* **spdz_MACCheck**(Party* **party**, int **dealer**, HammingDistance* **hd**);
Implementazione del protocollo del MAC check in SPDZ. In particolare, rispetto all'integrità dell'Hamming distance di cui **party** ha uno share **hd**. Si considerano tutti gli open values ed i relativi MAC utilizzati nella computazione per ottenere localmente uno share di 0: **sigma_i**. Si committa **hd** ed il **sigma_i** mediante comunicazione col TD. Si riceve da esso il **sigma_i** dell'altro partecipante. Se la somma dei due **sigma_i** è nulla, presumibilmente[7] nessuno ha barato e la valutazione della funzione in SPDZ è avvenuta con successo. In tal caso, invia un acknowledgement positivo al TD. Se anche l'altro partecipante ha inviato un acknowledgment positivo, TD invia lo share dell'Hamming distance dell'altro partecipante. Restituisce tale share. Altrimenti invia un acknowledgement negativo al TD e termina il programma con un messaggio di errore. In tal caso TD non apre le Hamming distance.

[7] Un attaccante ha una probabilità non nulla di riuscire a barare.

4.1.6 Converter.c

In questo modulo viene implementato il controllo di un file di input che contiene un'iride. Tale controllo avviene per step. Si accede allo step successivo se il controllo fallisce, e l'eventuale[8] correzione applicata non porta ad un file di input congruo:

1. verifica la correttezza del file di input;
2. prova a rimuovere tutti i caratteri diversi da '0' e '1';
3. termina il programma.

`char* conv_ofname(const char* input_file);`
Genera una nuova stringa allocata dinamicamente che rappresenta il nome del file di output (**o**utput**f**ile**name**) a partire dalla stringa di **input.** In particolare, concatena alla stringa FILENAME_CONVSTR (*Defines.h*) la stringa **input_file.** Restituisce tale stringa.

`char* conv_if(const char* input_file);`
Apre il file **input_file** e lo scandisce. Rimuove qualsiasi carattere che non sia '0' o '1'. Se non è stato rimosso alcun carattere, **input_file** diventa l'eventuale valore di ritorno. Se è stato rimosso almeno un carattere, invoca **conv_ofname** e crea un nuovo file di output con i soli caratteri '0' ed '1'. Il nome di tale file diventa l'eventuale valore di ritorno. Qualunque sia la stringa designata come valore di ritorno, viene invocato **check_features** con argomento il numero di features del file prescelto. Se tale check ha successo, restituisce il nome del file che conterrà un numero congruo di features. Altrimenti, il programma termina stampando un messaggio di errore, rimuovendo l'eventuale nuovo file creato.

[8] La correzione si dice *eventuale* poiché potrebbe succedere che il file di input contenga effettivamente solo '0' e '1' ma non abbia un corretto numero di features. Lo step 2 quindi fallirebbe automaticamente. In quest'ultimo caso nell'implementazione viene saltato lo step 2.

4.2 ./structures

Seguono diverse strutture che gestiamo come oggetti.
Tutti gli oggetti contemplano un'allocazione dinamica mediante un costruttore `*_create`. Ogni costruttore restituisce l'oggetto creato. Per la deallocazione è consigliabile usare il distruttore fornito: `*_destroy`.

4.2.1 %Array.c[9]

RandArray e *TripleArray* non contemplano un campo *nextFree*, a differenza di *OpenValArray*. Questo perché vengono interamente inizializzati, e non viene definita un'operazione di "rifornimento" dell'array. Seguono metodi comuni a tutti gli *%Array*.

`void %Array_print(%Array* arr, int count);`
Si segue print pattern su tutti i campi di tipo array.

`void %Array_destroy(%Array* arr);`
Distruttore dell'oggetto *%Array*.

[9] Usiamo % come wildcard per `OpenValArray`, `RandArray`, `TripleArray`.

4.2.2 OpenValArray.c

Necessario per il check a valle della valutazione delle funzioni in SPDZ. Utilizzato congiuntamente con *RandArray*.
Si noti che ogni partecipante avrà al termine del calcolo gli stessi open values. Tuttavia, soltanto il valore sarà noto ad entrambi, mentre ognuno avrà il proprio share del MAC di tale valore.
Questo è vero poiché abbiamo utilizzato partial opening, ed ogni partecipante ha comunicato all'altro soltanto il proprio share del valore, non del MAC.

struct *OpenValArray*
L'oggetto *OpenValArray* rappresenta la struttura dati che conterrà gli open values di ogni partecipante del calcolo delle funzioni in SPDZ. Gli open values seguono la rappresentazione estesa, e sono quindi presenti i campi *values*, *MAC_values*, due array di int di dimensione *size*. *nextAvailable* è l'indice del prossimo elemento da consumare. *nextFree* è l'indice dove inserire il prossimo elemento. Vengono opportunamente aggiornati in seguito ad operazioni di inserimento o consumo di elementi.

OpenValArray* **openValArray_create**(int **size**);
Costruttore di un oggetto *OpenValArray*. Setta *size* a **size**, *nextAvailable* e *nextFree* a 0. Alloca dinamicamente gli array *values*, *MAC_values* e ne inizializza il contenuto a 0.

void **openValArray_insert**(OpenValArray* **arr**, int **val**, int **MAC_val**);
Se **arr** è già pieno, termina il programma con un messaggio di errore. Altrimenti inserisce un nuovo open value di valore **val** e MAC **MAC_val** in posizione **arr**->*nextFree* rispettivamente degli array **arr**->*values* e **arr**->*MAC_values* e incrementa **arr**->*nextFree*.

4.2.3 RandArray.c

Necessario per il check a valle della valutazione di funzioni in SPDZ. Utilizzato congiuntamente con *OpenValArray*.
L'array generato da entrambi i partecipanti sarà lo stesso, poiché il *TD* ha comunicato lo stesso seed.

struct *RandArray*
L'oggetto *RandArray* rappresenta la struttura dati che conterrà dei valori random. Tali valori sono contenuti in *values*, array di int di dimensione *size*. *nextAvailable* è l'indice del prossimo valore da consumare. Viene aggiornato opportunamente in seguito ad operazioni di consumo di valori.

RandArray* **randArray_create**(int **size**);
Costruttore di un oggetto *RandArray*. Setta *size* a **size** e *nextAvailable* a 0. Alloca dinamicamente l'array *values*.

void **randArray_populate**(RandArray* **arr**, int **seed**);
Setta il seed a **seed**. Popola interamente l'array **arr**->*values* con valori generati casualmente, modulo MAXVAL_RANDARR (*Defines.h*).

4.2.4 TripleArray.c

Necessario per il calcolo di moltiplicazioni in SPDZ.

struct *Triple*
L'oggetto *Triple* è una rappresentazione di una tripla moltiplicativa, che segue la rappresentazione estesa. In particolare, si tratta di una tripla (*a*,*b*,*c*) tale che *c=ab*. Ad ogni elemento della tripla è associato un MAC: rispettivamente, *MAC_a*, *MAC_b*, *MAC_c*.

struct *TripleArray*
L'oggetto *TripleArray* rappresenta la struttura dati che conterrà delle triple moltiplicative. Tali triple sono contenute in *triples*, array di *Triple* di dimensione *size*. *nextAvailable* è l'indice della prossima tripla da consumare. Viene aggiornato opportunamente in seguito ad operazioni di consumo di triple.

TripleArray* **tripleArray_create**(int **size**);
Costruttore di un oggetto *TripleArray*. Setta *size* a **size** e *nextAvailable* a 0. Alloca dinamicamente l'array *triples*.

void **tripleArray_populate**(RandArray* **arr**, int **MACkey**);
Popola interamente l'array **arr**->*triples* con triple generate casualmente, con valori *a*,*b*,*c* modulo MAXVAL_TRIPLE (*Defines.h*). Inizializza inoltre i MAC delle triple a partire dai valori generati casualmente, mediante prodotto con **MACkey**[10].

[10] Solitamente nell'applicazione si tratta di uno share della MAC key.

Triple* **tripleArray_consume**(TripleArray* **arr**, int **count**);
Consuma **count** triple dall'array **arr**->*triples*. Restituisce un puntatore all'array a partire dalla tripla dell'array in posizione **arr**->*nextAvailable*, aggiornando inoltre **arr**->*nextAvailable* a *nextAvailable*+**count**.

TripleArray** **tripleArray_genShares**(TripleArray* **arr**);
Crea due nuovi TripleArray a partire da **arr** e genera gli shares delle triple e dei relativi MAC con valori random. Se è definito DEBUG > 0, controlla la correttezza degli shares generati; Se è definito VERBOSE, si segue print pattern per ottenere la stampa degli shares di entrambi i partecipanti. Restituisce un TripleArray*[2] allocato dinamicamente, di cui ogni elemento contiene uno share di **arr** ed è allocato dinamicamente.

int **tripleArray_send**(TripleArray* **arr**, int **numTriples**, int **to**);
Invia prima **numTriples**, poi **numTriples** triple, a partire dalla prima, del *TripleArray* **arr** al descrittore **to**. Usa **net_send**. Restituisce il numero di triple inviate.

TripleArray* **tripleArray_recv**(int **from**);
Riceve prima il numero di triple, poi le triple dal descrittore **from**. Da queste informazioni ricostruisce un nuovo oggetto *TripleArray*. Usa **net_recv**. Restituisce tale oggetto.

4.2.5 Iris.c

struct *Iris*
L'oggetto *Iris* rappresenta un'iride. La dimensione di un'iride viene qui definita come il numero di features che la compone. Tale informazione è nel campo *size*. L'iriscode e la maschera di tale iride saranno contenuti in degli array di interi: rispettivamente, *iriscode* e *mask*. *iriscode* e *mask* avranno associati rispettivamente due ulteriori array di interi, che ne conterranno il MAC: rispettivamente, *MAC_iriscode* e *MAC_mask*.

Iris* **iris_create**(int **size**);
Costruttore di un oggetto *Iris*. Setta *size* a **size**, *nextAvailable* a 0. Alloca dinamicamente gli array *iriscode*, *mask*, *MAC_iriscode*, *MAC_mask*. Inizializza a 0 il contenuto degli array *MAC_iriscode*, *MAC_mask*.[11]

void **iris_destroy**(Iris* **iris**);
Distruttore di un oggetto *Iris*.

Iris* **iris_read**(const char* **input_file**);
Determina la dimensione dell'iride facendo una seek di **input_file** fino a SEEK_END e resetta l'offset a 0; poi lo legge. Parsa i caratteri '0' e '1' in interi e costruisce una nuova iride, separando iriscode e mask. Restituisce tale iride.

[11] La `calloc` avviene solo sugli array dei MAC. Infatti, si suppone che l'iride venga inizializzata, ad esempio con `iris_read`. Si può tuttavia immaginare un'applicazione in cui i valori dei MAC non vengano considerati, come in un protocollo passivo. Per pulizia, vengono azzerati soltanto quei valori.

Iris* **iris_print**(Iris* **iris**);
Segue print pattern per l'iriscode e per la mask.

Iris** **iris_genShares**(Iris* **iris**);
Crea due nuove Iris a partire da **iris** e genera gli shares dell'iride; in particolare, dei soli valori del vettore di iriscode e mask, non del MAC (cf. spdz_genIrisMACShares). Gli shares sono generati nell'intervallo (-MAXVAL_IRISCODE-1, MAXVAL_IRISCODE). Se è definito DEBUG > 0, controlla la correttezza degli shares generati; Se è definito VERBOSE == 2, si segue print pattern per ottenere la stampa degli shares di entrambi i partecipanti. Restituisce un Iris*[2] allocato dinamicamente, di cui ogni elemento contiene uno share di **iris** ed è allocato dinamicamente.

void **iris_send**(Iris* **iris**, int **to**);
Invia prima il **iris**->*size*, poi l'iride **iris** (iriscode e mask) al descrittore **to**. Usa **net_send**.

Iris* **iris_recv**(int **from**);
Riceve prima il numero di features, poi l'iriscode e la mask dell'iride dal descrittore **from**. Da queste informazioni ricostruisce un nuovo oggetto *Iris*. Usa **net_recv**. Restituisce tale oggetto.

4.2.6 HammingDist.c

struct *HammingDistance*
L'oggetto *HammingDistance* rappresenta un Hamming distance di numeratore *num* e denominatore *den*.

HammingDistance* **hd_create**(int **num**, int **den**);
Costruttore di un oggetto *HammingDistance*. Inizializza il campo *num* a **num** ed il campo *den* a **den**.

void **hd_destroy**(HammingDistance* **hd**);
Distruttore di un oggetto *HammingDistance*.

int **hd_send**(HammingDistance* **hd**, int **to**);
Invia l'*HammingDistance* **hd** al descrittore **to**. Usa **net_send**.

HammingDistance* **hd_recv**(int **from**);
Riceve un *HammingDistance* dal descrittore **from** e ne crea un nuovo oggetto. Usa **net_recv**. Restituisce tale *HammingDistance*.

4.3 Print pattern

Nelle stampe di debug, è implementato un pattern ricorrente.
Sia **arr** un array. `DEBUG_PRINTELEM` è una costante definita in *Defines.h*.

```
Stampa i primi e gli ultimi DEBUG_PRINTELEM elementi di arr.
Se VERBOSE == 2, stampa tutti gli elementi di arr.
Se per tali elementi è definito un MAC, stamperà anche
quello.
```

Nell'implementazione, per accedere alle stampe controllate da `DEBUG_PRINTELEM` è necessario `VERBOSE == 1`. Questo può avvenire o internamente alla funzione stessa (per esempio, in `tripleArray_genShares`) o esternamente (per esempio, in `%Array_print`).

5. Conclusioni

Abbiamo presentato SPDZ con un *TD*, ne abbiamo esplicitato i meccanismi e ne abbiamo considerato un'applicazione in ambito biometrico con il calcolo dell'Hamming distance, con relativa implementazione.
Sono tuttavia possibili diverse migliorie, che ci consentono di avvicinarci di più ad un caso reale.

- Abbiamo implementato SPDZ con un *TD*. Questo comporta l'introduzione di un'entità aggiuntiva che deve essere *trusted*. Per i nostri scopi è sufficiente, e può in realtà essere attuato in taluni contesti reali.
  Per estendere il campo applicativo, è possibile implementare SPDZ in maniera più completa: senza TD. Si rimanda alla Bibliografia.

- Nell'implementazione si assume che il *TD* fornisca abbastanza materiale di preprocessing per la valutazione di funzioni in SPDZ una volta per tutte all'inizio. È ragionevole, visto che vogliamo calcolare soltanto una Hamming distance tra l'iride del client ed una sola iride del server.
  L'implementazione inoltre, segue l'online phase di [2]. Visto che non viene mai rivelata la MAC key α, possiamo riutilizzarla per il calcolo di più funzioni senza il bisogno di generare diverso materiale di preprocessing per ogni valutazione di funzione.
  Si può quindi pensare di introdurre la possibilità per i partecipanti di "rifornirsi" di materiale di preprocessing dal *TD* qualora ce ne fosse bisogno durante la valutazione di una funzione. Questo ha senso, ad esempio, laddove l'iride del client debba essere confrontata con un insieme di iridi del server, oppure nel caso in cui si debbano calcolare funzioni più complesse.
  Il codice comprende solo il calcolo dell'Hamming distance, ma sono presenti tutte le basi per espanderlo.

- Nell'implementazione rappresentiamo le informazioni mediante `int`.
  Si può utilizzare una libreria per la gestione di grandi numeri per considerare campi finiti più estesi. Questo rende il protocollo più sicuro, in quanto riduce le probabilità per un attaccante di barare con successo[12].

[12] È possibile barare con successo nel protocollo MACCheck con probabilità 2/p, con p numero di elementi del campo finito. Per la dimostrazione di questo risultato, si rimanda a [2].

A1. Operatori logici come funzioni

Operatore	Funzione
$a \wedge b$	ab
$a \vee b$	$a + b - ab$
$a \oplus b$	$a + b - 2ab$
$\overline{a}$	$1 - a$

Risorse

Versione live: PDF
Codice: Github, commit di riferimento (02.09.20).

Bibliografia

[1] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, “Multiparty Computation from Somewhat Homomorphic Encryption”, in Safavi-Naini R., Canetti R. (eds) Advances in Cryptology – CRYPTO 2012. CRYPTO 2012. Lecture Notes in Computer Science, vol 7417. Springer, Berlin, Heidelberg.
[2] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, “Practical Covertly Secure MPC for dishonest majority – or: Breaking the SPDZ Limits”, in Crampton J., Jajodia S., Mayes K. (eds) Computer Security – ESORICS 2013. ESORICS 2013. Lecture Notes in Computer Science, vol 8134. Springer, Berlin, Heidelberg.
[3] M. Barni, G. Droandi, R. Lazzeretti, and T. Pignata, “SEMBA: Secure Multi-Biometric Authentication”, in IET Biometrics, vol. 8, no. 6, pp. 411-421, 11 2019, doi: 10.1049/iet-bmt.2018.5138.

.