UNIVERSITÀ POLITECNICA DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica

TESI DI LAUREA MAGISTRALE

Prove a conoscenza zero per la scalabilità e la privacy in una filiera

Zero-knowledge proofs for scalability and privacy in a supply-chain

Relatore:
Prof. Luca Spalazzi

Candidato:
Sebastian George Moian

Correlatore:
Prof. Massimiliano Pirani

Anno Accademico 2023-2024

Abstract

In this thesis, the aim is to understand the possibilities and limitations of Zero Knowledge Proofs (ZKP) in blockchain technology. The first chapters focus on the presentation of distributed ledger systems and the tools that will be used, with particular reference to Remix, the IDE of choice for programming smart contracts in Solidity, and Sepolia, the Ethereum Testnet where the smart contracts were deployed.
Subsequently, zero knowledge proofs are defined, their characteristics are explained, and the difference between interactive and non-interactive proofs is discussed; the latter will be emphasized as they are the most interesting in this context.
The thesis then moves on to possible theoretical application contexts, with a particular focus on verifiable computation.
Zokrates, a framework used to implement the non-interactive ZKP protocol, will be presented, explaining its language, features, functioning, and differences compared to other programming languages. A focal point for explaining how to implement verifiable computation will be introducing the various representations of an arithmetic circuit, fundamentally of textual and matrix types.
Through various tests, the limitations of Zokrates and, to some extent, the entire arithmetic circuits sector for ZKP were highlighted. The thesis concludes with a possible implementation on the subject and an attack that this technology may be able to avoid.

Sommario

In questa tesi si pone come obiettivo comprendere possibilità e limiti delle Prove a Conoscenza Zero (ZKP) nella tecnologia blockchain.
Le prime sezioni si concentrano sulla presentazione dei sistemi a registro distribuito e degli strumenti che andranno utilizzati, facendo riferimento in particolar modo a Remix, IDE di riferimento per la programmazione di smart contracts in solidity, e Sepolia la Testnet Ethereum su cui è stato eseguito il deploy degli smart contracts.
In seguito vengono definite le prove a conoscenza zero, le caratteristiche di questi oggetti e la differenza tra prove interattive e non interattive; ci si soffermerà su queste ultime dato che sono le più interessanti in questo contesto.
Per poi passare ai possibili contesti di applicazione teorici, con un particolare riguardo al calcolo verificabile.
Verrà presentato Zokrates, framework utilizzato per implementare il protocollo ZKP non interattivo spiegandone il linguaggio, le caratteristiche, il funzionamento e le differenze rispetto ad altri linguaggi di programmazione.
Punto focale per spiegare come implementare il calcolo verificabile sarà introdurre le varie rappresentazioni di un circuito aritmetico, fondamentalmente di tipo testuale e matriciale.
Attraverso diverse prove si sono poi evidenziati quelli che sono i limiti di Zokrates e in parte di tutto il reparto dei circuiti aritmetici per ZKP per poi finire con una possibile implementazione al tema e un attacco che questa tecnologia può riuscire evitare.

Indice

1 Introduzione 1

2 Stato dell'arte 2
- 2.1 Bitcoin 2
- 2.2 Ethereum 4
- 2.3 Smart contracts 5
- 2.4 Solidity 7
- 2.5 Remix 10
- 2.6 Brownie 11
- 2.7 Ganache 11
- 2.8 Sepolia Testnet 12

3 Zero-knowledge proofs 14
- 3.1 Caratteristiche principali di una Zero-Knowledge Proof 14
- 3.2 interactive zero-knowledge proofs 15
- 3.3 Non-interactive zero-knowledge proofs 16
- 3.4 Tipologie di ZKP non interattive 17
  - 3.4.1 zkSNARK 17
  - 3.4.2 zkSTARK 18
  - 3.4.3 Bulletproofs 19
  - 3.4.4 zk-EVM (Zero-Knowledge Ethereum Virtual Machine) 19
- 3.5 Applicazioni teoriche 20
  - 3.5.1 Transazioni anonime 20
  - 3.5.2 Identità decentralizzata e autenticazione 21
  - 3.5.3 Calcolo Verificabile 22

4 ZoKrates 24
- 4.1 Introduzione 24
- 4.2 Linguaggio di programmazione 25
- 4.3 Workflow di Zokrates 26
- 4.4 Rappresentazione intermedia di Zokrates 27
  - 4.4.1 Circuiti aritmetici 27
  - 4.4.2 SLP 29
  - 4.4.3 Rank-1 Costraint Systems 30
- 4.5 Zokrates verifiable computation 33

5 Applicazione delle Zero-Knowledge Proofs alle filiere produttive . . 34
5.1 Introduzione 34
5.2 La filiera e i partecipanti 35
5.3 Applicazione delle ZKP nel contesto della filiera 37
5.4 Verifier 40
5.4.1 Script Verifier.sol 45
5.5 Multi-Verifier 54
5.5.1 Script Proxy.sol 56
5.6 Mapping stringhe di caratteri 58
5.7 Possibile attacco al protocollo 59

6 Limiti di Zokrates 61
6.1 Zokrates Test 61
6.2 Considerazioni sui test 62

7 Conclusioni 64

Elenco delle figure

1 Struttura base di Bitcoin 2
2 Diagramma semplificato funzionamento Smart contract 6
3 Altre Unità di ether 9
4 Layout Remix-IDE 10
5 Ganache UI 11
6 Deploy Smart contract 12
7 Smart contract non verificato 13
8 Smart contract verificato 13
9 Esempio funzionamento IZKP 15
10 Meccanismo del Non-interactive zero-knowledge proof 16
11 Rappresentazione circuito aritmetico 27
12 Vettori a,b,c 30
13 Singolo gate di un circuito aritmetico 31
14 Esempio circuito aritmetico 32
15 Architettura della filiera con tracciamento su blockchain . . . 36
16 Applicazione ZKP filiera 39
17 Computazione eseguita correttamente 41
18 Smart Contract contenente 2 verificatori 54
19 Multi_Verifier+Proxy 55
20 Mapping dei caratteri 58
21 Versione del sistema che usa un hash come commitment 59

1 Introduzione

In un contesto economico globalizzato, la gestione della filiera produttiva e la tracciabilità dei prodotti sono diventati aspetti cruciali per le aziende che desiderano garantire qualità e trasparenza ai propri clienti.
La tracciabilità di filiera, si riferisce alla capacità di tracciare ogni fase del processo produttivo, dalla materia prima al prodotto finito, garantendo così conformità normativa e sicurezza.
Ogni fase della filiera è interconnessa e dipende da tutte le aziende coinvolte nel processo, una gestione efficiente di queste relazioni è fondamentale per ottimizzare i costi e migliorare la qualità del prodotto. In questo ambito alcune tecnologie studiate sono[1]:

- **Sistemi ERP (Enterprise Resource Planning)**: Software che integra tutte le fasi produttive e amministrative, facilitando la raccolta di dati e la gestione delle informazioni in tempo reale.
- **RFID (Radio Frequency Identification)**: Tag che possono essere applicati ai prodotti per monitorare i loro spostamenti lungo la filiera, dalla produzione alla vendita.
- **Blockchain**: Registro immutabile e decentralizzato delle transazioni, garantisce trasparenza e sicurezza delle informazioni.

In questo documento, verrà esplorato lo stato dell'arte, le basi teoriche e come queste soluzioni possono essere implementate per affrontare i problemi di scalabilità e privacy nei sistemi basati su Blockchain, fornendo nelle sezioni finali esempi concrecti. Verranno analizzati inoltre limiti attuali di tali tecnologie.

2 Stato dell'arte

2.1 Bitcoin

Bitcoin, è il primo sistema basato su blockchain, proposto nel 2008 da Satoshi Nakamoto. Il suo obiettivo era creare una valuta digitale decentralizzata che non potesse essere controllata da un'autorità centrale e che permettesse il trasferimento di valore peer-to-peer senza fare affidamento su un intermediario fidato.
A tal fine, il protocollo Bitcoin combina transazioni protette da crittografia asimmetrica con un algoritmo di consenso per decidere l'ordine delle transazioni all'interno di una rete.
I partecipanti convalidano le transazioni in modo indipendente, eliminando la necessità di una parte fidata.
Il protocollo di consenso Proof of Work (PoW) rappresenta il pilastro su cui si basa il funzionamento del sistema Bitcoin. Attraverso questo protocollo, la possibilità di aggiungere nuove transazioni alla rete viene conferito in proporzione allo sforzo computazionale investito da un partecipante nella generazione di nuovi blocchi. per motivi di efficienza, le transazioni vengono aggregate in blocchi. Questi sono poi aggiunti attraverso il consenso, e ciascuno di essi fa riferimento al proprio predecessore, creando una struttura a catena immutabile e permanente, la blockchain. [2]

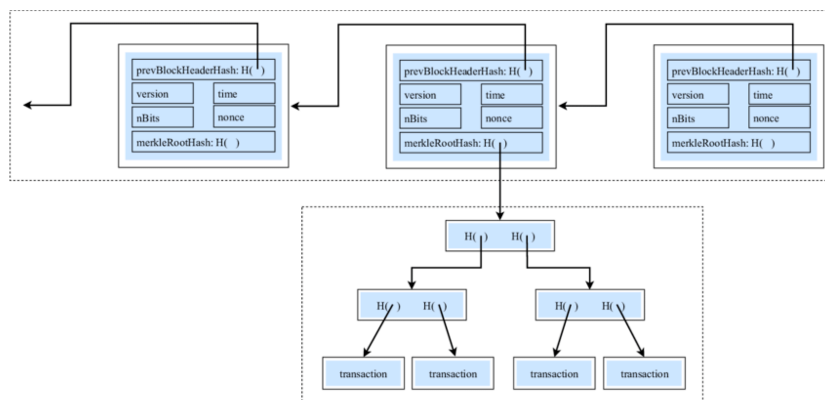

Figura 1: Struttura base di Bitcoin

All'interno della sua architettura, Bitcoin impiega un linguaggio di scripting basato su stack, semplice e limitato, per definire la logica di validazione

delle transazioni. Si noti come Bitcoin Script non è Turing-completa per assicurare che tutte le esecuzioni degli script terminino, prevenendo così il rischio che la rete possa subire rallentamenti o blocchi dovuti a loop infiniti. Il linguaggio permette agli sviluppatori di specificare le condizioni sotto le quali i bitcoin possono essere spesi, ma limita la possibilità di sviluppare altre applicazioni su blockchain, questa è la prima grande differenza rispetto a Ethereum.

2.2 Ethereum

Ci si rese conto da Bitcoin che le blockchain potessero essere utilizzate anche per altri scopi oltre lo scambio di denaro. nel 2015 Vitalik Buterin e altri cofondatori decisero di creare Ethereum, una nuova blockchain ideata in modo da poter essere più versatile e dove gli sviluppatori potessero crearci sopra delle applicazioni.
L'innovazione chiave di Ethereum è l'introduzione della Ethereum Virtual Machine (EVM), un ambiente di esecuzione che opera come una macchina virtuale decentralizzata per eseguire contratti intelligenti, questo permette di eseguire applicazioni sulla rete globale, non appogiandosi a sistemi centralizzati.
Come spiegato nella sezione precedente, ethereum si basa su un approccio Turing-completa, questo significa che l'EVM permette di eseguire script con una gamma molto più ampia di funzionalità computazionali, rendendo possibile lo sviluppo di applicazioni decentralizzate (dApps).
La nascita di Ethereum ha mostrato al mondo le numerose possibilità dei registri distribuiti e reso la blockchain una piattaforma non solo per la gestione immutabile dei dati ma anche per eseguire calcoli decentralizzati.

2.3 Smart contracts

Gli smart contracts sono programmi salvati sulla blockchain che si attivano quando condizioni predefinite dallo sviluppatore si verificano, i nodi della rete eseguono la stessa computazione per poi comparare i risultati, se viene trovato un consenso il risultato della computazione è salvato all'interno della blockchain.
La caratteristica che ha posto i riflettori su questa tecnologia è che questi contratti digitali, eseguono una serie di operazioni senza la necessità di intermediari e autorità centralizzate.
Per funzionare e assicurare affidabilità e coerenza, gli smart contract devono essere deterministici e garantire la terminazione. Questo evita discrepanze nei risultati tra i nodi della rete, essenziale per il raggiungimento di un consenso.
Operazioni che potrebbero portare a risultati non uniformi, come l'accesso al filesystem o alla rete, sono proibite. Inoltre, la natura ridondante della loro esecuzione su ogni nodo impone che non ci siano operazioni che possano bloccare la rete.
Per prevenire problemi come loop infiniti, Ethereum utilizza un sistema di "gas", che limita le risorse durante l'esecuzione di uno smart contract. Ogni operazione consuma una quantità di gas e l'esaurimento del gas assegnato determina la fine dell'esecuzione.
Infatti per non incorrere a una computazione parziale al momento della richiesta di esecuzione viene calcolato il gas usato e il max. Questo sistema garantisce che anche codice con potenziali loop infiniti terminino, preservando la funzionalità della rete.

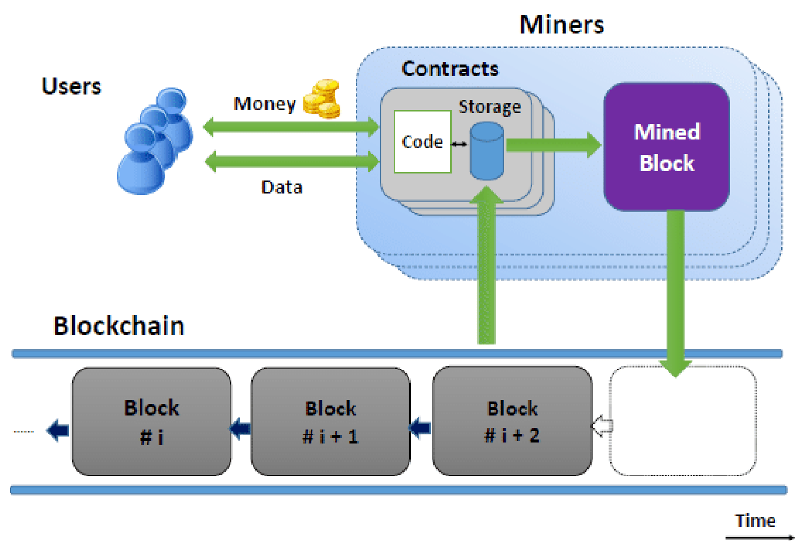

Figura 2: Diagramma semplificato funzionamento Smart contract

2.4 Solidity

Solidity è un linguaggio di programmazione ad alto livello, object-oriented, sviluppato specificamente per la creazione e l'implementazione di smart contracts su blockchain basate si EVM, la macchina virtuale di Ethereum. Il linguaggio è stato progettato in modo da essere simile a JavaScript e essere accessibile a un ampio numero di sviluppatori.
Facendo una breve panoramica, Solidity è costituito da 2 gruppi di tipologie di dati:

Value types

Le variabili Value sono sempre passati per valore. ciò significa che quando vengono passate come argomento a una funzione, Solidity non passa il riferimento all'originale, ma crea una nuova istanza della variabile contenente gli stessi dati. Questo implica che modifiche fatte alla variabile all'interno della funzione non influenzano l'originale fuori dalla funzione.
I tipi di dati di tipo valore in Solidity sono elencati di seguito:

- Integer
- Bool
- Byte
- Address
- Enum

Reference types

Le variabili di tipo riferimento memorizzano la posizione dei dati. Non condividono i dati direttamente. Questo significa che due diverse variabili possono fare riferimento alla stessa posizione e una qualsiasi modifica in una variabile può influenzare l'altra. I Reference types in Solidity sono elencati di seguito:

- Array
- String
- Struct
- Mapping

Control Flow

I loop sono utilizzati quando è necessario implementare dei cicli nel programma, rispecchia le stesse caratteristiche degli altri linguaggi ed è possibile utilizzare:

- While
- Do-While
- For

Units in Solidity

Un "unità",nella blockchain, si riferisce a una "denominazione". Le unità di Ether utilizzate per pagare i processi computazionali all'interno dell'EVM. Nella programmazione Solidity, un unità è una misura di valore o tempo utilizzata nel codice. Esistono 2 tipologie di Unit:

- **Ether** Le unità di Ether sono utilizzate per rappresentare il valore, come ad esempio la quantità di denaro trasferita tra account o il costo di una transazione.[3]

Unit	Wei Value	Wei
wei	1 wei	1
Kwei (babbage)	1e3 wei	1,000
Mwei (lovelace)	1e6 wei	1,000,000
Gwei (shannon)	1e9 wei	1,000,000,000
microether (szabo)	1e12 wei	1,000,000,000,000
milliether (finney)	1e15 wei	1,000,000,000,000,000
ether	1e18 wei	1,000,000,000,000,000,000

Figura 3: Altre Unità di ether

- **Unità di Tempo** Le unità di tempo, sono invece utilizzate per misurare la durata di determinati eventi nella blockchain, come ad esempio il tempo che deve trascorrere prima che una determinata azione sia eseguita.
  - **seconds** (s)
  - **minutes** (min)
  - **hours** (h)
  - **days** (days)
  - **weeks** (weeks)

2.5 Remix

Remix IDE è un ambiente di sviluppo basato su browser ideato per facilitare la scrittura, il testing e il deployment di smart contracts su blockchain che utilizzano EVM o simili.
È uno strumento a cui è semplice accedere essendo disponibile online senza necessità di installazioni locali, questa caratteristica permette di lavorare a un progetto anche tra sistemi operativi differenti. I linguaggi di programmazione supportati sono Solidity e Vyper, un altro linguaggio stile python.
Uno degli aspetti più potenti di Remix è la sua capacità di eseguire il deploy di contratti in modo veloce su blockchain, sia testnet che reti private. Inoltre, è possibile estendere le funzionalità dell'IDE attraverso una varietà di plugin, come Zokrates, toolbox fondamentale che verrà usato per costruire prove zkSNARK.
L'ambiente di Remix permette anche di utilizzare reti di prova come Ganache, permettendo di testare anche contratti computazionalmente intensivi prima di essere pubblicati su una vera blockchain. Inoltre, una volta che i contratti sono deployati, possono essere gestiti direttamente dall'IDE, facilitando il monitoraggio delle transazioni in tempo reale.
In conclusione Remix è un IDE essenziale per chiunque si occupi dello sviluppo di smart contracts.[4]

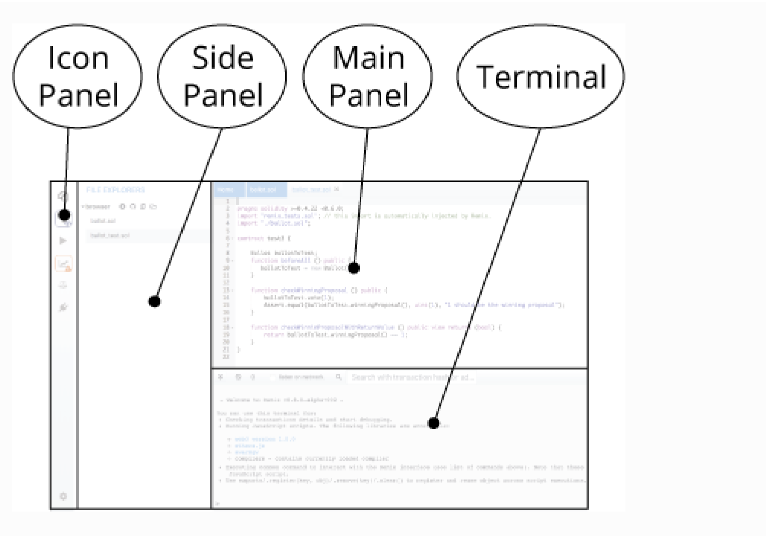

Figura 4: Layout Remix-IDE

- **Pannello delle Icone:** Pannello per selezionare e cambiare i plugin.
- **Pannello Laterale** - Pannello dove è possibile visualizzare e interagire con l'interfaccia grafica del plugin.
- **Pannello Principale** Pannello dove vengono visualizzati e modificati i file, non solo in formato solidity.
- **Terminale** Spazio dove vengono mostrati i risultati delle interazioni con il pannello principale, è anche possibile eseguire script.

2.6 Brownie

Brownie è un framework basato su web3.py(la libreria python per Ethereum) per lo sviluppo e il testing di smart contracts. Si può utilizzare per sviluppare smart contracts compatibili con Ethereum o altre reti basate su EVM. Brownie supporta contratti scritti sia in Solidity che in Vyper.

2.7 Ganache

Per simulare la rete è stato utilizzato Ganache , un ambiente locale di sviluppo che simula un singolo nodo della rete, utile per eseguire test e simulazioni senza preoccuparsi di Gas e altre problematiche.

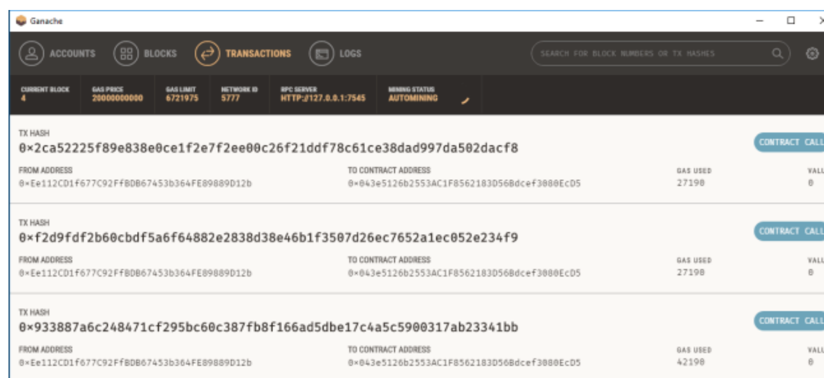

Figura 5: Ganache UI

2.8 Sepolia Testnet

Nel mondo delle blockchain, lo sviluppo e il testing sono fondamentali per garantire la sicurezza e l'affidabilità delle applicazioni decentralizzate (DApp). Le testnet sono blockchain utilizzate a scopo di test, separate dalla mainnet, la rete principale in cui vengono eseguite le transazioni reali con criptovalute reali.
Sepolia è una testnet progettata per Ethereum, che a differenza delle altre testnet come Goerli, Kovan o Rinkeby è stata concepita per fornire un ambiente di test più stabile e gestibile. Le testnet permettono agli sviluppatori di sperimentare nuovi Smart contract, migliorare il protocollo, ed eseguire prove senza rischiare fondi o sovraccaricare la mainnet.

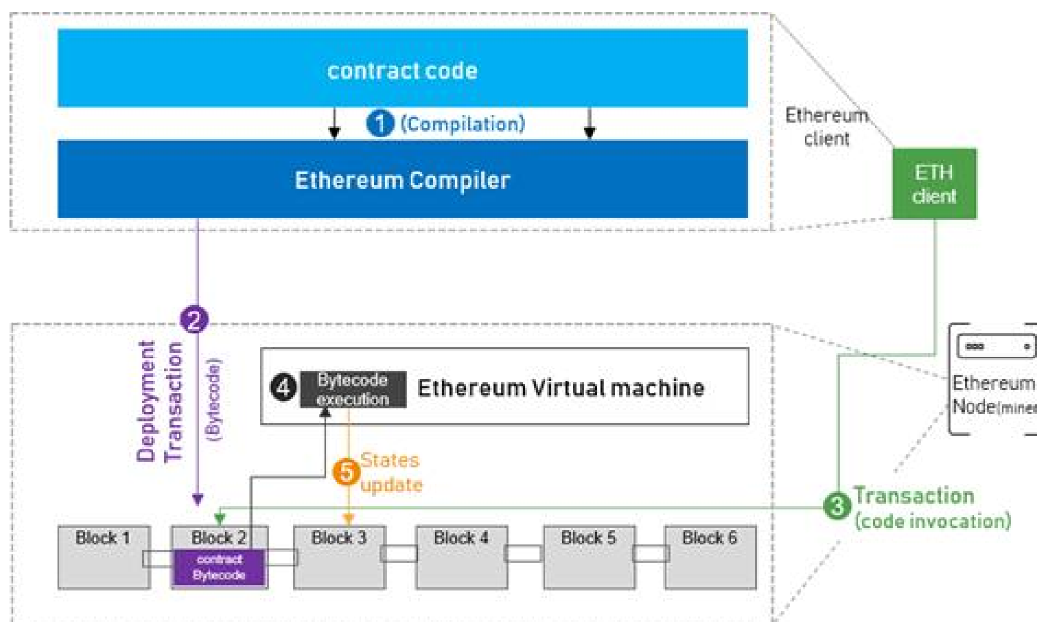

Figura 6: Deploy Smart contract

In figura è mosrtato come viene eseguito il Deploy di uno smart contract su una rete ethereum come Sepolia.
Il codice dopo essere compilato e trasformato in Bytecode viene salvato all'interno di un blocco della chain.
E' importante sottilineare che un partecipante non è in grado di visualizzare il codice sorgente ma solo il Bytecode, e per interagire con lo Smart contract è necessario procedere alla verifica del codice da "sepolia.etherscan.io".

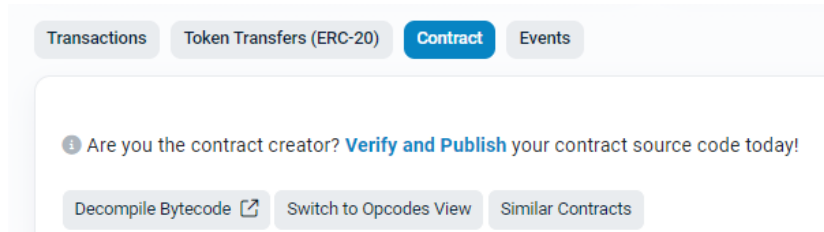

Figura 7: Smart contract non verificato

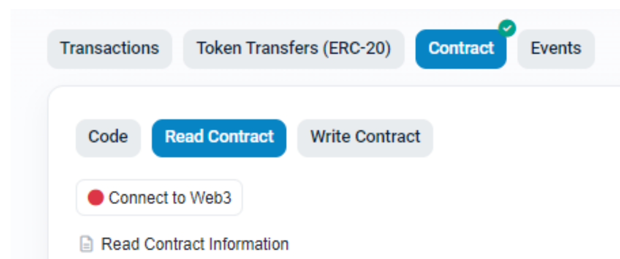

Figura 8: Smart contract verificato

Come mostrato in figura solo dopo la verifica è possibile collegare un indirizzo Ethereum e interagire con esso.

3 Zero-knowledge proofs

Una Zero-Knowledge Proof (ZKP), o Prova a Conoscenza Zero, è un metodo crittografico che consente a una parte (**il prover**) di dimostrare a un'altra parte (**il Verifier**) che *una dichiarazione è vera senza rivelare alcuna informazione aggiuntiva oltre alla validità della dichiarazione stessa.*

3.1 Caratteristiche principali di una Zero-Knowledge Proof

Formalmente tali prove godono di tre caratteristiche principali:

Completezza (Completeness)

Se la dichiarazione è vera, un Verifier onesto sarà convinto di questa verità da un prover onesto. In altre parole, se il prover e il Verifier seguono correttamente il protocollo, il Verifier accetterà sempre la prova se la dichiarazione è vera.

Solidità (Soundness)

Se la dichiarazione è falsa, nessun prover (anche se disonesto) può convincere un Verifier onesto che è vera, se non con una probabilità trascurabile. Questo significa che è estremamente difficile (quasi impossibile) per un prover disonesto ingannare il Verifier riguardo una dichiarazione falsa.

Conoscenza Zero (Zero-Knowledge)

Se la dichiarazione è vera, il Verifier non apprende alcuna informazione aggiuntiva oltre al fatto che la dichiarazione è vera. Significa che il Verifier non ottiene conoscenze utili che potrebbero rivelare il segreto o qualsiasi altra informazione confidenziale.

3.2 interactive zero-knowledge proofs

Le Interactive Zero-Knowledge Proofs (IZKP) sono una tipologia di protocollo crittografico che richiede molteplici interazioni tra due parti. Attraverso una serie di scambi, il prover dimostra la veridicità di una dichiarazione senza rivelare altre informazioni.[5]

Figura 9: Esempio funzionamento IZKP

nell'esempio in figura Alice(Prover) ripeterà l'interazione finchè Bob(Verifier) non sarà convinto che Alice conosca il codice della porta.
la probabilità che Alice preveda il percorso che Bob sceglierà per N iterazioni è pari a:

$$P = \left(\frac{1}{2}\right)^N = \frac{1}{2^N}$$

Per N elevato è estremamente probabile che alice conosca il codice.

3.3 Non-interactive zero-knowledge proofs

Le prove interattive sono limitate in quanto richiedono che entrambe le parti siano disponibili e interagiscano numerose volte.
Anche se un verificatore potesse fidarsi dell'integrità di un dimostratore, la prova non sarebbe utilizzabile per verifiche indipendenti, dato che ogni nuova prova necessiterebbe di una nuova serie di scambi tra il dimostratore e il verificatore.
Per superare questa limitazione 3 ricercatori: Paul Feldman, Silvio Micali e Manuel Blum, hanno introdotto le prime prove a conoscenza zero non interattive.
Che come verrà visto permette al dimostratore di mostrare la propria conoscenza di determinate informazioni con una singona interazione.
A differenza delle prove interattive, quelle non interattive richiedono un unico ciclo di comunicazione tra i partecipanti.
Il dimostratore utilizza le informazioni segrete in ingresso a un algoritmo che genera una prova a conoscenza zero, che verrà poi inviata al verificatore per il controllo.
Le prove non interattive riducono la comunicazione tra dimostratore e verificatore, rendendo le prove ZK più efficienti. Inoltre, una volta generata una prova, questa è disponibile per chiunque altro per la verifica.

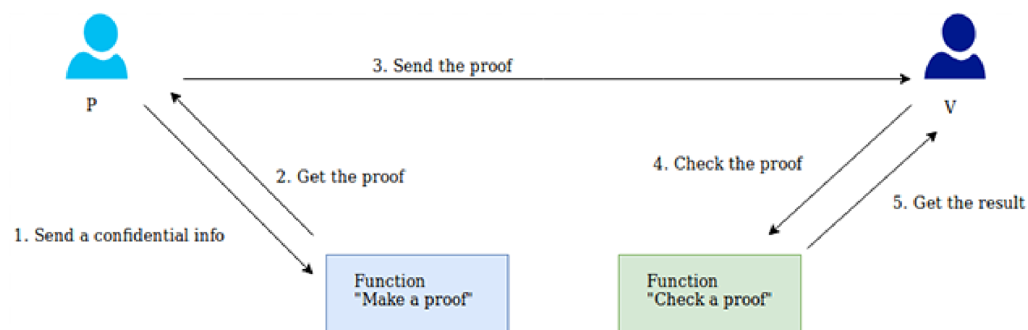

Figura 10: Meccanismo del Non-interactive zero-knowledge proof

3.4 Tipologie di ZKP non interattive

Esistono diversi tipi di ZKP, in questa sezione verranno elencati quelli che sono maggiormente soggetto di studio.

3.4.1 zkSNARK

zkSNARK è un acronimo per "Zero-Knowledge Succinct Non-Interactive Argument of Knowledge".
Il protocollo zkSNARK presenta una serie di caratteristiche:

- **Zero-knowledge (Conoscenza Zero)**: Un verificatore può convalidare un'affermazione senza sapere nient'altro sull'affermazione stessa. L'unica informazione che il verificatore possiede è se l'affermazione è vera o falsa.
- **Succinct (Succinto)**: La prova zero-knowledge è più piccola del "witness" (testimone) e può essere verificata rapidamente.
- **Non-interactive (Non Interattivo)**: La prova è 'non interattiva' perché il dimostratore ("prover") e il verificatore interagiscono una sola volta, a differenza delle prove interattive che richiedono più cicli di comunicazione.
- **Argument (Argomento)**: La prova soddisfa il requisito di 'soundness', quindi imbrogliare risulta estremamente improbabile.
- **Of Knowledge (Di Conoscenza)**: La prova zero-knowledge non può essere costruita senza l'accesso all'informazione segreta (il testimone). È difficile, se non impossibile, per un dimostratore che non dispone del testimone calcolare una prova zero-knowledge valida.

3.4.2 zkSTARK

zkSTARK, acronimo di "Zero-Knowledge Scalable Transparent Argument of Knowledge", rappresenta un'evoluzione rispetto agli zkSNARK, offrendo vantaggi significativi in termini di scalabilità e trasparenza.

- **Scalabilità:** Mentre gli zkSNARK soffrono di una crescita lineare dei tempi di generazione e verifica delle prove all'aumentare della dimensione del "witness" (insieme di dati oggetto della prova), zkSTARK mantiene un incremento solo lieve, rendendolo più efficiente per la gestione di grandi volumi di dati.

- **Trasparenza intrinseca:** A differenza degli zkSNARK che si basano su un setup iniziale affidato a terzi, zkSTARK trae la sua casualità da fonti pubblicamente verificabili, eliminando la necessità di riporre fiducia in entità esterne e garantendo una maggiore trasparenza e robustezza del sistema.

- **Dimensioni e costi:** Sebbene le prove generate da zkSTARK siano generalmente più grandi rispetto a quelle degli zkSNARK, comportando costi di verifica più elevati, la sua efficienza in termini di scalabilità lo rende una scelta vantaggiosa in scenari che coinvolgono grandi dataset.

3.4.3 Bulletproofs

I Bulletproofs sono una particolare forma di Zero-Knowledge Proofs non interattive, che non richiedono una configurazione fidata.
I Bulletproofs offrono una serie di vantaggi rispetto a sistemi come i SNARKs. A differenza dei SNARKs non richiedono una configurazione fidata , non richiedono la cosidetta "trusted setup", eliminando così una potenziale vulnerabilità.
La capacità di aggregare le proofs e la loro natura non interattiva li rende scalabili e adatti all'uso in sistemi di blockchain affollati e ad alta frequenza.

3.4.4 zk-EVM (Zero-Knowledge Ethereum Virtual Machine)

Una Ethereum Virtual Machine zero-knowledge (zkEVM) è una macchina virtuale che esegue transazioni di smart contract in modo compatibile sia con i calcoli delle prove a conoscenza zero sia con l'infrastruttura Ethereum esistente. Consentendo di eseguire rollup zero-knowledge, soluzioni di scaling layer-2 che aumentano il volume delle transazioni riducendo i costi.

3.5 Applicazioni teoriche

In questa sezione si andranno ad analizzare alcune applicazioni realizzabili grazie alla tecnologia delle ZKP.

3.5.1 Transazioni anonime

Nell'era digitale, la tutela della privacy finanziaria è sempre più sotto i riflettori. Le transazioni con carte di credito, lasciano tracce visibili a diversi enti, intaccando la segretezza degli utenti. Le criptovalute, create per decentralizzare e democratizzare i sistemi di pagamento, non sempre assicurano l'anonimato previsto. Sebbene le transazioni su blockchain pubbliche siano protette da pseudonimi, spesso è possibile collegarle a identità reali tramite analisi dei dati sia sulla blockchain che esterni ad essa. Per affrontare questo problema, sono state introdotte le "privacy coin", blockchain specializzate come Zcash che cifrano i dettagli delle transazioni, rendendo opachi indirizzi, importi e persino il tipo di asset scambiato.
Questa tecnologia, basata su prove a conoscenza zero, permette ai nodi di validare le operazioni senza accedere ai dati sensibili. [5] [6]
Nel contesto della filiera questa tecnologia può permettere lo scambio di denaro tra le varie aziende senza la necessita di rendere pubblico l'importo, il destinatario e il prodotto/servizio acquistato.
Il contesto però non riguarda solo l'aspetto economico, infatti le ZKP possono essere impiegate per confermare l'origine di un materiale o prodotto senza rivelare informazioni sensibili o proprietarie sul fornitore o sui processi di produzione.
Per esempio, un produttore potrebbe dimostrare che i suoi prodotti provengono da fonti sostenibili, senza dover esporre i dettagli specifici dei suoi fornitori.

3.5.2 Identità decentralizzata e autenticazione

I sistemi di gestione dell'identità attuali mettono a rischio i dati personali. Tuttavia, le prove a conoscenza zero offrono una soluzione interessante: permettono alle persone di verificare la propria identità senza dover rivelare informazioni sensibili. Queste sono particolarmente utili nel contesto dell'identità decentralizzata. Prendendo come riferimento [7], L'identità decentralizzata (anche detta "identità autogestita" o "self-sovereign") offre agli individui la possibilità di controllare l'accesso ai propri identificatori personali.

Questa tecnologia può essere molto utile nel contesto della filiera. Il primo quesito che ci si pone è dove porre la fiducia, cioè chi garantisce che l'identità decentralizzata legata all' indirizzo è corretta e che le credenziali verificabili associate sono lecite.
Sono necessarie una o più entità che creano questi oggetti e li distribuiscono all'interno della blockchain, successivamente ogni organizzazione può utilizzarle a proprio piacimento per usufruire di determinati servizi senza l'ausilio di un terzo.
Una volta emesse è possibile interagire con le altre organizzazioni della filiera in modo autonomo e diretto(sia identità sia credenziali è necessario che abbiano date di scadenza e siano rinnovate periodicamente).

3.5.3 Calcolo Verificabile

Il calcolo verificabile (verifiable computation) rappresenta un'altra importante applicazione della tecnologia a conoscenza zero per il miglioramento delle architetture blockchain. Questa tecnologia permette di esternalizzare il calcolo ad altre entità, mantenendo al contempo l'affidabilità dei risultati. L'entità che esegue il calcolo fornisce il risultato insieme a una prova che ne attesta la correttezza.
Il calcolo verificabile si occupa del seguente problema: "un verificatore fornisce un programma C con input $\vec{x}$ a un provatore. Successivamente, riceve una rivendicazione del risultato $\vec{y}$ dal provatore. Come può il verificatore V essere sicuro che il provatore P abbia calcolato correttamente $\vec{y} = C(\vec{x})$?"[8]
Per risolvere questo problema, V coinvolge P in un protocollo che consente a V di assicurarsi che P abbia calcolato C correttamente. Il calcolo verificabile è fondamentale per migliorare le velocità di elaborazione sulle blockchain senza ridurre la sicurezza.

Per comprendere meglio l'affermazione precedente si prende come esempio lo scaling [6]:

- Le soluzioni di scaling on-chain, come lo sharding, richiedono ampie modifiche al livello base della blockchain (base layer). Tuttavia, questo approccio è estremamente complesso e gli errori di implementazione possono compromettere il modello di sicurezza di Ethereum.
- Le soluzioni di scaling off-chain non richiedono la riprogettazione del protocollo Ethereum di base. Si basano, invece, su un modello di calcolo esternalizzato per migliorare il throughput sul base layer di Ethereum.

Funzionamento:

1. Invece di elaborare ogni transazione, Ethereum demanda l'esecuzione a una chain separata.
2. Dopo aver elaborato le transazioni, la chain esterna restituisce i risultati da applicare allo stato di Ethereum.

Il vantaggio sta nel fatto che Ethereum non deve effettuare esecuzioni dirette, ma solo applicare i risultati del calcolo esternalizzato al suo stato. Ciò riduce

la congestione della rete e migliora la velocità delle transazioni. Quando un nodo esegue una transazione al di fuori di Ethereum, fornisce una prova a conoscenza zero per dimostrare la correttezza dell'esecuzione off-chain. Questa prova (chiamata "validity proof") garantisce la validità di una transazione, consentendo a Ethereum di applicarne il risultato al suo stato senza dover attendere eventuali contestazioni.

4 ZoKrates

4.1 Introduzione

ZoKrates è un toolbox per zkSNARKs su Ethereum. Il suo scopo è implementare le ZKP in un linguaggio di alto livello, partendo dal circuito aritmetico fino al verificatore in Solidity.
La nascita di Zokrates deriva da una serie di fattori, in primo luogo le prove a conoscenza zero sono concettualmente complesse e difficili da implementare, basti pensare che gli zkSNARKs necessitano di una profonda conoscenza dei protocolli matematici crittografici e delle curve ellittiche(RSA non è usato).
Inoltre l'integrazione di questa tecnologia su blockchain pubbliche come Ethereum non è banale dato che queste non sono state originariamente sviluppate per supportare ZKP.
Zokrates permette agli sviluppatori di rendere più semplice la scrittura, il test e deployment di applicazioni che usano zkSNARK.

4.2 Linguaggio di programmazione

ZoKrates utilizza un linguaggio di programmazione specifico di alto livello progettato per facilitare lo sviluppo di applicazioni che incorporano prove a conoscenza zero. Il linguaggio è studiato per implementare in modo efficente la creazione di circuiti, che sono essenziali per la generazione e la verifica delle prove.
Ha una sintassi simile a C, si basa su una Tipizzazione statica per evitare errori di runtime e aumentare sicurezza del codice e come altri linguaggi supporta funzioni riusabili e strutturare il codice in moduli, facilitando la creazione di programmi complessi.
Implementare circuiti aritmetici pone diversi limiti, dovuti all'utilizzo di curve ellittiche e sicurezza del protocollo, Zokrates utilizza fondamentalmente 2 tipologie di elementi mentre le altre sono una derivazione:

Tipi Primitivi

- **Field**: Rappresenta un elemento di un campo finito, usato comunemente in applicazioni crittografiche. I valori di tipo *field* si comportano come interi non segnati, ma con un comportamento specifico per l'overflow basato sul numero primo grande p.
- **Bool**: Variabili booleane che possono assumere valori *true* o *false*.
- **Interi non segnati**: Tipi come `u8`, `u16`, `u32`, `u64` rappresentano numeri positivi con un'aritmetica definita modulo una potenza di due, ad esempio 2^8 per `u8`.

Tipi Complessi

- **Array**: Gli array in ZoKrates devono avere una lunghezza nota a tempo di compilazione. Possono contenere elementi di qualsiasi tipo e possono essere multidimensionali.
- **Structs**: Strutture che rappresentano una collezione di valori con nomi specifici. Possono anche essere generici, consentendo l'utilizzo di array di dimensione generica all'interno.

4.3 Workflow di Zokrates

1. **Scrivere lo script:** Scrivere un programma ZoKrates che delinea il calcolo che si vuole dimostrare e i vincoli che deve soddisfare (ad esempio, conoscere un numero che sommato a 5 dà come risultato 23).

2. **Compilazione:** Il compilatore ZoKrates trasforma il programma in un circuito aritmetico. Questo è sotto forma di R1CS permettendo il calcolo usando i polinomi.

3. **Computazione:** Per generare un Witness valido Zokrates richiede l'inserimento degli input pubblici e privati, tali che generino un corretto calcolo.

4. **Configurazione(Trusted Setup):** Durante questa fase viene generata una CRS(common Reference String) una stringa di bit pseudo casuali fondamentale per la sicurezza del protocollo, le chiavi di generazione della prova e di verifica(il CRS è usato per costruire entrambe le chiavi).

5. **Generazione della prova off-chain:** Utilizzando il witness e la chiave di generazione della prova, ZoKrates crea una prova zkSNARK.

6. **Esportazione del Verifier:** Per costruire il Verifier.sol, Zokrates utilizza il R1CS e la chiave di verifica (che sarà già integrata nel verificatore).

7. **Invio alla blockchain:** si inviano prova, smart contract di verifica e opzionalmente informazioni aggiuntive sulla blockchain.

8. **Verifica:** Il contratto utilizza prova, input opzionali e chiave di verifica per controllare la prova. Se la prova è valida, il contratto restituisce un messaggio di corretta esecuzione senza rivelare nessuna informazione.

4.4 Rappresentazione intermedia di Zokrates

4.4.1 Circuiti aritmetici

Definizione: Un circuito è un grafo orientato e aciclico in cui i nodi sono chiamati gate e gli archi sono chiamati wires[8]. I gate calcolano funzioni sui loro fili di input per derivare valori per i loro wires di output. I circuiti aritmetici sono circuiti in cui i gate eseguono operazioni aritmetiche e i wires hanno valori che generalmente non sono binari. Ogni filo porta un valore da Fp e ogni gate esegue un'operazione di campo primo, cioè addizione o moltiplicazione.

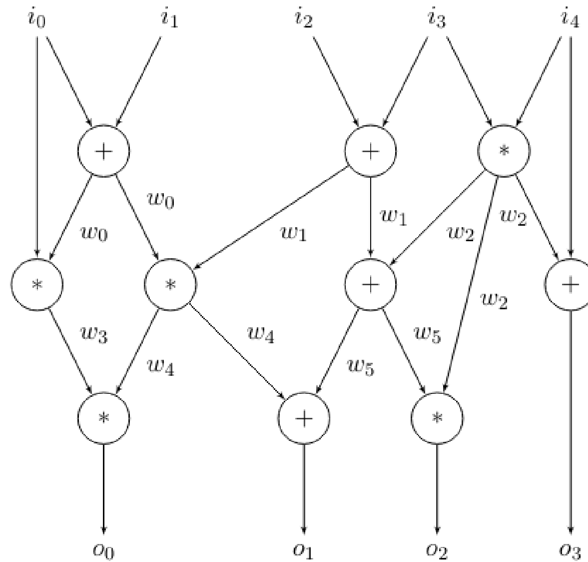

Figura 11: Rappresentazione circuito aritmetico

Gli zk-SNARK possono essere utilizzati per dimostrare la soddisfacibilità di un circuito per dati input $\vec{i}$ e output $\vec{o}$, cioè per certificare che esiste un testimone $\vec{w}$ tale che $C(\vec{i}, \vec{w}) = \vec{o}$ per un dato circuito aritmetico C.
Alcune caratteristiche del circuito sono:

- Un circuito aritmetico è "soddisfatto" se esiste un'assegnazione di valori ai suoi fili (un "witness") che rende vere tutte le operazioni del circuito.

- In questo processo, la proprietà zero-knowledge permette al witness di rimanere privato al prover.

- Il problema di trovare un witness che soddisfi un circuito è noto come *Circuit Satisfiability Problem* (CSP) ed è un problema NP-completo (computazionalmente difficile).

4.4.2 SLP

A differenza dei circuiti aritmetici, gli SLP (Straight-Line Programs) sono una sequenza di istruzioni relativa a una computazione e ogni "passo" di un SLP corrisponde a un gate di un circuito aritmetico.[8]

Gli SLP sono chiamati "straight-line" (lineari) perché esprimono programmi senza salti o cicli, il che significa che le istruzioni vengono eseguite in sequenza, dall'inizio alla fine.

- Dato un circuito aritmetico, è possibile costruire un SLP equivalente che esegue lo stesso calcolo.
- Viceversa, dato un SLP, è possibile costruire un circuito aritmetico equivalente rappresentando ogni passo del programma come un gate nel circuito.

Eseguendo un SLP si passa a una esecuzione sequenziale dove ad ogni passo si ottiene un valore intermedio. Questo set di valori viene definito come "execution track" che può essere trasformata in un witness per il circuito aritmetico corrispondente.
Questo witness è poi utilizzato per generare una prova zk-SNARK che attesta la correttezza del calcolo.

4.4.3 Rank-1 Costraint Systems

I Rank-1 Costraint Systems (R1CS) sono una rappresentazione matriciale dei circuiti aritmetici. Questa astrazione è spesso usata nelle implementazioni dei protocolli crittografici.
Un R1CS è essenzialmente un sistema di equazioni in cui ogni equazione è un vincolo che deve essere soddisfatto. Questi vincoli sono espressi come prodotti di combinazioni lineari.
In altre parole, l'obiettivo è trovare un'assegnazione di valori alle variabili (input, output e variabili interne) che soddisfi tutte le equazioni nel sistema di vincoli.
Questa rappresentazione è ampiamente utilizzata nei zk-SNARK per la sua maggiore efficenza computazionale.
Formalmente:
Consideriamo un circuito con n variabili (inclusi input, output e variabili interne).
Un vincolo R1CS può essere espresso come:

$$a \cdot w \times b \cdot w = c \cdot w$$

$$a \cdot w \times b \cdot w - c \cdot w = 0$$

Figura 12: Vettori a,b,c

dove:

$$w = (w_1, w_2, \dots, w_n)$$

è il vettore delle variabili del circuito.
a, b, c sono vettori di coefficienti che specificano il contributo di ciascuna variabile di w al vincolo.

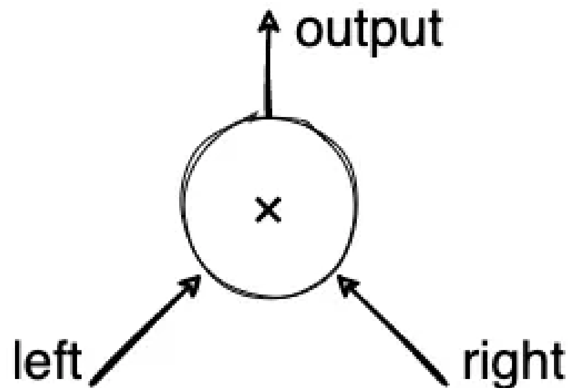

Figura 13: Singolo gate di un circuito aritmetico

w nel contesto ZKP corrisponde al witness, e comprende l'insieme di valori che soddisfano tutti i vincoli del circuito.

- **Vettore a**: Contiene i coefficienti che moltiplicano le variabili nel termine "left" del prodotto di R1CS.
- **Vettore b**: Contiene i coefficienti per le variabili nel termine "right".
- **Vettore c**: Contiene i coefficienti per le variabili nel termine "output", ovvero il risultato del prodotto dei primi due termini che deve essere eguagliato.

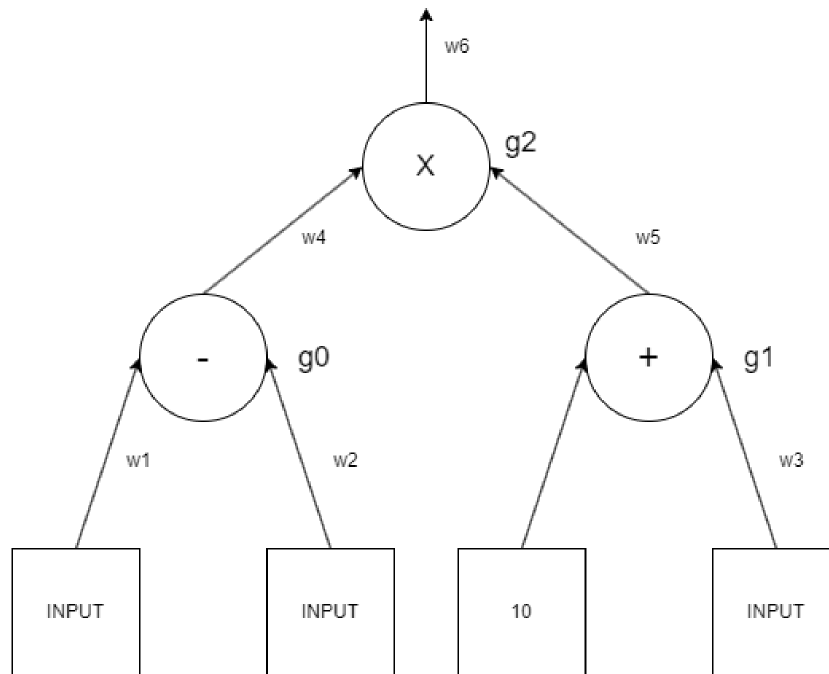

Figura 14: Esempio circuito aritmetico

Le equazioni relative:
g0: IN - IN_2 = w4
g1: 10 + IN = w5
g2: w4 + w5 = w6

4.5 Zokrates verifiable computation

è stato discusso nella sezione 3.5.3 le caratteristiche del calcolo verificabile dal punto di vista puramente teorico, in questa sezione si vuole discutere come eseguire questa operazione usando Zokrates.
Si parte da una premessa:

La correttezza dell'esecuzione di un programma può essere dimostrata attraverso la soddisfacibilità di un circuito che rispecchia la logica del programma.

(supponendo che il programma possa essere rappresentato come mostrato in figura 11). L'idea è partire dal programma, trasformarlo in circuito aritmetico e usare uno zkSNARK per dimostrare la validità della traccia di esecuzione di un SLP o R1CS.

1. **Trasformazione programma in SLP:** Viene generato un R1CS o SLP che segue la logica del programma di partenza.
2. **Esecuzione dell'SLP:** L'SLP viene eseguito passo dopo passo per ottenere una traccia di esecuzione.
3. **Trasformazione in witness:** La traccia di esecuzione viene trasformata in un witness del circuito aritmetico corrispondente.
4. **Generazione della prova zk-SNARK:** Dato questo witness, si calcola una prova zkSNARK che certifica la soddisfazione di questo circuito. Questo avviene se e solo se la traccia di esecuzione è valida, cioè il calcolo è avvenuto correttamente.

$$w = (w_1, w_2, \dots, w_n)$$

sarà la traccia che rappresenta la corretta computazione di un algoritmo.

5 Applicazione delle Zero-Knowledge Proofs alle filiere produttive

5.1 Introduzione

Nell'ambito della filiera produttiva si sono individuati 3 applicazioni dove ZKP può trovare implementazione:

- **Commitment Scheme per Documenti Confidenziali:** Utilizzo di un "commitment scheme"[9] per un documento contenente informazioni confidenziali (segreto), dove il commitment è salvato su blockchain. Questo assicura che il documento sia immutabile e verificabile pubblicamente senza rivelare le informazioni sensibili contenute.

- **Certificazione del Calcolo del GHG:** Un utente utilizza un servizio per il calcolo dell'emissione dei gas serra (GHG) e desidera certificare che il calcolo sia stato eseguito correttamente. Per fare ciò, genera una prova, conservata all'interno della blockchain, che è visualizzabile da tutti, mantenendo segreti gli input e il servizio di terze parti utilizzato.

- **Certificazione dell'Uso di un Algoritmo:** Certificazione dell' utilizzo di un algoritmo interno all'azienda per svolgere dei calcoli. In questo caso, è necessario dimostrare di aver usato l'algoritmo in questione e che il calcolo è stato eseguito correttamente.

In questo capitolo ci si soffermerà sul primo punto e verranno proposti due sistemi, il primo utilizza zkSNARK per generare un commitment sotto forma di prova, il secondo utilizza solamente un hash.
Nella sezione successiva verranno quindi esposti il contesto e le ragioni per cui è necessario tracciare le emissioni di gas serra senza rivelare determinate informazioni a tutti i partecipanti della rete.
Nella sezione 5.3 verrà mostrata un esempio di implementazione pratica che sfrutta il toolbox Zokrates.
Mentre nella sezione 5.7 viene esposto un sistema che utilizza un hash come commitment e un possibile attacco ad esso.
Questo sarà fondamentale per spiegare il vantaggio più rilevante delle ZKP.

5.2 La filiera e i partecipanti

Una filiera è un insieme di processi e attività che portano alla produzione e distribuzione di un prodotto o servizio, coinvolgendo diverse organizzazioni e soggetti.
Ogni partecipante nella filiera contribuisce con una fase specifica, come la produzione di materie prime, la trasformazione, la distribuzione e la vendita al dettaglio.
La gestione efficiente della filiera è cruciale per garantire la qualità del prodotto finale, la tempestività delle consegne e la riduzione dei costi.
Il tracciamento delle emissioni di gas serra lungo la filiera è obbligatorio. Secondo le normative europee le aziende devono rendere conto del loro impatto ambientale, il Monitoring and Reporting Regulation (MMR) richiede che queste monitorino, riportino e verifichino le loro emissioni di CO2 in conformità con standard specifici.
L'obiettivo di queste normative mira a migliorare la trasparenza e la responsabilità delle aziende riguardo il loro impatto ambientale, facilitando il monitoraggio e la riduzione delle emissioni di gas serra.
Però è necessario comprendere che parte parte di queste informazioni riguardano processi o dati confidenziali dell'azienda, che devono rimanere tali agli occhi degli altri partecipanti.
Le ZKP sono una tecnologia che può essere utile nel tracciamento dei dati di una filiera.

La situazione teorica è la seguente: E' presente una blockchain di tipo permissioned, dove a ogni azienda o ente è assegnato un "public address", attribuito da un "issuer" autorità che distribuisce identità e credenziali.

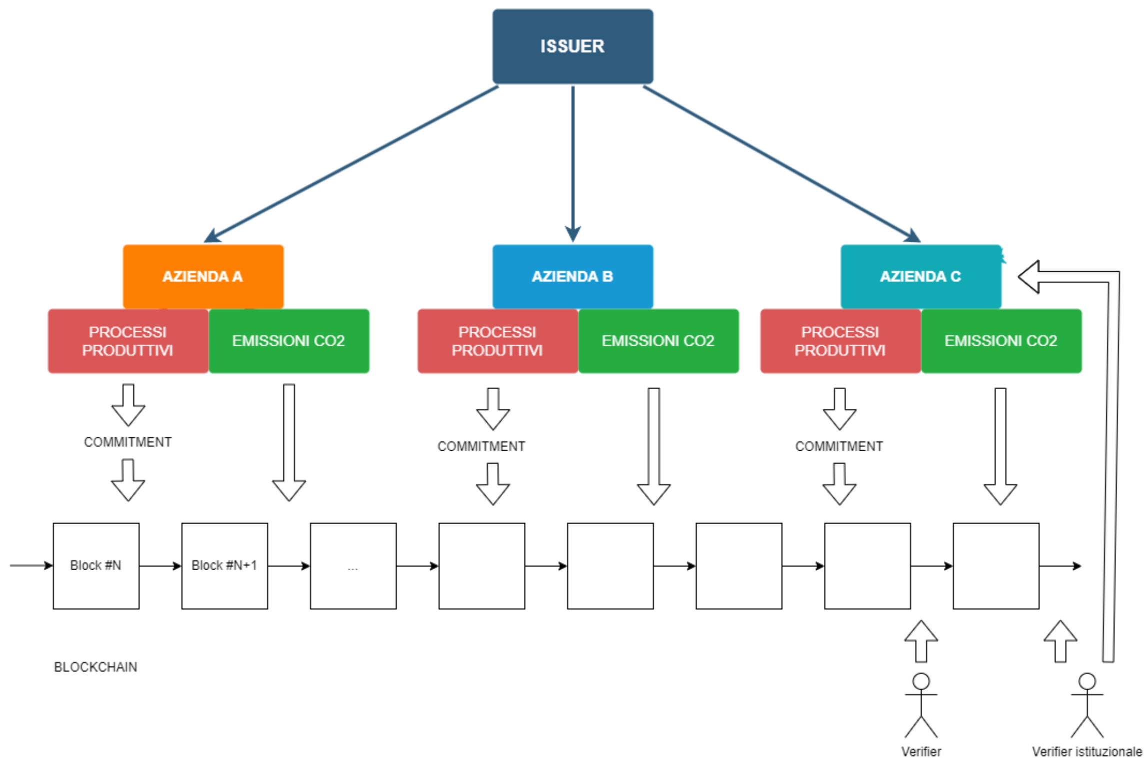

Figura 15: Architettura della filiera con tracciamento su blockchain

Come mostrato in figura la filiera è composta da diverse organizzazione che operano fra loro, queste hanno l'obbligo di generare 2 documenti, il primo riguardante i gas CO2 emessi dalle attività della filiera, e il secondo i processi produttivi.
Una volta ottenuti questi 2 documenti, l'azienda deve pubblicare su blockchain "emissioni CO2" mentre il secondo documento deve rimanere segreto e deve essere rilasciata una traccia della generazione su blockhain, in modo tale che se in futuro un "Verifier istituzionale" volesse controllare che è stato eseguito tutto a norma e che il documento non è stato modificato(commitment) può andare fisicamente dall'organizzazione e in loco eseguire il controllo.
Il "Verifier" invece è un qualsiasi partecipante della rete che può accedere alla blockchain e controllare le transazioni degli altri utenti.

5.3 Applicazione delle ZKP nel contesto della filiera

In figura 16 è mostrato "Gas_emessi" documento che contiene le misurazioni in un determinato intervallo di tempo, questo deve essere salvato sulla blockchain pubblica all'interno di uno Smart Contract che contiene tutte le registrazioni depositate dagli utenti. Attraverso le apposite 3 funzioni è quindi possibile caricare e scaricare il documento pubblico:

- **Upload**: Caricare il documento all'interno di un array.
- **Full_retrieve**: Recuperare l'intero array di dati salvati.
- **Retrieve**: Recuperare un elemento dell'array.

```
1  pragma solidity ^0.8.0;
2
3  contract TextArrayStorage {
4      string[] private storedTexts; // array che memorizza il
       testo
5
6      function upload(string memory _text) public {
7          storedTexts.push(_text); // aggiunge testo all'array
8      }
9
10     function full_retrieve() public view returns (string[]
       memory) {
11         return storedTexts; // restituisce l'intero array
12     }
13      function retrieve(uint256 index) public view returns (
       string memory) {
14
15         if (index < storedTexts.length) {
16             return storedTexts[index];
17         } else {
18             return "Invalid index";
19         }
20     }
21 }
```

Il valore di CO2 emesso è pubblico e visualizzabile da tutti i partecipanti. Il documento "Processi_produzione_gas" contiene invece informazioni segrete che non vogliono essere rese pubbliche, ma che è comunque necessario caricare per dimostrare a un controllore di aver svolto i processi secondo le norme vigenti[9].

La soluzione è quella di andare a generare una prova a conoscenza zero per poi caricarla su blockchain, in tal modo qualsiasi partecipante potrà verificare che la prova insieme ai dati pubblici forniti sia valida, senza che però vengano a sapere dei processi produttivi dell'azienda. In particolare, la prova viene prodotta off-chain attraverso un algoritmo ZKP che richiede in ingresso l'hash della transazione legata al caricamento del documento Gas_emessi e il documento segreto.
L'algoritmo genererà in primo luogo il circuito aritmetico associato allo script Zokrates, successivamente grazie ai valori in ingresso sarà in grado di generare un witness valido(si ricorda che il witness si dice valido se e solo se questo insieme di valori soddisfa il circuito aritmetico, sezione 4.4.1).
Per generare chiave di prova e verifica viene usata una fase di "trusted setup" dove viene utilizzata una fonte casuale per generare le chiavi(le chiavi non sono funzioni del witness ma solo della fonte random e dei vincoli del circuito aritmetico). Attraverso chiave di prova e witness, Zokrates a questo punto è in grado di generare una prova valida.
Come ultimo step il circuito aritmetico viene trasformato in un Verifier in linguaggio solidity. Il Verifier riceve in input tre elementi fondamentali:

- **Prova a conoscenza zero**
- **Dati pubblici (hash_tx):** Nell'ambito di questa applicazione, i dati pubblici includono solo l'hash della transazione.
- **Chiave di verifica:** Già contenuta all'interno del Verifier, utilizzata per confermare la validità della prova fornita.

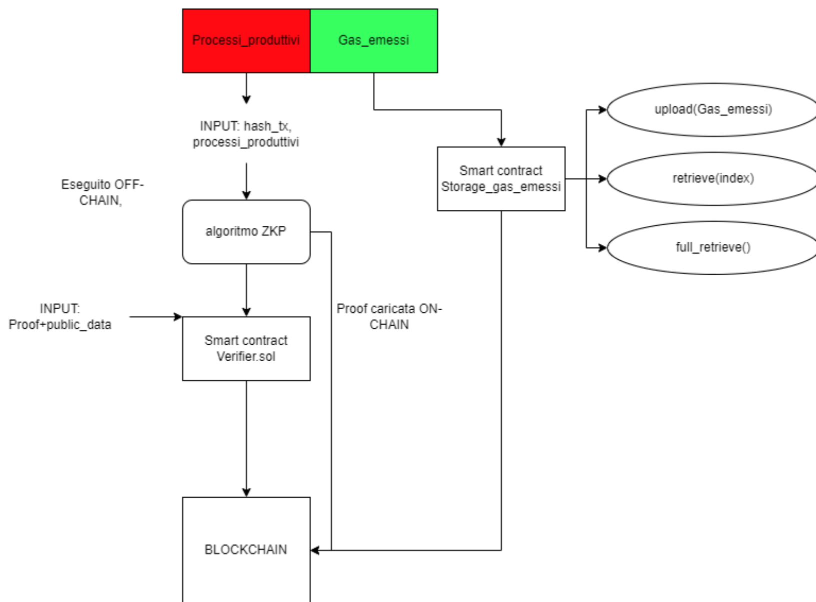

Figura 16: Applicazione ZKP filiera

5.4 Verifier

La generazione del Verifier segue il Workflow di Zokrates, è possibile recuperare i passaggi principali alla sezione 4.3. Partendo dal primo punto è stato scritto lo Script .zok che specifica il calcolo da dimostrare e i relativi vincoli da soddisfare.

```
1 def main(field hash_tx, private field pr_gas_serra) {
2
3     field expected_hash_tx = 41255345435647;          //mapping
    hash
4     field expected_pr_gas_serra = 5436534674574;      //mapping
    documento segreto(string)
5
6     assert(hash_tx == expected_hash_tx);
7     //bool match1 = hash_tx == expected_hash_tx;
8
9     assert(pr_gas_serra == expected_pr_gas_serra);
10    //bool match2 = hash_pr_gas_serra ==
    expected_hash_pr_gas_serra;
11
12    return ;
13 }
```

Attraverso il compilatore ZoKrates lo script viene trasformato in circuito aritmetico. ZoKrates come detto impiega R1CS per le sue rappresentazioni, e quindi genera 3 matrici A, B, C. Per generare un Witness valido il circuito aritmetico dovrà soddisfare la condizione:
Un circuito aritmetico è "soddisfatto" se esiste un'assegnazione di valori ai suoi fili (un "witness") che rende vere tutte le operazioni del circuito.
In altre parole il circuito è soddisfatto se gli input forniti al programma generano una computazione corretta e non errore.

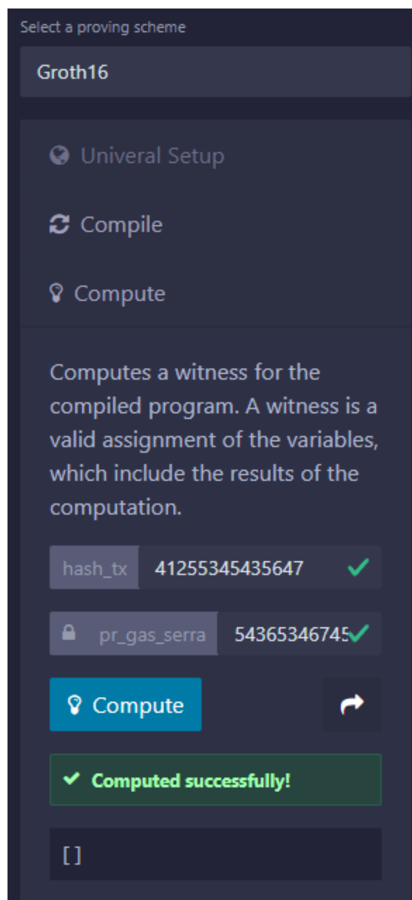

Figura 17: Computazione eseguita correttamente

Come esposto nella sezione 3.4.1 le chiavi di prova e verifica vengono generate durante la fase di Trusted Setup, è importante sottolineare che questa è fondamentale per assicurare la sicurezza del protocollo.

```
{
  "scheme": "g16",
  "curve": "bn128",
  "alpha": [
    "0
   x1a1666231bff684f4fa986c44efc36871308f3f90c8d48e6ad67864d8aa06

    4b4",
    "0
   x0d5ecc5a78c56e432883c80a9daa556e0340efe51353f14cb6d5aa59431c
```

```
8      1ffe"
9    ],
10   "beta": [
11     [
12       "0
     x08f4ff5b8ab01a73ac5f56bffb5b56505b1030bcc0472a4c1391de5632c
13       3e1b6",
14       "0
     x04b37ec94dde0bd36b803a859a33ce6ac3a01936357b9e6c74603f7cc9c
15       dffd2"
16     ],
17     [
18       "0
     x2031d80ce411f19c18108847a81eaa6b2e6db964407f6ff3c7bb6bb5345
19       8b3ca",
20       "0
     x0f8238d498b0695f1e7f82276ea13511b692dd19fd30ef7f8dabbde694d
21       395a8"
22     ]
23   ],
24   "gamma": [
25     [
26       "0
     x2202456700801b1133629b6d1d08808e1d7a09360433b400b527cb82634
27       bb690",
28       "0
     x250893f9086b8d44ffde26b3adf9f5b05373298cd94ed06fa7ef4f623ba
29       9ae1b"
30     ],
31     [
32       "0
     x186e6ac0286b41b6f140d9ea8b196a6a2110b2e658005e4a70568c6182b
33       fdcca",
34       "0
     x20fa92b85fb1ed82f72c86b5f5596a344ac03954b70f6a8798947e177b1
35       a3264"
36     ]
37   ],
38   "delta": [
39     [
40       "0
     x184ab1fd5dfc9124ab6eee9820e696cfa13bee6eafc415ad610a1875e89
41       295a1",
42       "0
     x21f4f6316cd6055e8031df45abac3a1e153e947b46c3cac0db6b5766127
```

```
43          4cd26"
44        ],
45        [
46          "0
      x2611eea1ad4b11b1e8783722f1b1e7f860571f7163f150d9ee7db0b57cc
47          d1342",
48          "0
      x2e7c7c6cffa4c6e57159af18fae05126fd08fb349779205865f8f36ff57
49          ab604"
50        ]
51    ],
52    "gamma_abc": [
53        [
54          "0
      x178fd43ef127d4f38318336aa684a9a744da8cd956de8ac051edc6a863a
55          4acd4",
56          "0
      x2dd85b5045736c36f4abf0328fc519ddbe2db983d219a78499f8fb0d02f
57          7cd86"
58        ],
59        [
60          "0
      x242fb7cbae9165082b6185245877e0d8267a360bf8caf94f6e009191f51
61          8d994",
62          "0
      x2c6986eeb6961056866b2112c50f290de9294547709aae90df4242df110
63          eafa1"
64        ]
65    ]
66 }
```

Verification_key.json

Ogni elemento rappresenta un punto sulla curva crittografica. (Es: "alpha":[coordinate_X,coordinate_Y]
La chiave di verifica è usata per controllare se la proof soddisfa effettivamente i vincoli legati ai punti crittografici(alpha, beta, gamma, delta, e gamma_abc), senza rivelare gli input originali[10].
Passando alla proof, questa è costituita da 3 componenti principali + input:

- **a**: Il primo elemento

```
1 ["0x2599842c3cf19dc360ba5dd982f7358bb1b957b4a828a5a3f75ca5f2
```

```
19319d1a", "0
    x2ebb061023e35fc97a8af540309c71f627f480b3f57ecf96b8bed095

687990d9"]
```

rappresenta un punto sulla curva ellittica in G1.

- **b**: Il secondo elemento

```
[["0x077b46c5e8a6e719a98b2f0c328a0409570f7b77b70dbd0ad0bb063
d9fe68861", "0
    x07e0eb4d05cf37c06cfa79a745314e630bfda5442362e33218436f5b

d210d776"], ["0
    x1464e941064ff5b7da65f18bc61a3f55e04041e472ad69d01c5302
f6770b58e5", "0
    x10c9eeb9fe40c679470e242b73318251c323d63b52a9039e1493946
2cc40bff1"]]
```

rappresenta un punto in G2.

- **c**: Il terzo elemento

```
["0x2b91a6a501584cd7bdd7bb79f118f251b9c718d0d302ca2e2a952a5
ccb57aed3", "0
    x280bb408010f71e70bc6c6bf7607eec45af1e87376570f8349f82d5e

94b3fd20"]
```

è un altro punto sulla curva G1, rappresenta la parte finale della proof ed è essenziale per completare la verifica dell'equazione nell'accoppiamento.

- **Input**: Il quarto elemento

```
["0x0000000000000000000000000000000000000000000000000000258
58235b3ff"]
```

non fa parte della proof, infatti in fase di verifica deve essere rimosso.

[10] [11]

5.4.1 Script Verifier.sol

Come definito nella sezione 4.3 il Verifier.sol è generato grazie a chiave di verifica e circuito aritmetico.

```
pragma solidity ^0.8.0;
library Pairing {
    struct G1Point {
        uint X;
        uint Y;
    }
    // Encoding of field elements is: X[0] * z + X[1]
    struct G2Point {
        uint[2] X;
        uint[2] Y;
    }
    /// @return the generator of G1
    function P1() pure internal returns (G1Point memory) {
        return G1Point(1, 2);
    }
    /// @return the generator of G2
    function P2() pure internal returns (G2Point memory) {
        return G2Point(
            
   [10857046999023057135944570762232829481370756359578518086990519
            9993285655852781,
            
   11559732032986387107991004021392285783925812861821192530917403151452391805634],
            
   [84956539231234314176049732474892724384181905872636001487702806
            49306958101930,
            
   4082367875863433681332203403145435568316851327593401208105741
             076214120093531]
        );
    }
    /// @return the negation of p, i.e. p.addition(p.negate())
   should be zero.
    function negate(G1Point memory p) pure internal returns (
   G1Point memory) {
        // The prime q in the base field F_q for G1
        uint q =
   21888242871839275222246405745257275088696311157297823662689037
```

```
33         78946
34         45226208583;
35         if (p.X == 0 && p.Y == 0)
36             return G1Point(0, 0);
37         return G1Point(p.X, q - (p.Y % q));
38     }
39     /// @return r the sum of two points of G1
40     function addition(G1Point memory p1, G1Point memory p2)
    internal view returns (G1Point memory r) {
41         uint[4] memory input;
42         input[0] = p1.X;
43         input[1] = p1.Y;
44         input[2] = p2.X;
45         input[3] = p2.Y;
46         bool success;
47         assembly {
48             success := staticcall(sub(gas(), 2000), 6, input, 0
    xc0, r, 0x60)
49             // Use "invalid" to make gas estimation work
50             switch success case 0 { invalid() }
51         }
52         require(success);
53     }
54 
55 
56     /// @return r the product of a point on G1 and a scalar, i.e
    .
57     /// p == p.scalar_mul(1) and p.addition(p) == p.scalar_mul
    (2) for all points p.
58     function scalar_mul(G1Point memory p, uint s) internal view
    returns (G1Point memory r) {
59         uint[3] memory input;
60         input[0] = p.X;
61         input[1] = p.Y;
62         input[2] = s;
63         bool success;
64         assembly {
65             success := staticcall(sub(gas(), 2000), 7, input, 0
    x80, r, 0x60)
66             // Use "invalid" to make gas estimation work
67             switch success case 0 { invalid() }
68         }
69         require (success);
70     }
71     /// @return the result of computing the pairing check
```

```
72     /// e(p1[0], p2[0]) *  .... * e(p1[n], p2[n]) == 1
73     /// For example pairing([P1(), P1().negate()], [P2(), P2()])
        should
74     /// return true.
75     function pairing(G1Point[] memory p1, G2Point[] memory p2)
       internal view returns (bool) {
76         require(p1.length == p2.length);
77         uint elements = p1.length;
78         uint inputSize = elements * 6;
79         uint[] memory input = new uint[](inputSize);
80         for (uint i = 0; i < elements; i++)
81         {
82             input[i * 6 + 0] = p1[i].X;
83             input[i * 6 + 1] = p1[i].Y;
84             input[i * 6 + 2] = p2[i].X[1];
85             input[i * 6 + 3] = p2[i].X[0];
86             input[i * 6 + 4] = p2[i].Y[1];
87             input[i * 6 + 5] = p2[i].Y[0];
88         }
89         uint[1] memory out;
90         bool success;
91         assembly {
92             success := staticcall(sub(gas(), 2000), 8, add(input
       , 0x20), mul(inputSize, 0x20), out, 0x20)
93             // Use "invalid" to make gas estimation work
94             switch success case 0 { invalid() }
95         }
96         require(success);
97         return out[0] != 0;
98     }
99     /// Convenience method for a pairing check for two pairs.
100    function pairingProd2(G1Point memory a1, G2Point memory a2,
       G1Point memory b1, G2Point memory b2) internal view returns (
       bool) {
101        G1Point[] memory p1 = new G1Point[](2);
102        G2Point[] memory p2 = new G2Point[](2);
103        p1[0] = a1;
104        p1[1] = b1;
105        p2[0] = a2;
106        p2[1] = b2;
107        return pairing(p1, p2);
108    }
109    /// Convenience method for a pairing check for three pairs.
110    function pairingProd3(
111            G1Point memory a1, G2Point memory a2,
```

```
112             G1Point memory b1, G2Point memory b2,
113             G1Point memory c1, G2Point memory c2
114     ) internal view returns (bool) {
115         G1Point[] memory p1 = new G1Point[](3);
116         G2Point[] memory p2 = new G2Point[](3);
117         p1[0] = a1;
118         p1[1] = b1;
119         p1[2] = c1;
120         p2[0] = a2;
121         p2[1] = b2;
122         p2[2] = c2;
123         return pairing(p1, p2);
124     }
125     /// Convenience method for a pairing check for four pairs.
126     function pairingProd4(
127             G1Point memory a1, G2Point memory a2,
128             G1Point memory b1, G2Point memory b2,
129             G1Point memory c1, G2Point memory c2,
130             G1Point memory d1, G2Point memory d2
131     ) internal view returns (bool) {
132         G1Point[] memory p1 = new G1Point[](4);
133         G2Point[] memory p2 = new G2Point[](4);
134         p1[0] = a1;
135         p1[1] = b1;
136         p1[2] = c1;
137         p1[3] = d1;
138         p2[0] = a2;
139         p2[1] = b2;
140         p2[2] = c2;
141         p2[3] = d2;
142         return pairing(p1, p2);
143     }
144 }
145
146 contract Verifier {
147     using Pairing for *;
148     struct VerifyingKey {
149         Pairing.G1Point alpha;
150         Pairing.G2Point beta;
151         Pairing.G2Point gamma;
152         Pairing.G2Point delta;
153         Pairing.G1Point[] gamma_abc;
154     }
155     struct Proof {
156         Pairing.G1Point a;
```

```
157         Pairing.G2Point b;
158         Pairing.G1Point c;
159     }
160     function verifyingKey() pure internal returns (VerifyingKey
    memory vk) {
161         vk.alpha = Pairing.G1Point(uint256(0
    x057226b40c33875a2dd6ddea7471741e6108ec04542b7fe9e95de1f52d24
162         17c6), uint256(0
    x152de22a2ae74a9d2a37b2d2f826d5d5221d8027b2b30481a2f42e381ff2
163         eccf));
164         vk.beta = Pairing.G2Point([uint256(0
    x0435e07b0a6dc679170471fbb9683b9cde7124de74285630a63df1ca6aa1
165         baae), uint256(0
    x2d987bceab72cc103d6c2401d87382e245d54f66599f6e191e5a528fa520
166         3ee6)], [uint256(0
    x25c626b588cc36ecddfbf4b0175e10c2fbba36251ed26af6c5c87a04a0e2
167         d584), uint256(0
    x074ff5e751e471118bf644c8f21efcc4f5837493bbbca977df85b7c00fb
168         0b1f3)]);
169         vk.gamma = Pairing.G2Point([uint256(0
    x1d1a265f3bcdd3ecd8e6747b9b13f3013dccc79364993e91228d1c01386
170         b54c2), uint256(0
    x067581d6728b7922f593b484de3bf825ebf4f32e32590da546a0648d6aa7
171         4dc7)], [uint256(0
    x0992ae351f52d5f4020e80de80840be99cf2e7a4906e8218c0b52c6759bc
172         cf8c), uint256(0
    x275582a98356fc3c68b298698ff7a5cd8bee26d726b5376dd6e96a933a7e
173         0ac3)]);
174         vk.delta = Pairing.G2Point([uint256(0
    x0273f9e1db9f6eeb01bb85257360d499d2b29e07ba61536b914db71ebe8d
175         8c6e), uint256(0
    x2fe945ed51390aae859bcfe25c13cb3724e43bde358d0ef892a15c5e48f3
176         8e46)], [uint256(0
    x16760bcd9ea0fa072391e3233b9612ac312ef97c31d4755fff1f8bc992cc
177         77b8), uint256(0
    x2fd85f59a9cf381e93cdd8018b802a9abd9619426edd1230050174dc47e0e

178         73a)]);
179         vk.gamma_abc = new Pairing.G1Point[](3);
180         vk.gamma_abc[0] = Pairing.G1Point(uint256(0
    x1c3ecf4aa6ce1b3297d1ecbf9658ab70901f7a3a3b631e9060c0549971f1
181         7e3b), uint256(0
    x00ee2e23bb43c3de1a7fca2c0aa1b56d289717cc0786be7a62be5319f471
182         4a3c));
```

```
183         vk.gamma_abc[1] = Pairing.G1Point(uint256(0
    x11ff138aa9e20a180157eca3d58d9b4245c1123eb456bf9e2249553c4a4f6

184         37b), uint256(0
    x0d42183d9d8ff63ea60f25efe785bc625b531aafd7a82aab2ec409865a0d8

185         03e));
186         vk.gamma_abc[2] = Pairing.G1Point(uint256(0
    x2884f1c20583c8be447028e43f07c78a62610f6fe945805f61754fa01b28
187         0677), uint256(0
    x1c44a3df521ee0a767d149b35388862d44dab1aa83074c83666086dc0671
188         06dc));
189     }
190     function verify(uint[] memory input, Proof memory proof)
    internal view returns (uint) {
191         uint256 snark_scalar_field =
    21888242871839275222246405745257275088548364400416034343698204
192         186575808495617;
193         VerifyingKey memory vk = verifyingKey();
194         require(input.length + 1 == vk.gamma_abc.length);
195         // Compute the linear combination vk_x
196         Pairing.G1Point memory vk_x = Pairing.G1Point(0, 0);
197         for (uint i = 0; i < input.length; i++) {
198             require(input[i] < snark_scalar_field);
199             vk_x = Pairing.addition(vk_x, Pairing.scalar_mul(vk.
    gamma_abc[i + 1], input[i]));
200         }
201         vk_x = Pairing.addition(vk_x, vk.gamma_abc[0]);
202         if(!Pairing.pairingProd4(
203              proof.a, proof.b,
204              Pairing.negate(vk_x), vk.gamma,
205              Pairing.negate(proof.c), vk.delta,
206              Pairing.negate(vk.alpha), vk.beta)) return 1;
207         return 0;
208     }
209     function verifyTx(
210             Proof memory proof, uint[2] memory input
211         ) public view returns (bool r) {
212         uint[] memory inputValues = new uint[](2);
213 
214         for(uint i = 0; i < input.length; i++){
215             inputValues[i] = input[i];
216         }
217         if (verify(inputValues, proof) == 0) {
218             return true;
```

```
        } else {
            return false;
        }
    }
}
```

Libreria Pairing

La libreria *Pairing* gestisce le operazioni matematiche necessarie:

- **Tipi di Punti (G1Point, G2Point)**: Nel contesto dei ZK-SNARK, si utilizzano due diverse curve ellittiche.
  G1Point rappresenta un punto sulla prima curva (G1), mentre G2Point rappresenta un punto sulla seconda curva (G2).
  Queste due curve hanno proprietà matematiche diverse, ma sono collegate tra loro attraverso una funzione chiamata "pairing".

- **Generatori (P1, P2)**:
  Un generatore è un punto speciale su una curva ellittica. Partendo da questo punto e sommandolo ripetutamente a se stesso, si possono ottenere tutti gli altri punti della curva.
  P1 è il punto generatore della curva G1, mentre P2 è il punto generatore della curva G2. Questi punti sono predefiniti e pubblici.

- **Negate**: Inverte un punto.

- **Addition**: Somma due punti.

- **scalar_mul**: Moltiplica un punto per uno scalare.

- **Pairing**: Verifica una relazione matematica tra punti.

- **Funzioni aggiuntive(pairingProd2, pairingProd3, pairingProd4)**: Servono a semplificare la verifica di coppie di punti.

Pairing di Curve Ellittiche

Il pairing è una funzione matematica che prende in input due punti: uno dalla curva G1 e uno dalla curva G2. Il pairing produce come output un elemento di un terzo gruppo matematico, indicato con GT.

$$G_1 \times G_2 \rightarrow G_T$$

Questo elemento ha proprietà speciali che lo rendono utile per la verifica.
La verifica ZK-SNARK si basa su un'equazione che coinvolge il pairing. Questa equazione ha la forma generale:

$$e(P_1, Q_1) \cdot e(P_2, Q_2) \cdot ... \cdot e(P_n, Q_n) = 1$$

dove:

- e è la funzione di pairing
- $P_1, P_2, ..., P_n$ sono punti sulla curva G1
- $Q_1, Q_2, ..., Q_n$ sono punti sulla curva G2

Contract Verifier

Il contratto Verifier fa uso della libreria *Pairing* per la verifica vera e propria:
Strutture Dati

- **VerifyingKey**: Contiene i parametri pubblici della verifica (chiavi pubbliche generate durante la fase di setup di ZK-SNARK).
- **Proof**: Contiene la prova ZK-SNARK generata da chi vuole dimostrare la conoscenza del segreto, una volta ricevuta in ingresso viene suddivisa nelle 3 componenti a,b,c.

Funzioni

- **verifyingKey()**: Restituisce una chiave di verifica
- **verify()**: Questa funzione è il cuore della verifica. Implementa i controlli crittografici per determinare se la prova è valida rispetto alla chiave di verifica e agli input pubblici.
- **verifyTx()**: Una funzione wrapper che chiama verify e restituisce true se la verifica ha successo, altrimenti false.

Processo di Verifica

1. **Input:** La funzione *verifyTx* riceve una prova e un array di input pubblici .
2. **Verifying Key:** Viene recuperata la chiave di verifica dalla funzione *verifyingKey.*
3. **Verifica ZK-SNARK:** La funzione *verify* esegue i controlli crittografici (pairing di curve ellittiche) utilizzando la prova, la chiave di verifica e gli input pubblici.
4. **Risultato:** La funzione *verifyTx* restituisce **true** se la prova è valida, **false** altrimenti.

5.5 Multi-Verifier

Utilizzando Zokrates un Verifier è in grado di verificare la prova di un singolo documento segreto, questo perchè non esiste una relazione tra documenti precedenti o successivi. Questo implica che non è possibile definire una funzione in grado di distinguere documenti giusti e sbagliati; e quindi non è possibile generare un circuito aritmetico e successivamente un Verifier che possa verificare più documenti segreti.
Non si esclude che utilizzando software a più basso livello si possano trovare soluzioni alternative anche per ottimizzare il sistema.
Questo implica che secondo l'impostazione appena presentata ogni Verifier verifica un solo documento segreto.
Come si può immaginare questo approccio è poco efficente, per queste ragioni in questa sezione si andranno a presentare dei metodi per andare a incrementare la scalabilità e semplicità di utilizzo per i partecipanti della rete.

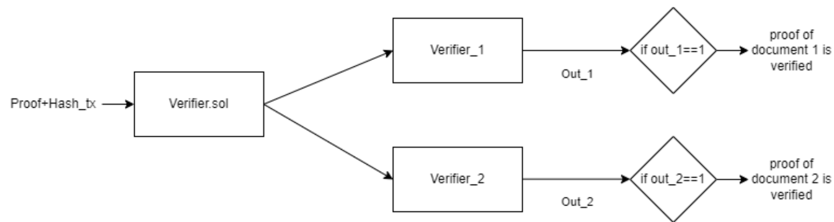

Figura 18: Smart Contract contenente 2 verificatori

come mostrato in figura 18La prima intuizione è quella di inserire all'interno di uno smart contract più verificatori, questo è possibile perchè come mostrato nella sezione 5.4.1 (script Verifier.sol) il "Verifier.sol" utilizza una libreria Pairing per generare quelle funzioni utili a verificare la prova.
Dal punto di vista teorico essendo i 2 script .zok di partenza uguali(e di conseguenza il circuito aritmetico), è possibile usare la stessa libreria per entrambi i verificatori.
riducendo così la dimensione dello smart Contract e il costo di deployment sulla blockchain.
Si pone adesso un problema, un partecipante desisdera verificare una determinata prova di quelle rilasciate da "Azienda A". Per farlo deve conoscere il Public Address del Verifier associato alla prova (si ricorda che 1 prova corri-

sponde a 1 Verifier), per farlo in questa tesi si propongono 2 soluzioni:

- **Lista di Verifier:** On-chain viene salvata una lista di tutti i Verifier generati, essendo pubblica è possibile consultarla per trovare il verificatore corretto.
- **Proxy:** Si genera uno smart contract "Verifier_Proxy" che punta ad ogni gruppo di verificatori, questo semplifica molto il lavoro per una persona che volesse controllare perché non deve eseguire alcuna ricerca, dovrà solamente inserire `hash_tx` + `Proof`. Il Proxy punterà il verificatore corretto (in base al hash_tx) e restituirà True or False (in base alla correttezza della prova).

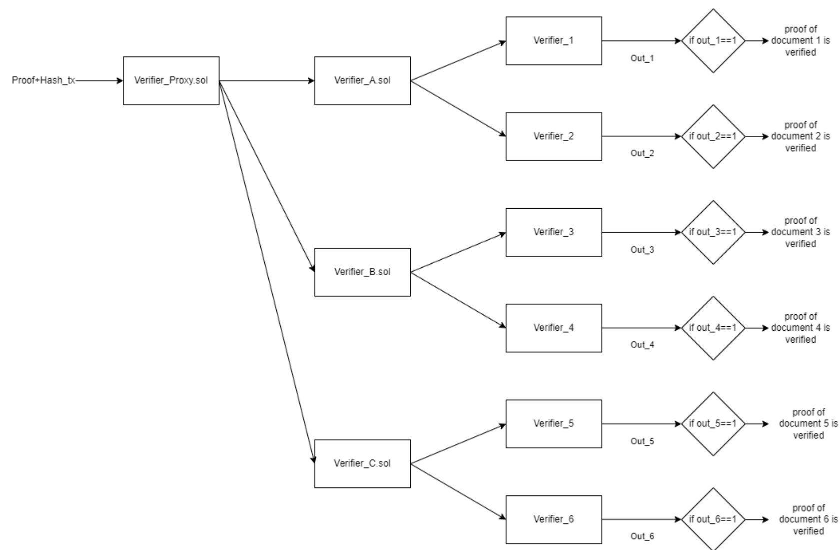

Figura 19: Multi_Verifier+Proxy

5.5.1 Script Proxy.sol

Riguardo la configurazione del Proxy si è tentato di scrivere uno script per implementare il corretto indirizzamento ai vari Verifier

```
pragma solidity ^0.8.0;


contract Proxy {
    struct ContractDetails {
        string name;
        address targetAddress;
        bytes32 hash;
    }


    address public owner;
    mapping(bytes32 => ContractDetails) public contracts;

    constructor() {
        owner = msg.sender;
    }


    modifier onlyowner() {
        require(msg.sender == owner, "solo il proprietario ha i permessi");
        _;
    }


    function addContract(string memory name, address targetAddress, bytes32 hash) public onlyOwner {
        require(contracts[hash].targetAddress == address(0), "Hash gia salvato all'interno della lista");
        contracts[hash] = ContractDetails(name, targetAddress, hash);
    }


    function execute(bytes32 hash, bytes memory data) public {
        ContractDetails memory details = contracts[hash];
        require(details.targetAddress != address(0), "Contratto non salvato all'interno della lista");

```

```
36         (bool success, ) = details.targetAddress.delegatecall(
    data);
37         require(success, "Delegatecall failed");
38     }
39 }
```

- **ContractDetails**: Una struct per memorizzare i dettagli di ciascun contratto. Include il nome del contratto, il suo Address e l'hash_tx.
- **Construct**: Imposta l'indirizzo del creatore del contratto (colui che effettua il deploy) come owner del contratto Proxy.
- **onlyOwner**: permette l'accesso ad alcune funzioni solo al proprietario del contratto. viene usato "require" per verificare se "msg.sender" (l'indirizzo che invoca la funzione) è lo stesso dell'owner. Se la condizione è falsa, la transazione viene rifiutata con un messaggio di errore.
- **addContract**: Permette al proprietario di aggiungere un nuovo contratto al proxy. Questa funzione può essere chiamata solo dall'owner grazie al modificatore onlyOwner.
- **execute**: Consente di invocare una funzione su un contratto target specificato da un hash.
- **delegatecall**: Invia una chiamata al contratto puntato. delegatecall è una funzione che esegue il codice del contratto puntato, permettendo di mantenere il sender originale e l'ammontare di ether inviato.

[12] [13] [14]

5.6 Mapping stringhe di caratteri

Zokrates come detto in precedenza ricevere in input solo Valori numerici e Booleani, se si volesse avere in input delle parole è necessario passare per un Mapping.

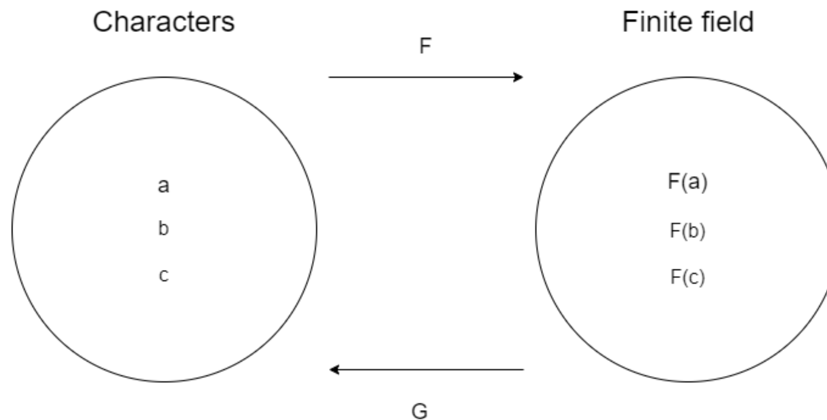

Figura 20: Mapping dei caratteri

Come mostrato in figura 20, è necessario definire una funzione che associa a ogni elemento di A un solo elemento di B, e che da un elemento di B è possibile risalire ad A. E cioè deve essere definita una funzione iniettiva e invertibile:

$$F : A \to B \quad \text{è iniettiva se} \quad F(a_1) = F(a_2) \Rightarrow a_1 = a_2 \quad \text{per ogni} \quad a_1, a_2 \in A.$$

$$F : A \to B \text{ è invertibile se esiste una funzione inversa } G : B \to A$$

Se al posto dei caratteri si volesse accettare in input un hash il discorso è analogo, è necessario eseguire un mapping definendo una funzione iniettiva e invertibile.

5.7 Possibile attacco al protocollo

Nella sezione 5.3 "Applicazione delle ZKP nel contesto della filiera" è stato proposto un sistema utile a mascherare un documento segreto e rilasciare una traccia del caricamento su blockchain, questa traccia viene solitamente chiamata "commitment" e generalmente in molte altre applicazioni è sotto forma di hash.

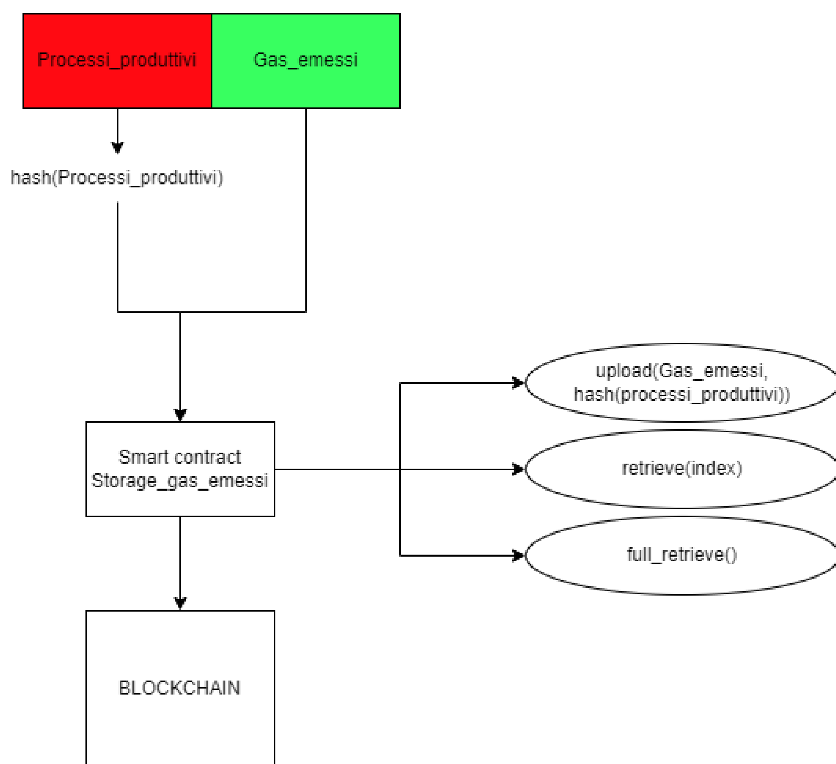

Figura 21: Versione del sistema che usa un hash come commitment

Questo approccio è certamente più efficente ma a differenza di prima (no informazione) fornisce un'informazione all'attaccante e cioè l'hash del documento segreto.
A prima vista può sembrare che non sia un problema, un attaccante non è in grado di risalire al documento segreto dall'hash.
Però quello che può fare è in primo luogo studiare la struttura del documento. E' probabile che sia standardizzato, è plausibile che ci siano parole che

siano più utilizzate rispetto ad altre, il contesto del documento è pubblico e questa è un'ulteriore informazioni utile per l'attacco che si andrà ad esporre.
L'attaccante conosce l'hash del documento segreto ma non il suo contenuto.
Quello che fa è generare una enorme quantita di documenti plausibili restringendo il campo in base agli indizi in suo possesso.
Per ogni documento generato viene eseguito l'hash e si confronta con quello salvato sulla blockchain.
Se l'hash corrisponde, signfica che l'attaccante ha trovato il documento segreto.
Se si tiene conto dell'evoluzione delle reti neurali nell'ultimo decennio e della loro capacità di generare testo, un attacco di questa tipologia è più che teorico e piuttosto pericoloso.

Con ZKP invece non viene fornita alcuna informazione o indizio che possa ricondurre al documento segreto, l'attaccante sia prima che dopo la pubblicazione su blockchain ha la stessa quantità di informazioni.

6 Limiti di Zokrates

6.1 Zokrates Test

Per testare le potenzialità di questo linguaggio e i limiti che comporta usare circuiti aritmetici si è iniziato ad eseguire dei test di prova.
In questa sezione è riportato uno script che genera prove valide solo se viene inserito come input un numero primo, per riconoscere se il valore è divisibile per se stesso e per 1 (condizione per la primalità) viene utilizzato un test di primalità di tipo deterministico, in cui tutti i numeri primi maggiori di 5 hanno la caratteristica di essere multipli di 6 ± 1. In tal modo viene ridotta la quantità di numeri da controllare.

```
1      def main(private u32 num) {
2    u32 mut y = 0;
3      //Verifica delle condizioni iniziali
4      y = if (num == 2 || num == 3) {0} else {1};
5    assert(y==1);
6      y = if (num == 1 || num % 2 == 0 || num % 3 == 0) {0} else
     {1};
7    assert(y==1);
8
9      //Inizio ciclo di verifica con i = 5
10     //u32 mut i = 5;
11     u32 increment = 6;
12   u32 maxnum = 200;
13     u32 mut s=0;
14
15     //Condizione del ciclo: i * i <= num
16     //field non possono usare %
17   //max loop 1048576
18   for u32 mut i in 5..maxnum
19     {
20         y = if (num % i == 0 || num % (i + 2) == 0) {0} else
     {1};
21         s = if ( y==0 ) {1} else {s};
22
23         // y = (i*i > num) {0} else {1};
24         i = i + increment;
25
26     }
27
28     assert(s==1);
29 }
```

6.2 Considerazioni sui test

Inanzitutto è bene specificare che i circuiti aritmetici utilizzati in Zokrates e altri linguaggi simili come Circom, per garantire la sicurezza delle prove devono essere:

- **Completi**: Tutte le possibili esecuzioni del circuito devono essere rappresentate, indipendentemente dai valori di input.
- **Deterministici**: L'output del circuito deve essere univocamente determinato dagli input.

Svolgendo vari test sono state trovate le seguenti limitazioni:

- Operazioni sui parametri "field": Non supportano tutte le operazioni aritmetiche standard (ad esempio, il calcolo del modulo come in $a\%2 == 0$).
- Inizializzazione obbligatoria: Tutti i parametri devono essere inizializzati con un valore predefinito.
- Condizioni nei blocchi if/for/while: Le condizioni che possono modificare il circuito aritmetico devono essere note al momento della generazione del circuito stesso(Esempio: il ciclo for con N_input iterazioni darà errore, il ciclo con 10 iterazioni sarà corretto).
- Utilizzo dell'if statement in Zokrates: Limitato alla modifica del valore dei parametri.
- Cicli for: I cicli for nei circuiti aritmetici vengono "srotolati" (unrolled) dal compilatore. Questo significa che, invece di un ciclo dinamico che si ripete a runtime, il compilatore genera una sequenza lineare di blocchi di codice identici, uno per ogni iterazione del ciclo. Il numero di iterazioni (n) deve essere noto al momento della compilazione.
- Limite massimo di cicli for: 2^{20}.
- Immutabilità del circuito: Una volta creato, il circuito non può essere modificato dinamicamente durante l'esecuzione.
- Divisione: La divisione nel dominio finito restituisce sempre un valore del dominio, se il risultato è di tipo float è necessario passare al datatype u-bit (u32), il valore è approssimato all'intero più vicino.

Se gli *if statement* funzionassero come nei programmi tradizionali, il circuito dovrebbe scegliere un ramo da eseguire in base al valore di una condizione. Questo renderebbe il circuito non deterministico e non completo, poiché non tutte le possibili esecuzioni sarebbero rappresentate.

7 Conclusioni

In conclusione, lo scopo di questo elaborato è stato quello di comprendere le potenzialità della tecnologia ZKP applicata in una filiera produttiva.
È stato spiegato lo stato dell'arte delle prove a conoscenza zero, per poi introdurre un toolbox per generare i blocchi di un sistema N-IZKP (10).
Si sono individuati 3 problemi principali, citati nella sezione 5.1, dove le ZKP possono essere implementate in modo costruttivo.
Come dimostrato nella sezione 5.7, il vantaggio principale delle ZKP è la loro capacità di provare la correttezza di una dichiarazione senza trasmettere nessun'altra informazione, negando all'attaccante la possibilità di sviluppare una qualsiasi forma di attacco e assicurando la sicurezza del protocollo.
Le ZKP, nell'ambito dei registri distribuiti, sono una soluzione computazionalmente più complessa e difficile da scrivere per uno sviluppatore, ma che può portare seri vantaggi in contesti dove la sicurezza è al primo posto.

Ci sono forti motivazioni per l'adozione delle ZKP nelle filiere produttive, principalmente per la loro capacità di non trasmettere informazioni sensibili durante il processo di verifica. I primi esperimenti condotti sembrano promettenti, tuttavia esistono ancora dei limiti significativi da superare, come la capacità di utilizzare un singolo verificatore per un set di documenti distinti, la grandezza dei circuiti aritmetici e i limiti di tali oggetti.
È necessario studiare modi per superare questi limiti, in particolar modo riguardo ZoKrates, e condurre test più significativi su dati reali e su scala maggiore. Un altro step sarebbe eseguire un'analisi dettagliata dei circuiti aritmetici più a basso livello, comprendendo a fondo il meccanismo di sicurezza che sta alla base di zkSNARK.

Le barriere in ingresso per questo contesto non sono banali e la tecnologia nel complesso è ancora in sviluppo, tuttavia è possibile affermare con confidenza che in futuro troverà maggiori applicazioni, specialmente in settori dove la sicurezza e la protezione delle informazioni sono di primaria importanza.

Riferimenti bibliografici

[1] Pejvak Oghazi et al. «RFID and ERP systems in supply chain management». In: *European Journal of Management and Business Economics* 27.2 (2018), pp. 171–182.

[2] Wikipedia contributors. *Bitcoin*. https://it.wikipedia.org/wiki/Bitcoin. 2024.

[3] GeeksforGeeks. *Solidity Ether Units*. https://www.geeksforgeeks.org/solidity-ether-units/?ref=lbp. 2023.

[4] ResearchGate. *How a Smart Contract Works*. https://www.researchgate.net/figure/How-a-smart-contract-works-15_fig3_317349621. 2018.

[5] ChainLink. *Zero Knowledge Proof (ZKP) Education*. https://chain.link/education/zero-knowledge-proof-zkp. 2023.

[6] Ethereum. *Zero Knowledge Proofs*. https://ethereum.org/en/zero-knowledge-proofs. 2024.

[7] Amal Abid et al. «A blockchain-based self-sovereign identity approach for inter-organizational business processes». In: *2022 17th Conference on Computer Science and Intelligence Systems (FedCSIS)*. IEEE. 2022, pp. 685–694.

[8] Jacob Eberhardt. «Scalable and privacy-preserving off-chain computations». In: (2021).

[9] Wikipedia. *Commitment Scheme — Wikipedia*. https://en.wikipedia.org/wiki/Commitment_scheme. 2024.

[10] Vitalik Buterin. «Exploring Elliptic Curve Pairings». In: (2017). https://medium.com/@VitalikButerin/exploring-elliptic-curve-pairings-c73c1864e627.

[11] Youssef El Housni e Aurore Guillevic. «Families of SNARK-friendly 2-chains of elliptic curves». In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2022, pp. 367–396.

[12] Solidity by Example. *App Upgradeable Proxy*. https://solidity-by-example.org/app/upgradeable-proxy.

[13] Talentica. *Implementing Upgradeable Smart Contracts Using Proxy Patterns.* `https://www.talentica.com/blogs/implementing-upgradeable-smart-contracts-using-proxy-patterns`. 2024.

[14] Cloudflare. *A Relatively Easy to Understand Primer on Elliptic Curve Cryptography.* `https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography`. 2021.

Ringraziamenti

Ebbene si, siamo giunti al termine di questo lungo viaggio, con questa ultima sezione desidero ringraziare tutte le persone che mi hanno accompagnato in questo percorso.

Un ringraziamento speciale va ai prof Luca Spalazzi e Massimiliano Pirani che attraverso la loro competenza e esperienza mi hanno dato una mano nella realizzazione di questo elaborato.
La loro dedizione ha mostrato che oltre ad essere dei gran professori sono anche delle grandi persone.

Ringrazio i miei genitori la "mami" e il "papi", che in tutti questi anni mi hanno incoraggiato a dare sempre il meglio, grazie per non avermi fatto mancare nulla, grazie per l'affetto che mi date quotidianamente.
Se ce l'ho fatta è in gran parte merito vostro.

Alla mia sorellina bellissima per aver creduto nelle mie potenzialità e avermi sempre dato una mano nel momento del bisogno.

Anche questa volta ringrazio il mio ex Prof di elettronica per avermi introdotto il metodo "pane e salsiccia" che in questo viaggio mi ha numerose volte guidato verso il successo. Ringrazio tutti i miei amici per la presenza e il sostegno.

Grazie a tutti finalmente ho raggiunto il traguardo!