Università di Catania

Dipartimento di Matematica e Informatica
Corso di Laurea Triennale in Informatica

Giada Rossana Margarone

Vulnerability assessment su libreria Python che implementa Secure Multi-Party Computation

Relazione Progetto Finale

Relatore: Prof. Giampaolo Bella

Anno Accademico 2023 - 2024

A mio nonno Vito,
che mi ha insegnato che
essere curiosi è una virtù.

Abstract

L'esponenziale diffusione di applicazioni che necessitano di protezione di dati sensibili ha portato un interesse crescente verso i protocolli di Secure Multi-Party Computation, che consentono l'elaborazione congiunta di calcoli su dati privati senza rivelare il loro contenuto. Lo scopo di questa tesi è analizzare MPyC, una delle librerie Python più promettenti tra quelle che implementano SMPC, al fine di valutarne la robustezza contro gli attacchi e verificare quanto essa protegga la privacy degli utenti. A tal fine sono stati condotti degli esperimenti su scenari simulati in ambienti distribuiti. I risultati hanno poi evidenziato delle vulnerabilità nella gestione degli input e della loro condivisione, oltre a una limitata robustezza contro azioni non previste dal protocollo. Questo lavoro ha dunque evidenziato la necessità di miglioramenti sia operativi che crittografici.

Indice

Capitolo 1

Introduzione

Al giorno d'oggi, la protezione dei dati personali riveste un'importanza cruciale nelle comunicazioni via Internet, tanto che la crittografia è diventata un campo di studio e applicazione essenziale. Con il rapido progresso tecnologico, anche gli strumenti per la salvaguardia dei dati si sono evoluti per far fronte a minacce sempre più sofisticate.

Ma cos'è esattamente la crittografia? Essa è la branca della crittologia che tratta delle "scritture nascoste", ovvero dei metodi per rendere un messaggio non comprensibile/intelligibile a persone non autorizzate a leggerlo, garantendo così, in chiave moderna, il requisito di confidenzialità o riservatezza tipico della sicurezza informatica. [1] I processi di encryption e decryption avvengono mediante l'utilizzo di algoritmi crittografici e chiavi di cifratura: solo un utente dotato della chiave corretta può decifrare il testo cifrato, ripristinando il contenuto originale. In tal modo, la crittografia garantisce la riservatezza e l'integrità dei dati anche in contesti altamente vulnerabili, come le comunicazioni online.

Tra i protocolli crittografici più avanzati troviamo quelli che includono il Secure Multiparty Computation: essi consentono a un gruppo di eseguire

congiuntamente un calcolo senza rivelare gli input privati dei partecipanti.

I partecipanti si accordano su una funzione da calcolare e poi possono utilizzare un protocollo SMPC per calcolare congiuntamente l'output di tale funzione sui loro input segreti senza doverli rivelare. Introdotto da Andrew Yao negli anni '80, il calcolo multi-party è passato dall'interesse teorico a uno strumento importante per la costruzione di applicazioni su larga scala che rispettino la privacy. [2]

L'obiettivo di questa tesi è analizzare MpyC, una libreria Python particolare che implementa questo protocollo, capirne i funzionamenti, constatare se è vulnerabile a delle parti corrotte e malevole e vedere quanto essa preservi la privacy di chi le utilizza.

Capitolo 2

Secure Multiparty Computation

2.1 Cos'è Secure Multiparty Computation?

Un protocollo crittografico è definito come una serie di passaggi e scambi di messaggi tra più entità per raggiungere uno specifico obiettivo di sicurezza. Secure Multiparty Computation (SMPC) è un protocollo che ha l'obiettivo di distribuire un calcolo tra più parti, ciascuna delle quali non può vedere i dati delle altre parti. [3] In altri termini, SMPC consente a n parti distinte di calcolare congiuntamente una funzione sui loro input privati, garantendo che ciascuna parte conosca solo il proprio input e il risultato finale della computazione. Questo viene realizzato attraverso un sistema di protocolli distribuiti dove:

1. Le parti coinvolte $P_1, P_2, ..., P_n$ possiedono rispettivamente input privati $x_1, x_2, ..., x_n$
2. L'obiettivo è calcolare $f(x_1, x_2, ..., x_n) = y$, dove f è una funzione predefinita

3. Alla fine del protocollo, tutte le parti ottengono y senza aver mai appreso gli input delle altre parti

Questo approccio permette di superare il tradizionale compromesso tra utilità dei dati e privacy: invece di condividere i dati con una terza parte fidata per il calcolo, le parti possono collaborare per ottenere il risultato desiderato mantenendo la riservatezza dei propri dati.

2.1.1 Proprietà fondamentali di SMPC

Le propeietà fondamentali di un protocollo SMPC sono le seguenti:

1. *Privacy*: nessuna parte dovrebbe apprendere più del suo output prescritto. In particolare, le uniche informazioni che dovrebbero essere apprese sugli input delle altre parti sono quelle che possono essere derivate dall'output stesso. Per esempio, in un'asta dove viene rivelata solo l'offerta più alta, è chiaramente possibile dedurre che tutte le altre offerte erano inferiori all'offerta vincente.

2. *Correctness*: a ogni parte è garantito che l'output che riceve sia corretto.

3. *Independence of Inputs*: le parti corrotte devono scegliere i loro input indipendentemente dagli input delle parti oneste. Notiamo che l'independence of inputs non è implicata dalla privacy. Per esempio, potrebbe essere possibile generare un'offerta più alta, senza conoscere il valore di quella originale. Tale attacco può essere effettivamente eseguito su alcuni schemi di crittografia (cioè, data una crittografia di \$100, è possibile generare una crittografia valida di \$101, senza conoscere il valore crittografato originale).

4. *Guaranteed Output Delivery*: le parti corrotte non dovrebbero essere in grado di impedire alle parti oneste di ricevere il loro output. In altre parole, l'avversario non dovrebbe essere in grado di interrompere il calcolo eseguendo un attacco "*denial of service*".

5. *Fairness*: le parti corrotte dovrebbero ricevere i loro output se e solo se anche le parti oneste ricevono i loro output. Lo scenario in cui una parte corrotta ottiene l'output e una parte onesta no non dovrebbe essere permesso. Va notato che il guaranteed output delivery implica la fairness, ma il contrario non è necessariamente vero. [4]

2.1.2 Real-Ideal Paradigm

Il Real-Ideal Paradigm rappresenta un approccio per definire la sicurezza di un protocollo, introducendo un "mondo ideale" che incorpora implicitamente tutte le garanzie di sicurezza. La sicurezza viene così definita confrontando le interazioni del mondo reale con quelle di questo mondo ideale.

Nel mondo ideale, le parti calcolano in sicurezza la funzione F inviando privatamente i loro input a una terza parte completamente fidata T, chiamata *funzionalità*. Ogni parte P_i ha un input associato x_i, che viene inviato a T, la quale semplicemente calcola $F(x_1, \ldots, x_n)$ e restituisce il risultato a tutte le parti.

Possiamo immaginare un avversario che tenti di attaccare l'interazione del mondo ideale. Un avversario può prendere il controllo di una qualunque delle parti P_i, ma non di T (in questo senso, T è descritto come una terza parte fidata). Sebbene il mondo ideale sia facile da comprendere, la presenza di una terza parte completamente fidata lo rende immaginario.

Nel mondo reale, non esiste una parte fidata. Invece, tutte le parti comunicano tra loro utilizzando un protocollo. Il protocollo π specifica per ogni parte P_i una "*funzione di messaggio successivo*" π_i. Questa funzione riceve come input un parametro di sicurezza, l'input privato della parte x_i, un nastro casuale (sequenza di valori casuali) e l'elenco dei messaggi che P_i ha ricevuto finora. Poi, π_i produce o un messaggio successivo da inviare con la sua destinazione, oppure istruisce la parte a terminare con un output specifico.
Nel mondo reale, un avversario può corrompere le parti. Intuitivamente, il protocollo π è considerato sicuro se qualsiasi effetto che un avversario può ottenere nel mondo reale può essere ottenuto anche da un corrispondente avversario nel mondo ideale. [2]

2.2 Stato dell'arte

Attualmente SMPC è stato implementato in protocolli che spaziano in diversi ambiti dove la protezione dei dati è di fondamentale importanza. Tra questi troviamo:

- Analisi finanziarie tra banche concorrenti;
- Ricerche mediche su dati sensibili di pazienti;
- Aste e negoziazioni confidenziali;
- Analisi statistiche su dati privati;
- Machine learning distribuito preservando la privacy.

Le implementazioni di SMPC richiedono tecniche crittografiche avanzate per bilanciare l'efficienza computazionale con la sicurezza. Tra queste, alcune delle più rilevanti includono:

- *Crittografia omomorfica*: permette di eseguire operazioni direttamente sui dati cifrati, senza bisogno di decifratura, mantenendo l'informazione riservata anche durante l'elaborazione.

- *Circuiti Garbled*: rappresentazione crittografica di un circuito logico, in cui gli input e gli output sono codificati, permettendo a due parti di calcolare una funzione in sicurezza senza rivelare i propri dati privati durante l'esecuzione.

- *Secret sharing*: divide un segreto in n parti in modo che qualsiasi insieme di t parti consenta di ricostruire il segreto, mentre qualsiasi insieme di meno di t parti non rivela alcuna informazione su di esso. Molto diffuso è il *Shamir's Secret Sharing* scheme: esso permette di suddividere un'informazione in più parti (o "quote") e di ricostruire il segreto originale utilizzando solo un numero minimo di queste quote, noto come "soglia". Questo significa che, anziché tutte le quote, è sufficiente raggiungere il numero minimo stabilito per ricostruire il segreto. [5]

- *Oblivious transfer*: protocollo che consente la trasmissione sicura di più messaggi da parte di un mittente, il quale non sarà a conoscenza di quale dei tanti il ricevente ha ricevuto.

SMPC si adatta a diversi modelli di sicurezza per rispondere a varie minacce, ciascuna delle quali richiede livelli di protezione differenti:

- Avversari semi-onesti (honest-but-curious): è un avversario che corrompe alcune delle parti coinvolte, le quali seguono il protocollo in modo corretto ma cercano di ottenere quante più informazioni possibile dai messaggi ricevuti dalle altre parti. Un protocollo è sicuro contro avversari semi-onesti se le parti corrotte nel mondo reale hanno visioni indistinguibili rispetto a quelle che avrebbero nel mondo ideale.[2]

- Avversari malevoli (active): può causare deviazioni arbitrarie dal protocollo prescritto per cercare di comprometterne la sicurezza. Un avversario malevolo possiede tutte le capacità di un avversario semi-onesto nell'analisi dell'esecuzione del protocollo, ma ha anche la libertà di intraprendere qualsiasi azione desideri durante l'esecuzione del protocollo. [2]

2.3 Protocolli noti

Ogni protocollo che implementa Secure Multiparty Computation si basa su tecniche crittografiche uniche per garantire la privacy e l'integrità dei dati durante il calcolo. Seguono alcuni dei protocolli crittografici più rilevanti.

Yao's Garbled Circuits è una tecnica di calcolo sicuro in cui due parti calcolano una funzione su input privati senza rivelare i loro dati. P_1 codifica il circuito della funzione come una tabella cifrata e invia questa tabella a P_2 . Solo le chiavi corrette permettono a P_2 di decrittare la riga giusta per ottenere il risultato. Il protocollo garantisce sicurezza nel modello semi-onesto, dove le parti non apprendono informazioni oltre il necessario.

In **Goldreich-Micali-Wigderson (GMW) Protocol** i valori sono invece divisi in quote additive tra i partecipanti, il che permette un'estensione naturale a più di due parti, e permette ai partecipanti di sommare le loro quote per ottenere il valore totale senza rivelare gli input individuali. Il GMW è particolarmente adatto per il calcolo sicuro in scenari multiparty perché ogni partecipante mantiene solo una parte del valore. Questo approccio semplifica la gestione della privacy e consente di evitare la complessità associata alla condivisione completa dei dati, risultando efficace per calcoli distribuiti con più di due partecipanti.

Il protocollo **BGW**, introdotto da Ben-Or, Goldwasser e Wigderson, si basa sul Shamir's secret sharing anziché sui circuiti garbled. Questo protocollo permette di calcolare funzioni su un campo F rappresentando tali funzioni come circuiti aritmetici composti da operazioni di addizione, moltiplicazione e moltiplicazione per una costante. Per garantire la privacy, ogni valore di un wire è mantenuto segreto tramite un polinomio di grado t, condiviso tra le parti, evitando la rivelazione di valori intermedi durante l'elaborazione. A differenza degli altri due, esso è progettato per resistere anche ad avversari malevoli. [2]

2.3.1 Esempio del Yao's Millionaires' problem

L'idea fondamentale dietro SMPC può essere illustrata attraverso il classico "problema dei milionari di Yao": due milionari, Alice e Bob, vogliono sapere chi tra loro è più ricco senza rivelare il proprio patrimonio all'altro. Questo problema, apparentemente semplice, rappresenta perfettamente le sfide e gli obiettivi dell'SMPC: ottenere un risultato booleano del tipo $x_1 \leq x_2$, dove x_1 è l'input privato del primo milionario e x_2 l'input privato del secondo. [6] Questo problema "giocattolo" rappresenta perfettamente le sfide e gli obiettivi dell'SMPC.

Un esempio pratico di Yao's Millionaires' problem potrebbe essere:

```
1 from mpyc.runtime import mpc
2
3 async def main():
4     await mpc.start()
5     patrimonio = int(input("Inserisci il tuo patrimonio:"))
6     patrimonio_sicuro = mpc.input(mpc.SecInt(16)(patrimonio))
```

```
7     if await mpc.output(mpc.max(patrimonio_sicuro)) ==
          patrimonio:
8         print("Hai il patrimonio piu' alto")
9     else
10        print("Hai il patrimonio piu' basso")
11    await mpc.shutdown()
```

Codice 2.1: Yao's Millionaires' problem

Capitolo 3

Strumenti utilizzati

Per il lavoro di tesi sono stati utilizzati vari strumenti che hanno facilitato l'analisi della libreria e la simulazione del suo utilizzo in scenari distribuiti. Gli strumenti principali includono linguaggi di programmazione, tecnologie per la containerizzazione e strumenti di monitoraggio delle comunicazioni di rete.

3.1 Python

Python è stato il linguaggio di programmazione scelto per l'analisi effettuata durante questa tesi. Le sue caratteristiche principali quali la semplicità della sintassi e la leggibilità del codice hanno sicuramente semplificato la scrittura degli script per effettuare i test e le simulazioni; inoltre la sua ricca collezione di librerie, l'ampia documentazione e il supporto per il calcolo scientifico e distribuito lo hanno reso una scelta ideale.

3.2 Docker

Docker è una open platform per sviluppare, distribuire ed eseguire applicazioni. Essa consente di separare le applicazioni dall'infrastruttura e di distribuire rapidamente il software. [7] In questo lavoro, Docker è stato utilizzato per:

- configurare e creare container che simulano nodi distribuiti all'interno di una rete, anch'essa simulata;
- isolare le librerie e le dipendenze necessarie per ciascun nodo, evitando così i conflitti.

Questo approccio ha facilitato l'esecuzione di test distribuiti SMPC.

3.3 Wireshark

Wireshark è un analizzatore di pacchetti gratuito e open-source. È utilizzato per la risoluzione dei problemi di rete, l'analisi e lo sviluppo di software e protocolli di comunicazione. [8] Esso è stato utilizzato per analizzare il traffico di rete generato dai nodi durante l'esecuzione dei protocolli SMPC. Grazie alla sua capacità di catturare e analizzare pacchetti in tempo reale, è stato possibile monitorare le comunicazioni tra i nodi, identificare eventuali vulnerabilità nei protocolli di sicurezza e verificare l'effettiva privacy delle comunicazioni.

3.4 Altri strumenti

Oltre agli strumenti principali descritti sopra, sono stati utilizzati anche altri strumenti per facilitare il lavoro di tesi:

- Git: per la gestione del versionamento del codice e la collaborazione tra i vari ambienti di sviluppo.

- Visual Studio Code: come ambiente di sviluppo per scrivere e testare il codice Python.

Capitolo 4

Analisi della libreria

4.1 Cos'è MPyC?

MPyC è una libreria Python open-source progettata per effettuare Secure MultiParty Computation; essa supporta secure *m-computation*, tollerando un certo numero di parti passive disoneste fino ad un massimo di t, dove $m \geq 1$ e $0 \leq t < m/2$. I protocolli crittografici che utilizza si basano su *secret sharing* con soglia su campi finiti (Shamir's secret sharing) [9].

Essa permette dunque a più parti coinvolte di scambiarsi degli input e degli output e di effettuare calcoli distribuiti: per farlo utilizza TCP come protocollo di rete a livello di trasporto.

I nodi coinvolti in questo protocollo hanno tutti eguali privilegi tra loro; i calcoli e gli output prodotti devono essere effettuati e rivelati concordemente e contemporaneamente. Essi non sono però da considerarsi fidati e, anzi, hanno la possibilità di effettuare azioni semi-oneste o malevole volte a danneggiare i calcoli, gli output o far fallire il protocollo.

Gli assets che vanno protetti sono gli input degli utenti e gli output di calcolo, sia dal punto di vista della privacy che dell'integrità. Ad avere accesso

a queste informazioni devono essere solo e soltanto i nodi coinvolti nel calcolo, i quali non devono avere la possibilità di alterare in alcun modo gli input altrui o gli output di calcolo.

4.2 Criteri di valutazione

Per valutare la robustezza di questa libreria, sono stati definiti ed utilizzati i seguenti criteri:

- **Integrità della Privacy dell'Input**: verifica se i dati degli utenti rimangono riservati durante l'elaborazione, senza esporre involontariamente informazioni sensibili.
- **Resistenza agli Attacchi Semi-Onesti**: analisi della capacità della libreria di mantenere la privacy in presenza di avversari semi-onesti che seguono il protocollo ma cercano di estrarre informazioni non autorizzate.
- **Resistenza agli Attacchi Malevoli**: valutazione della protezione contro avversari che potrebbero deviare dal protocollo standard per compromettere la sicurezza.
- **Robustezza Crittografica**: valutazione dell'efficacia dei meccanismi crittografici utilizzati, come il secret sharing, per garantire la protezione degli input e degli output.

4.3 Modello d'attaccante

Prima di analizzare la libreria ed effettuare gli eventuali attacchi, è necessario definire il modello di attaccante.

Possiamo dunque definire attaccante come un nodo coinvolto nel protocollo che però ha degli interessi nel farlo deviare e far ottenere informazioni errate agli altri partecipanti oppure ottenere egli stesso delle informazioni private di altri partecipanti. Per farlo egli si può avvalere di strumenti terzi o utilizzare funzioni di libreria stesse per deviare le comunicazioni in atto e deviare così dal corretto utilizzo del protocollo.

Gli attaccanti si possono, nello specifico, dividere in due categorie:

- **attaccante semi-onesto** (o curioso): egli non effettua azioni non previste dal protocollo ma cerca di trarre quante più informazioni possibili riguardo gli input altrui.
- **attaccante malevolo**: egli effettua azioni al di fuori del protocollo con uno scopo malevolo che va oltre lo scoprire informazioni riservate.

4.4 Attacchi e analisi

L'analisi è iniziata con un codice basilare che prevedeva lo scambio di dati tra due nodi:

```
1 from mpyc.runtime import mpc
2
3 async def main():
4     await mpc.start()
5     a = mpc.input(mpc.SecInt(16)(2))
6     print("a = ", await mpc.output(a))
7     await mpc.shutdown()
```

Codice 4.1: Codice basilare

Qui si possono distinguere 4 fasi:

- Fase 1 - *mpc.start()*: viene avviata la comunicazione tra le varie parti;
- Fase 2 - *mpc.input()*: gli input vengono condivisi tramite la tecnica del Secret Sharing;
- Fase 3 - *mpc.output()*: concordemente tra tutte le parti, vengono stampati tutti gli input condivisi;
- Fase 4 - *mpc.shoutdown()*: la comunicazione tra le parti viene interrotta.

Questo codice d'esempio rappresenta la base per testare la robustezza di MPyC. A seguire, verranno analizzati alcuni comportamenti non previsti nel protocollo, che potrebbero essere sfruttati per compromettere la correttezza o la sicurezza dei calcoli condivisi. Questi comportamenti, pur non rappresentando necessariamente attacchi malevoli veri e propri, mettono in luce potenziali debolezze nella gestione degli input condivisi e delle comunicazioni tra i nodi.

Per ciascun caso, verranno presentati esempi pratici basati sulla rete Docker utilizzata per i test, accompagnati da una spiegazione dei meccanismi che rendono possibile il comportamento non previsto e da una valutazione delle sue conseguenze sul protocollo complessivo.

4.4.1 Attacco 1: Input rivelati

Il primo approccio utilizzato è stato quello di cercare di eludere il protocollo per come è stato strutturato: l'attacco presentato sfrutta il comportamento di un nodo semi-onesto, il quale esegue operazioni aggiuntive che non rispettano il protocollo concordato, riuscendo ad ottenere informazioni private. Secondo il modello STRIDE, un processo metodologico che aiuta ad individuare le minacce di sicurezza in un sistema complesso, questo tipo di attacco può essere categorizzato in Information Disclosure, ovvero l'esposizione delle informazioni a persone non autorizzate alla loro visione. Per completezza, le altre categorie di minacce del modello STRIDE sono: Spoofing, Tampering, Repudiation, Denial of service e Elevation of privilege. [10]

In particolare, per quanto riguarda questo primo attacco, prendono parte due nodi, che hanno i seguenti comportamenti:

- **Nodo onesto:** Esegue il protocollo previsto, calcolando e comunicando solo il valore massimo degli input condivisi `mpc.max`.

```
1 async def main():
2     await mpc.start()
3     a = mpc.input(mpc.SecInt(16)(2))
4     print("massimo = ", await mpc.output(mpc.max(a)))
5     await mpc.shutdown()
```

Codice 4.2: Codice parte onesta

- **Nodo semi-onesto:** Oltre al calcolo del massimo, utilizza un'azione aggiuntiva per estrarre e visualizzare tutti i valori degli input condivisi `a`, sfruttando il comando `mpc.output()`.

```
1 async def main():
2     await mpc.start()
3     a = mpc.input(mpc.SecInt(16)(4))
4     print("massimo (malevolo= ", await mpc.output(mpc.max
          (a)))
5     print("a (malevolo)= ", await mpc.output(a)) #azione
          aggiuntiva
6     await mpc.shutdown()
```

Codice 4.3: Codice avversario semi-onesto

I log di output mostrano come il nodo malevolo riesca a recuperare gli input condivisi, violando le proprietà di privacy del protocollo SMPC. Si noti in particolare la *riga 6* dei log.

```
1 docker-mpyc-node-malicious-1  | 2024-11-23 11:50:48,795 Start MPyC
    ↪ runtime v0.10
2 docker-mpyc-node-1-1          | 2024-11-23 11:50:48,801 Start MPyC
    ↪ runtime v0.10
3 docker-mpyc-node-1-1          | 2024-11-23 11:50:48,801 All 2 parties
    ↪ connected.
4 docker-mpyc-node-malicious-1  | 2024-11-23 11:50:48,801 All 2 parties
    ↪ connected.
5 docker-mpyc-node-malicious-1  | massimo (malicious)=  4
6 docker-mpyc-node-malicious-1  | a (malicious)=  [2, 4]
7 docker-mpyc-node-1-1          | massimo =  4
8 docker-mpyc-node-1-1          | 2024-11-23 11:50:48,804 Stop MPyC --
    ↪ elapsed time: 0:00:00.002|bytes sent: 18
9 docker-mpyc-node-malicious-1  | 2024-11-23 11:50:48,804 Stop MPyC --
    ↪ elapsed time: 0:00:00.002|bytes sent: 18
```

Codice 4.4: Log di output Attacco 1

L'attacco ha dimostrato vulnerabilità per $m = 2$ e $t = 1$, ma non per $m > 2$, dove la libreria resiste meglio.

4.4.2 Attacco 2: Somma manipolata

L'idea dietro questo secondo attacco era quella di estendere l'attacco 1 a $m > 2$ parti. Nonostante ciò si è poi rivelato infattibile nella pratica, è emersa una nuova falla nel protocollo: un nodo malevolo può alterare la somma condivisa, causando errori senza che gli altri nodi si accorgano del problema. Esempio:

- Un nodo malevolo effettua un'operazione aggiuntiva;
- Un nodo onesto riceve un risultato falsato a causa di questa azione.

In questo caso, i nodi onesti inviano un input e chiedono indietro, concordemente, prima un output dei loro input e poi la loro somma. Il nodo malevolo, invece, chiede l'output degli input di tutti due volte, nella *riga 7*:

```
1 from mpyc.runtime import mpc
2
3 async def main():
4     await mpc.start()
5     a = mpc.input(mpc.SecInt(16)(2))
6     print("a = ", await mpc.output((a)))
7     print("a = ", await mpc.output((a)))
8     print("sum a = ", await mpc.output(mpc.sum(a)))
9     await mpc.shutdown()
```

Codice 4.5: Codice avversario malevolo

Il risultato è che uno degli altri nodi onesti riceverà una somma totale falsata, prendendola come onesta.

```
1 mpyc-node-1-1          | 2024-11-26 12:36:07,467 All 5 parties
    ↪ connected.
2 mpyc-node2-1           | 2024-11-26 12:36:07,475 All 5 parties
    ↪ connected.
3 mpyc-node3-1           | 2024-11-26 12:36:07,527 All 5 parties
    ↪ connected.
4 mpyc-node-2-1          | 2024-11-26 12:36:07,882 All 5 parties
    ↪ connected.
5 mpyc-node-malicious-1  | 2024-11-26 12:36:07,882 All 5 parties
    ↪ connected.
6 mpyc-node2-1           | a =  [2, 2, 2, 4, 2]
7 mpyc-node2-1           | sum a =  12
8 mpyc-node-malicious-1  | a =  [2, 2, 2, 4, 2]
9 mpyc-node-1-1          | a =  [2, 2, 2, 4, 2]
10 mpyc-node-2-1         | a =  [2, 2, 2, 4, 2]
11 mpyc-node2-1          | 2024-11-26 12:36:07,885 Stop MPyC -- elapsed
    ↪ time: 0:00:00.409|bytes sent: 192
12 mpyc-node3-1          | a =  [2, 2, 2, 4, 2]
13 mpyc-node-malicious-1 | a =  [-140236197026730]
14 mpyc-node-2-1         | sum a =  138231031712540
15 mpyc-node3-1          | sum a =  12
16 mpyc-node-malicious-1 | sum a =  44449267129935
```

Codice 4.6: Log di output Attacco 2

Come previsto, il nodo *mpyc-node-2-1* riceve un numero totalmente diverso da quello degli altri nodi onesti, com'è possibile notare dalla *riga 14*.

Se volessimo categorizzare questo attacco appena descritto, potrebbe rien-

trare tra le minacce di Tampering, ovvero l'alterazione di dati condivisi.

4.4.3 Attacco 3: Interruzione della catena di output

Per quest'ultimo attacco, simulato con m=3, è stato necessario analizzare con Wireshark i pacchetti che i vari nodi si scambiano tra loro quando eseguono il protocollo. Il codice per il quale sono stati analizzati i pacchetti è il seguente:

```
1 async def main():
2     await mpc.start()
3     a = mpc.input(mpc.SecInt(16)(2))
4     print("a = ", await mpc.output((a)))
5     await mpc.shutdown()
```

Codice 4.7: Codice in analisi

I pacchetti sono risultati estremamente facili da distinguere tra loro: il loro contenuto sembra essere semplicemente convertito in esadecimale e risulta essere di facile interpretazione. In particolare è emerso che:

- Durante `mpc.start()` i nodi si scambiano dei dati di inizio comunicazione, sembrerebbe in maniera randomica.
- Durante `mpc.input()` ogni nodo manda uno dei suoi segreti agli altri due, tenendosi il terzo per se. Questa operazione è facilmente riconoscibile poiché ogni input segreto inviato è preceduto sempre dalla stessa stringa, "*189baad3c99353b806000000*".
- Durante `mpc.output()` ogni nodo manda agli altri dei dati, anch'essi riconoscibili poiché iniziano tutti con *ecedcb8ccc55064112000000*; la restante parte dei dati risulta essere la concatenazione dei segreti dei nodi

1, 2 e 3, già scoperti dai pacchetti precedenti. L'invio avviene secondo il seguente schema:

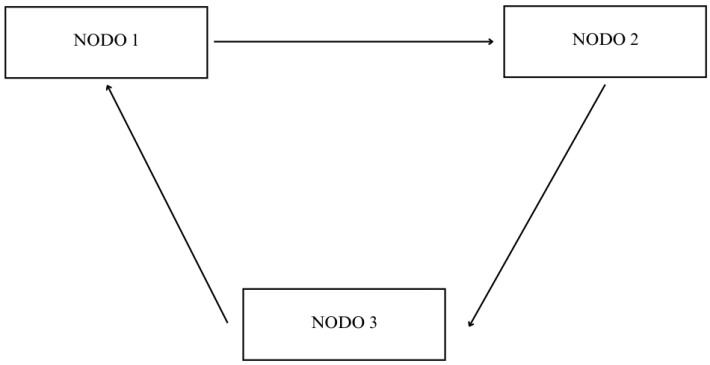

Figura 4.1: Schema invio pacchetti per `mpc.output()`

- Durante `mpc.shutdown()` ogni nodo invia agli altri dei dati per comunicare la fine della connessione.

Da questa analisi è sorta l'idea di cercare di interrompere la catena di comunicazioni tra i nodi, in particolar modo quella durante l'esecuzione di *mpc.ouptut*. Per farlo, è stata riscritta una funzione, *_send_message*, utilizzata nel codice della libreria durante questa fase e alterata affinché inviasse una stringa di soli zero, per alterare l'output del nodi successivi.

```
1 async def main():
2     await mpc.start()
3     a = mpc.input(mpc.SecInt(32)(2))
4 
5     def _send_message( peer_pid, share):
6         if peer_pid == 0:
7             share = b"0000000000000000"
8 
9     print("a = ", _send_message(peer_pid=0, share=a))
```

```
10      await mpc.shutdown()
```

Codice 4.8: Codice in analisi

Una volta eseguito questo attacco, è possibile notare come gli output dei nodi non malevoli risultino essere entrambi falsati. È possibile notare ciò in questi log di output alle righe 6 ed 8.

```
1 docker-mpyc-node-1-1          | 2024-11-27 07:33:21,691 All 3 parties
    ↪ connected.
2 docker-mpyc-node-malicious-1  | 2024-11-27 07:33:21,691 All 3 parties
    ↪ connected.
3 docker-mpyc-node-2-1          | 2024-11-27 07:33:21,691 All 3 parties
    ↪ connected.
4 docker-mpyc-node-malicious-1  | a =  None
5 docker-mpyc-node-malicious-1  | 2024-11-27 07:33:21,695 Stop MPyC --
    ↪ elapsed time: 0:00:00.003|bytes sent: 40
6 docker-mpyc-node-1-1          | a =  [2, -60, 2]
7 docker-mpyc-node-1-1          | 2024-11-27 07:33:21,696 Stop MPyC --
    ↪ elapsed time: 0:00:00.004|bytes sent: 66
8 docker-mpyc-node-2-1          | a =  [20533595276216]
9 docker-mpyc-node-2-1          | 2024-11-27 07:33:21,696 Stop MPyC --
    ↪ elapsed time: 0:00:00.004|bytes sent: 66
```

Codice 4.9: Log di output Attacco 3

4.5 Valutazione del lavoro

Gli attacchi descritti evidenziano che, sebbene MPyC garantisca una robustezza significativa contro avversari semi-onesti e malevoli nelle configurazioni standard, emergono vulnerabilità specifiche in presenza di input manipolati o di comunicazioni non correttamente protette. Questi risultati sottolineano l'importanza di condurre test approfonditi e sistematici e di implementare misure di mitigazione per garantire che i sistemi SMPC rimangano effettivamente privacy-preserving anche in scenari non convenzionali.

Conclusione

Il lavoro presentato in questa tesi si è focalizzato sull'analisi della libreria MPyC, la quale implementa protocolli di Secure MultiParty Computation per garantire privacy e sicurezza in calcoli distribuiti. L'obiettivo principale è stato verificare l'effettiva robustezza della libreria in scenari realistici, valutando la sua capacità di resistere a comportamenti semi-onesti o malevoli da parte dei nodi partecipanti.

Durante lo studio sono stati realizzati diversi attacchi per analizzare il comportamento della libreria; questi test hanno evidenziato alcune vulnerabilità nelle implementazioni attuali, in particolare: la possibilità di rivelare informazioni private attraverso operazioni non previste nel protocollo, la manipolazione dei dati condivisi tramite nodi malevoli, la facilità di decryption dei dati trasmessi tra i nodi e la difficoltà nell'identificazione di input o comunicazioni errate. Questi risultati sottolineano l'importanza di integrare ulteriori contromisure, come l'uso di zero-knowledge proofs o la definizione di protocolli con verifiche crittografiche più rigorose, per migliorare la resilienza contro avversari attivi.

Le potenzialità di MPyC e delle tecnologie SMPC sono evidenti, ma ci sono ancora molte aree che meriterebbero ulteriori approfondimenti. Tra le possibili direzioni future si possono includere:

- Irrobustimento della crittografia dei dati in transito, aggiungendo dei

meccanismi di crittografia asimmetrica, al fine di garantire un ulteriore livello di sicurezza contro attacchi man-in-the-middle o intercettazioni non autorizzate durante la comunicazione tra i nodi;

- Integrazione con servizi e applicativi che possano sopperire alle vulnerabilità di questa libreria, implementando funzionalità aggiuntive, come sistemi di auditing crittografico o protocolli per la gestione degli attacchi e delle deviazioni dal protocollo standard.

Concludendo, la libreria MPyC rappresenta un passo significativo verso l'adozione di calcoli distribuiti privacy-preserving, ma il lavoro svolto ha evidenziato la necessità di ulteriori sviluppi per garantire piena sicurezza e affidabilità in scenari critici. Questo studio costituisce un contributo preliminare per rafforzare le basi crittografiche di tali sistemi e stimolare ulteriori ricerche nel campo.

Bibliografia

[1] Alfred J. *Handbook of applied cryptography.* Boca Raton : CRC Press, 1997.

[2] Vladimir Kolesnikov David Evans and Mike Rosulek. *A Pragmatic Introduction to Secure Multi-Party Computation.* 2018.

[3] What is secure multiparty computation? https://inpher.io/technology/what-is-secure-multiparty-computation/.

[4] https://eprint.iacr.org/2020/300.pdfhttps://eprint.iacr.org/2020/300.pdf/.

[5] Shamir's secret sharing. https://medium.com/@keylesstech/a-beginners-guide-to-shamir-s-secret-sharing-e864efbf3648.

[6] Yao's millionaires' problem. https://en.wikipedia.org/wiki/Yao%27s_Millionaires%27_problem.

[7] Docker. https://docs.docker.com/get-started/docker-overview/.

[8] Wireshark. https://en.wikipedia.org/wiki/Wireshark).

[9] MPyC Multiparty Computation in Python. https://github.com/lschoe/mpyc?tab=readme-ov-file.

[10] Linee guida per la modellazione delle minacce ed individuazione delle azioni di mitigazione conformi ai principi del secure/privacy by design. https://www.agid.gov.it/sites/default/files/repository_files/documentazione/linee_guida_modellazione_minacce_e_individuazione_azioni_di_mitigazionev1.0.pdf.

Ringraziamenti

Se mi guardo indietro, non riesco a non pensare che questi tre anni di percorso siano volati: arrivare qui significa per me aver attraversato tutti i momenti difficili e le sfide che conseguire una laurea può portare, averli superati e dimenticati. Solo gioia e gratificazione riempiono il mio cuore adesso, e non potrei essere più felice di così.

Questo percorso ha significato tanto per me: mi ha fatto capire cosa voglio, cosa mi piace fare, in cosa sono brava, mi ha fatto ritrovare la passione di studiare che tanto mi era passata prima di intraprendere questo percorso.

Ma se sono arrivata fin qui non è solo grazie a me stessa, ma anche alle persone che sempre mi hanno sostenuta più di tutti: i miei genitori. Siete stati fondamentali in questo percorso, non solo per il supporto *materiale*, ma anche per quello emotivo: non mi è mai mancata una parola di supporto e di comprensione da parte vostra; siete stati presenti e coinvolti nel mio percorso senza essere oppressivi, lasciandomi la mia giusta libertà e avendo fiducia in me: se sono qui, sono sicura che pensate che quella fiducia sia stata ben riposta. Vi voglio bene e vi sono grata per tutto ciò che ogni giorno fate per me.

Però la mia famiglia non finisce qui, ad essere sempre stata presente c'è anche la mia piccola sorellina Fabiana. Con i tuoi commenti acidi, le tue parole dolci e avendomi sempre ascoltata mi hai sempre portato un sorriso e

conforto quando ne avevo bisogno. Ti voglio e ti vorrò sempre bene, e ci sarò sempre quando ne avrai bisogno, e anche quando non ne avrai.

Voglio ringraziare anche tutta la mia famiglia: mia nonna Rosa, i miei nonni Salvatore e Giuseppina, mio zio Gaetano e mia zia Loredana. So quanto ci avete tenuto a questo mio percorso e la vostra presenza, anche solo con una chiamata, si è fatta sempre sentire e ha scaldato il cuore.

Un grazie particolare ai miei (*unici*) cugini, Giulia e Andrea: nonostante siamo sempre lontani, nonostante non ci vediamo spesso, so quanto siamo legati, quanto bene ci vogliamo e che questo non cambierà mai a prescindere da dove ci troveremo, e voglio che lo sappiate anche voi. Grazie Giulia, per tutto quello che hai sempre fatto per me, in te oltre che una cugina ho sempre trovato un'amica speciale. Vi voglio bene, cugini per sempre.

Alla mia seconda famiglia, a Natascia, Zaira, Fadua, Jenny, Alessia, Flavio e Marta: sapete che non sono brava con le parole, ma comunque non basterebbero a descrivere ciò che io provo per voi e ciò che siamo. Siete stati sempre tutto per me, da quando mi avete accolta tra le vostre braccia ho trovato il mio luogo sicuro in cui rifugiarmi sempre, essere me stessa senza mai esse giudicata, mi avete fatta sentire davvero a casa. Sono grata per tutti gli anni trascorsi insieme, di aver potuto condividere gioia e dolore con voi e di essere ancora così legati. Sapete quanto ognuno di voi, nei vostri speciali ed unici modi di essere, significhi per me. Vi voglio bene e anche di più.

A Federico, che in questi tre, e più, anni mi è sempre stato accanto nei miei momenti peggiori, mi ha vista crescere, cadere e rialzarmi in questo percorso e non ha mai esitato nell'aiutarmi. A te che, nonostante tutto, avrai sempre un posto speciale nel mio cuore: grazie per tutto.

Agli "Amici del DMI", a tutti i miei colleghi e amici che ho trovato in questo posto e che mi hanno allietato tra una lezione e l'altra, durante i pranzi alla mensa, i caffè presi alle macchinette, durante lo studio intenso e le pause di cinque minuti che diventavano ore a ridere e scherzare insieme. Grazie a Baffo, Miriana, Luca, Gaia, Lorenzo, Giulio, Alessandro, Andrea: avete reso questo luogo una seconda casa per me.

E, Fabrizio, anche tu hai fatto tutto questo per me, ma non solo. Non sei stato solo un collega o un amico fidato per me, non mi hai solo allietato ogni singolo giorno passato qui dentro e fuori da qui, ma hai fatto molto altro. Sei per me una di quelle persone speciali che la vita ti fa conoscere quando vuole farti un regalo. Non so se meritavo questo regalo, ma sono contenta di averti con me. Sono felice della persona che sei e spero tu possa sempre emanare questa luce che ti contraddistingue: non farla spegnere mai. Grazie per tutto ciò che hai fatto e continui a fare per me.