Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Rollup Ottimistici e di Validità: Confronto e Analisi tra Optimism e StarkNet

Relatore:
Chiar.mo Prof.
Cosimo Laneve

Presentata da:
Luca Donno

Sessione II
Anno Accademico 2021/2022

Indice

Capitolo 1

Introduzione

Una blockchain è una struttura dati distribuita formata da una lista di blocchi concatenati [1]. Un blocco è formato da una sequenza ordinata di transazioni utilizzata in una transizione di stato e da una testata che contiene informazioni come il numero del blocco, l'hash della testata del blocco precedente e la radice dell'albero di Merkle rappresentante le transazioni.

Un albero di Merkle [64] è una struttura dati dove ogni foglia è costituita dall'hash di un blocco di dati e ogni nodo interno viene calcolato come l'hash dei suoi figli. In questo modo, la radice rappresenta in maniera succinta i dati sottostanti e l'inclusione di una foglia può essere dimostrata con un numero di passi e una dimensione della prova proporzionale all'altezza dell'albero, ovvero $O(\log n)$. Ethereum utilizza una variante dell'albero di Merkle Patricia [66] per ottimizzarne le prove e le ricerche, ma ai fini di questo scritto è sufficiente pensarli come alberi di Merkle.

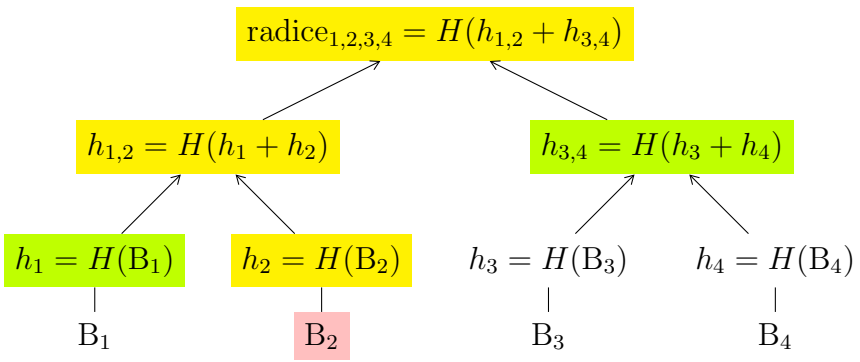


Figura 1.1: Un albero di Merkle. Per dimostrare l'inclusione nella radice del blocco B_2 è sufficiente fornire un numero logaritmico di nodi (in verde) rispetto al numero di blocchi, mentre per la verifica è sufficiente calcolare un numero logaritmico di hash (in giallo).

[1] raramente questa struttura viene rappresentata come un grafo aciclico direzionato, come nella blockchain Nano (XNO).

Nelle blockchain più semplici, come Bitcoin, le transizioni consistono nell'aggiornare le assegnazioni della criptovaluta agli utenti, mentre in blockchain che integrano una macchina virtuale, come Ethereum [2], esse possono interagire con programmi arbitrari (chiamati *smart contract*) e modificarne la memoria (detta *storage*).

Ogni nodo della rete contiene una replica dello stato corrente della blockchain a cui vengono aggiunti blocchi ad intervalli regolari. Nonostante la loro struttura decentralizzata, esse sono logicamente centralizzate in quanto ogni nodo deve osservare la stessa versione dei fatti. Siccome ognuno può aggiungere un blocco in testa alla blockchain, più versioni della nuova lista vengono proposte alla rete e ognuna di queste prende il nome di *fork*. Per questo motivo, le blockchain implementano un algoritmo di consenso che permette di decidere quali delle possibili versioni dello stato seguire e a cui aggiungere nuove transazioni, chiamato *fork choice rule*. Le due categorie di algoritmi di consenso più frequentemente usate sono *Proof of Work* e *Proof of Stake*.

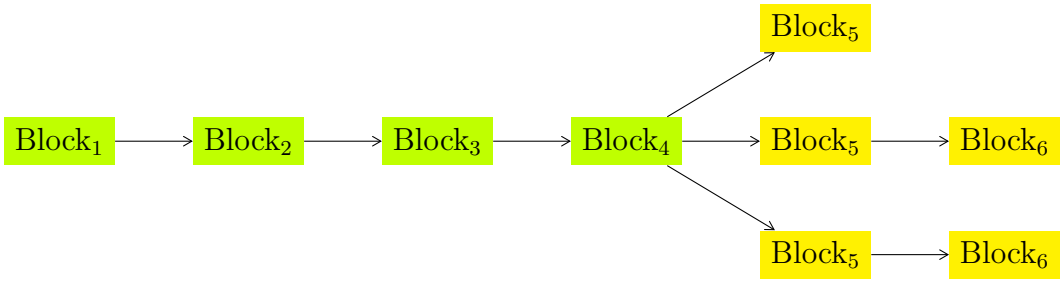


Figura 1.2: Una blockchain con tre fork dal blocco 4. In verde i blocchi su cui tutta la rete ha raggiunto il consenso, in giallo gli altri.

I nodi, oltre a seguire la regola di consenso, verificano che le transazioni in un blocco rispettano delle regole di validità: ad esempio, un utente non può spendere più monete di quelle che possiede, non può effettuare transazioni utilizzando il conto di un altro utente, un blocco non può affermare che il risultato di $2 + 2$ non sia 4.

Un nodo della rete è in grado di verificare in maniera indipendente queste regole, che hanno priorità sulle regole di consenso: se un blocco viene scelto dalla regola di consenso ma contiene transazioni invalide, esso viene scartato. I nodi che verificano sia la validità dei blocchi che la regola di consenso prendono il nome di *full node*. Utilizzare un proprio full node per interagire con una blockchain è fondamentale per non affidarsi ad alcun intermediario, che potrebbe mostrare una versione maliziosa del suo stato.

La portata di una blockchain, ovvero il numero di transazioni per secondo (TPS), è proporzionale alla dimensione dei blocchi, che determina il numero di transazioni che un blocco può contenere, e alla loro frequenza nel tempo. Siccome lo spazio in un blocco è finito, il costo per includere una transazione è una funzione della domanda e dell'offerta di esso e per questo motivo si desidera avere una portata maggiore.

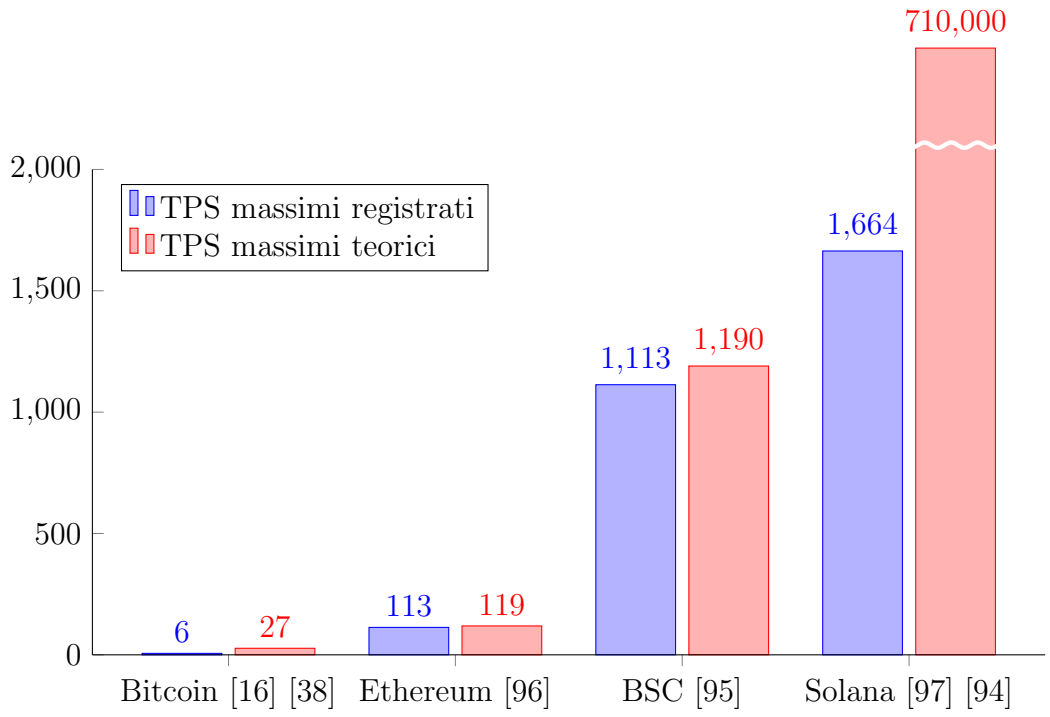
[2] la macchine virtuale di Ethereum prende il nome di Ethereum Virtual Machine (EVM).

La dimensione di un blocco può essere definita in byte o in *gas*. Il gas rappresenta un'unità di misura del costo, in termini di risorse utilizzate, di una computazione. Ad esempio, un trasferimento semplice di Ether su Ethereum costa 21000 gas.

Opcode	**Gas**
ADD	3
XOR	3
MUL	5
MULMOD	8
JUMP	8
BLOCKHASH	20

Tabella 1.1: Alcuni opcode della EVM e i rispettivi costi in gas.

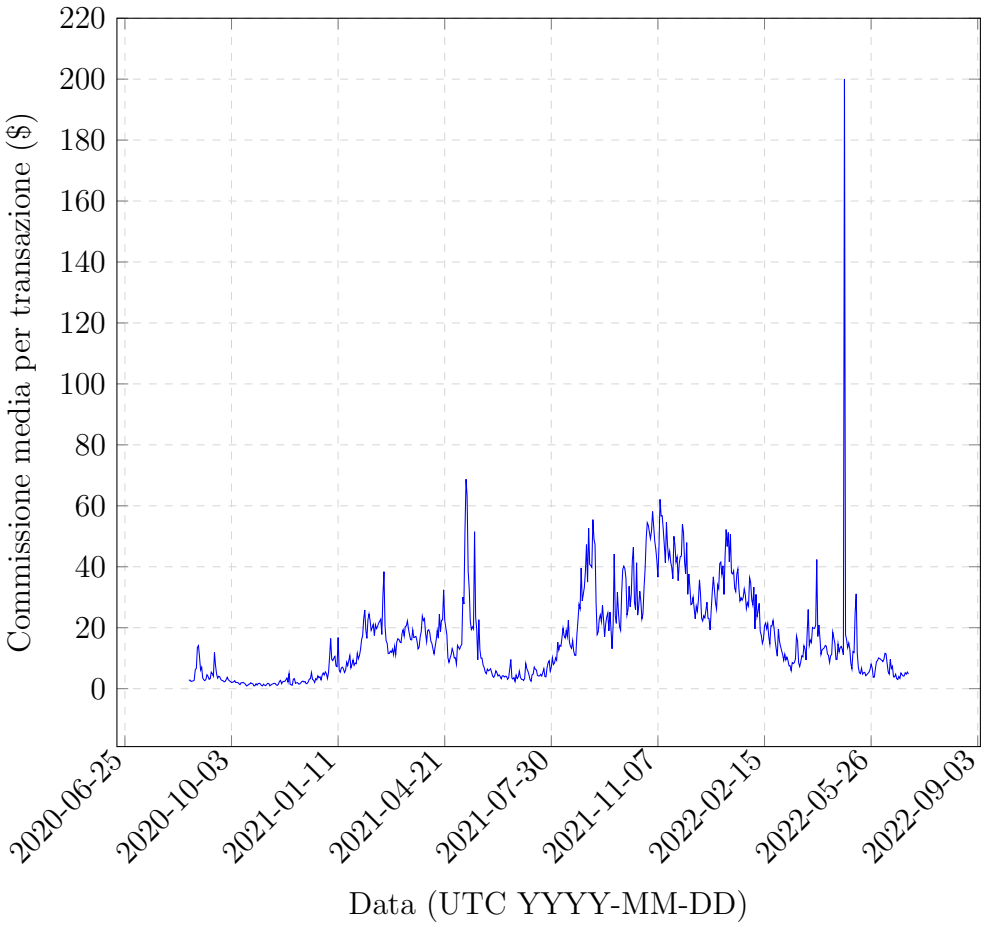
Diverse blockchain hanno diverse prestazioni:



1.1 La crisi del gas

A causa dell'elevata utilizzazione della rete [32] e della limitata portata, Ethereum ha mostrato come il prezzo del gas può aumentare di due ordini di grandezza in un solo giorno [31] a causa di singoli eventi. Ad esempio, durante il rilascio del token UNI del

protocollo Uniswap [58] il prezzo del gas ha raggiunto un prezzo medio di circa 538 Gwei [3], con un massimo di 133053 Gwei. Un semplice scambio di token a 488 Gwei [50] costa circa 0.055 ETH in commissioni, che al prezzo di 1800$ corrisponde a 99$. Più recentemente, il 1 Maggio 2022, durante il rilascio della collezione NFT Otherside [59], alcuni utenti hanno speso più di 4000$ in commissioni per ottenere un paio di NFT [93], portando la commissione media per transazione a 200$, la più alta di sempre [30]. Vitalik Buterin, l'ideatore di Ethereum, in una famosa intervista del 2017 [18] affermò che «l'Internet delle monete non deve costare più di 5 centesimi[4] per transazione».

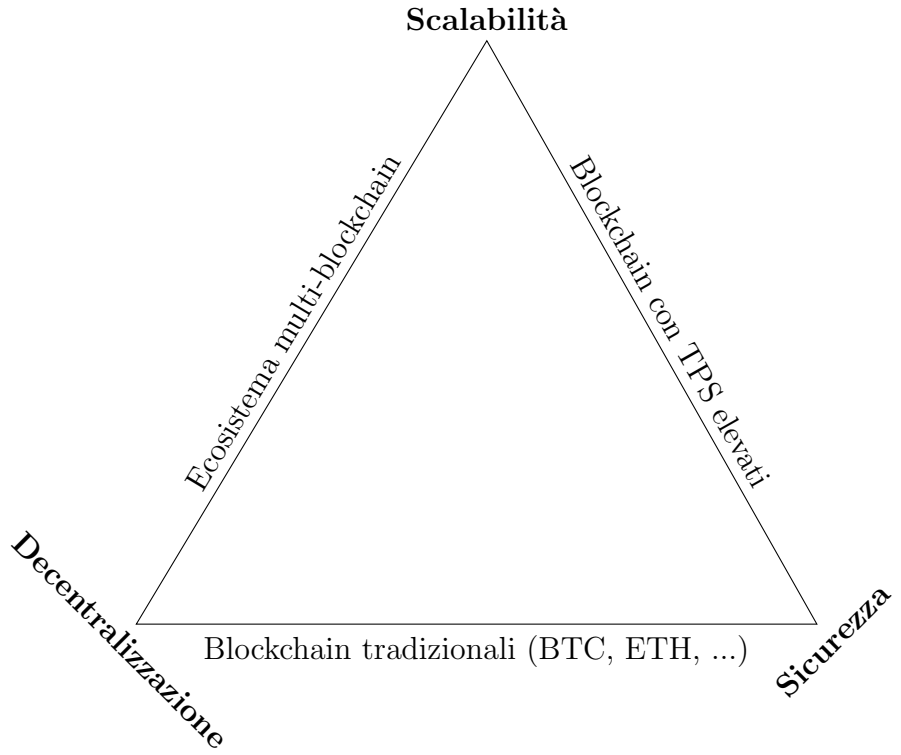


1.2 Scalabilità decentralizzata

Per aumentare la portata di una blockchain, una soluzione banale è aumentare la dimensione dei blocchi (o la loro frequenza). Nel contesto di Ethereum, questo significa

[3] 1 Gwei equivale a 10^9 wei. 10^{18} wei = 1 ETH, quindi un miliardo di Gwei equivale a 1 Ether.
[4] di dollaro, N.d.T.

aumentare la quantità massima di gas che un blocco può contenere. Con questo metodo, in quanto ogni full node deve elaborare ogni transazione di ogni blocco per verificarne la validità, all'aumentare della portata aumentano anche i requisiti hardware, portando di conseguenza ad una maggiore centralizzazione della rete. Alcune blockchain, come Bitcoin ed Ethereum, ottimizzano il proprio design per massimizzare la decentralizzazione architetturale, altre, come la Binance Smart Chain (BSC) e Solana, per essere il più veloci ed economiche possibili. Le reti decentralizzate limitano artificialmente la portata della blockchain per diminuire i requisiti hardware per partecipare nella rete. Questo trade-off prende il nome di Trilemma della Scalabilità [65] [45]:



Il Trilemma afferma che non esiste una tecnica semplice per ottenere tutte e tre le proprietà, ma soltanto due di esse. Le proprietà sono:

- **Scalabilità**: la rete processa più transazioni di quelle che un semplice nodo (ad esempio un comune laptop) è in grado di verificare.
- **Decentralizzazione**: la rete opera senza l'affidamento di un numero ristretto di grandi operatori centralizzati.

- **Sicurezza**: la rete è in grado di resistere ad una grande percentuale di nodi ostili.

Le blockchain tradizionali puntano ad avere decine o centinaia di milioni di nodi indipendenti a discapito della scalabilità, mentre le blockchain ad alte prestazioni di solito non superano il centinaio di nodi, a causa degli elevati requisiti hardware. Solana richiede almeno una CPU a 16 core e 256GB di memoria RAM per un nodo RPC [35]. Un ecosistema multi-chain consiste nell'avere diverse applicazioni su diverse blockchain che comunicano tra di loro: questo sistema è decentralizzato e scalabile, ma permette ad un attaccante di prendere il controllo di soltanto di una di queste per causare effetti a catena sulle altre reti.

1.3 Soluzioni per la scalabilità

Nel corso degli anni si è cercato di trovare una soluzione per risolvere il Trilemma. Tutte queste soluzioni hanno la caratteristica di spostare qualche attività off-chain, collegare l'attività on-chain a quella off-chain utilizzando degli smart contract e di verificare on-chain ciò che accade off-chain. Le tre principali soluzioni di scalabilità sono i canali di stato, Plasma e i Rollup.

1.3.1 Canali di stato

I canali di stato [68] sono correlati ai canali di pagamento di Bitcoin [63] [6] in quanto permettono pagamenti istantanei senza commissioni, ma per operazioni che alterano lo stato della macchina virtuale. I componenti di un canale di stato sono:

1. Uno smart contract che definisce e blocca lo stato iniziale con cui interagire.
2. Dei partecipanti che aggiornano lo stato tra di loro costruendo e firmando transazioni che potrebbero essere pubblicate on-chain ma non lo sono in prima battuta. Ogni alterazione di stato prende il posto di quella precedente.
3. Un meccanismo per chiudere il canale che sblocca lo stato.

Il punto chiave è che il passo 2 non ha bisogno di interagire per nulla con la blockchain, ma deve essere garantito ad ogni partecipante di poter aggiornare lo stato in qualunque momento. La limitazione di questo approccio è che i partecipanti, o qualcuno per loro, devono rimanere online finché il canale non viene chiuso. Ciò non rappresenta un problema per quelle applicazioni che richiedono già l'essere sempre online, come ad esempio i giochi a turni.

Esempio 1: una partita a Tris

Lo smart contract è inizializzato con un campo da Tris 3x3 inizialmente vuoto e l'indirizzo dei due sfidanti. Alice inizia la partita scegliendo dove posizionare il proprio segno, firma il messaggio contenente la matrice aggiornata e la invia a Bob che la firma a sua volta. Il messaggio non viene pubblicato on-chain: è dovere dei partecipanti, o qualcuno per loro, tenerne traccia. Il turno passa a Bob, che sceglie la posizione del proprio simbolo, aggiorna la matrice, la firma e la invia ad Alice. Alice può però rifiutarsi di firmare il messaggio se non gradisce la scelta di Bob: deve perciò esistere un meccanismo per evitare questa situazione. Bob pubblica lo stato precedente sullo smart contract, che ne verifica la validità e le firme, ed esegue la sua mossa corrente on-chain. Se Alice si rifiuta di continuare a giocare (o perde la connessione), un time-out può essere implementato che assegna eventualmente la vittoria a Bob. Se invece entrambi i giocatori sono disposti a giocare fino al termine della partita, l'unico stato che si pubblica sullo smart contract è quello finale, dove una funzione ne verifica la configurazione e assegna il premio al vincitore.

I canali di stato non sono generali in quanto l'implementazione dipende dall'applicazione specifica. Inoltre non sono adatti per quelle situazioni in cui gli oggetti interessati o le azioni da eseguire non hanno associato un partecipante esplicito.

1.3.2 Plasma

Una Plasma chain [75] è una blockchain autonoma che viene ancorata ad Ethereum ed esegue le transazioni off-chain. Essa viene gestita tramite uno smart contract sulla blockchain principale che gestisce i depositi e i prelievi. Lo stato del Plasma viene pubblicato periodicamente su Ethereum sotto forma di radice di Merkle.

Per accedere, gli utenti depositano una quantità di token (ETH o qualsiasi token ERC-20 [89]), che vengono ricreati in pari quantità sulla Plasma chain.

Per ritirare i fondi su Ethereum il meccanismo è più complicato, in quanto Ethereum non è in grado di verificare, avendo soltanto la radice di Merkle, che lo stato del Plasma sia corretto. Un utente malizioso potrebbe creare uno stato fittizio che afferma che possiede 1000 ETH, e pubblicare una prova di Merkle di inclusione nella radice di Merkle. Per questo motivo si implementa un periodo di disputa, solitamente di 7 giorni, in cui ognuno può dimostrare che la prova di Merkle sia invalida, invalidando il prelievo. Il corretto comportamento può essere incentivato facendo depositare una quantità di ETH prima del ritiro, che viene sottratta in caso di comportamento invalido.

Le due implementazioni di Plasma più utilizzate sono Plasma MVP [49] e Plasma Cash [51].

Esempio 2: Doppia spesa

Alice utilizza Plasma MVP per effettuare le proprie transazioni, utilizzando un modello UTXO [a]. Alice invia dei token a Bob creando una transazione utilizzando un UTXO. Alice successivamente vuole utilizzare lo stesso UTXO, che è ora di Bob, per ritirare i fondi. Bob dimostra, sul contratto rappresentante la Plasma chain su Ethereum, che Alice ha già speso l'UTXO in una transazione precedente. Se Bob è invece offline, Alice ha successo nell'effettuare una doppia spesa [22].

[a] Unspent Transaction Output [25]

Plasma, come i canali di stato, non è una soluzione di scalabilità generale in quanto non supportano l'utilizzo di smart contract [9].

La limitazione principale nasce dal Problema della Disponibilità dei Dati [5]: se un operatore pubblica una transizione di stato invalida, gli utenti non possono creare dimostrazioni d'invalidità se esso non pubblica i dati corrispondenti alle radici di Merkle.

1.3.3 Rollup

I Rollup sono blockchain che pubblicano i propri blocchi su un'altra blockchain, rispettivamente chiamate Layer 2 (L2) e Layer 1 (L1), ereditandone interamente il consenso e la disponibilità dei dati [62].

I Rollup hanno tre componenti principali:

- **Sequenziatori**: nodi che ricevono transazioni del Rollup dagli utenti e le combinano in un blocco che inviano al Layer 1. Il blocco è formato almeno dalla radice dello stato (ad esempio come radice di Merkle) e i dati relativi alle transazioni, risolvendo il Problema della Disponibilità dei Dati.

- **Full Node del Rollup**: nodi che ottengono i blocchi dal Layer 1, processano e validano tutti i dati relativi alle transazioni verificando che la radice sia corretta. Se un blocco contiene transazioni invalide viene scartato. In questo modo, i Sequenziatori non possono creare blocchi validi contenenti transazioni invalide.

- **Light Node del Rollup**: nodi che ottengono i blocchi dal Layer 1 ad eccezione delle transazioni. Essi non possono calcolare da sé il nuovo stato, ma verificano che esso sia valido utilizzando tecniche come le prove d'invalidità o di validità.

I Rollup utilizzano l'ordinamento dei blocchi del Layer 1 per determinare la testa della blockchain. Un blocco si dice finalizzato se è il primo blocco valido alla sua altezza. Essi,

[5] in inglese Data Availability Problem (DAP).

a differenza delle altre soluzioni, supportano computazione arbitraria. Essi sono scalabili perché il costo ammortizzato delle transazioni decresce all'aumentare degli utenti, in quanto il costo di assicurare la validità della blockchain cresce sub-linearmente rispetto al costo di verificare le transazioni singolarmente.

I Rollup si distinguono in base al meccanismo con cui assicurano la validità dell'esecuzione delle transazioni ai light node: negli Rollup Ottimistici viene garantita tramite un modello economico e prove d'invalidità, mentre nei Rollup di Validità viene garantita matematicamente da prove di validità.

Siccome l'esecuzione delle transazioni avviene non più sulla blockchain base ma off-chain, si è iniziato a pensare alle blockchain non più come strutture monolitiche ma modulari. I light node possono essere implementati come smart contract sul Layer 1: essi accettano la radice del nuovo stato e verificano le prove di validità o invalidità. Questi Rollup prendono il nome di Smart Contract Rollup, se invece i nodi sono indipendenti, vengono chiamati Rollup Sovrani. Il vantaggio di usare uno Smart Contract Rollup è il poter costruire un ponte tra le due blockchain: siccome la validità dello stato di L2 viene dimostrata a L1, è possibile implementare un sistema di transazioni da L2 a L1, permettendo i prelievi. Lo svantaggio è che il costo delle transazioni dipende così dal costo della verifica dello stato su L1: se il livello base è saturato da altre attività anche il costo delle transazioni sul Rollup aumenta; costo che viene evitato utilizzando invece un Rollup Sovrano. Il livello dei dati e del consenso sono quelli che determinano la sicurezza del sistema in quanto definiscono l'ordinamento delle transazioni, prevengono attacchi e rendono disponibili i dati per dimostrare la validità dello stato. In alcune costruzioni dette Validium i dati possono essere pubblicati su una rete differente da quella dove avviene il consenso, ma in questo caso vengono introdotte nuove assunzioni di sicurezza e per questo non rientrano a far parte dei Rollup.

<table>
<tr><th></th><th>Monolitico</th><th>Smart Con. Rollup</th><th>Rollup Sovrano</th><th>Validium</th></tr>
<tr><td>Dati</td><td rowspan="4">Ethereum</td><td rowspan="3">Ethereum</td><td rowspan="2">Ethereum</td><td>Off-chain</td></tr>
<tr><td>Consenso</td><td rowspan="3">Ethereum</td></tr>
<tr><td>Verifica</td><td rowspan="2">Rollup Sovrano</td></tr>
<tr><td>Esecuzione</td><td>Smart Con. Rollup</td><td>Validium</td></tr>
</table>

Tabella 1.2: Tipi di Rollup possibili su Ethereum. I Validium non vengono considerati Rollup perché aggiungono delle assunzioni di sicurezza. Se esso permette all'utente di scegliere se pubblicare i dati on-chain o off-chain, la soluzione prende il nome di Volition [83].

1.4 Struttura della tesi

Nel secondo capitolo vengono introdotti i Rollup Ottimistici e in particolare viene discusso Optimism Bedrock entrando nel dettaglio del funzionamento dei depositi, del sequenziamento, dei prelievi e del sistema di prove d'invalidità.

Nel terzo capitolo vengono introdotti i Rollup di Validità insieme alla presentazione delle diverse tecniche di dimostrazione studiate nell'ambito della crittografia computazionale: dimostrazioni probabilistiche, dimostrazioni interattive, dimostrazioni a conoscenza zero, prove SNARK e STARK. In seguito viene presentato StarkNet utilizzando una struttura simile a quella di Optimism Bedrock.

Il quarto capitolo è un confronto delle due tecnologie: viene svolta una analisi delle motivazioni dietro la differenza nei tempi di prelievo, della possibilità di applicare la tecnologia ricorsivamente (L3 e oltre), del costo delle transazioni con delle soluzioni ad hoc per minimizzarlo, della compatibilità con Ethereum e delle licenze utilizzate.

Capitolo 2

Rollup Ottimistici

I Rollup ottimistici sono attualmente la soluzione di scalabilità che racchiude più valore. Nella data 2 settembre 2022, Arbitrum [56] e Optimism [88], i due Rollup ottimistici più utilizzati, racchiudono complessivamente 4.2 miliardi di dollari, circa l'80.3% del valore racchiuso nella totalità dei Rollup [54]. Entrambi questi Rollup basano il loro design sulla EVM, scelta che ha permesso di vedere un forte sviluppo a causa della compatibilità con gli strumenti esistenti per Ethereum e l'uso di Solidity. Fuel [55] è attualmente l'unico tentativo di costruire un Rollup Ottimistico differente dalla EVM: esso sfrutta un modello basato su UTXO che rende possibile elaborare transazioni in parallelo e utilizza Sway per la scrittura di smart contract, un linguaggio ispirato a Rust.

2.1 Preliminari

L'idea di accettare *ottimisticamente* l'output dei blocchi senza verificarne l'esecuzione è già presente nel whitepaper di Bitcoin [67], discutendo dei light node. Questi nodi seguono soltanto la catena delle testate verificando la regola di consenso, rendendoli vulnerabili ad accettare blocchi contenenti transazioni invalide in caso di un attacco 51%, dove invece un full node li rifiuterebbe. Nakamoto propone di risolvere questo problema utilizzando un sistema di "allarme" per avvertire i light node che un blocco contiene transazioni invalide. Questo meccanismo viene implementato per la prima volta da Al-Bassam, Sonnino e Buterin [11] in cui si utilizza un sistema di prove d'invalidità basato sui codici di correzione degli errori [28]. Per permettere la creazione di prove d'invalidità è necessario che i dati di tutti i blocchi, anche quelli invalidi, siano disponibili alla rete: questo è il Problema della Disponibilità dei Dati, che viene risolto utilizzando un meccanismo probabilistico di campionamento dei dati.

Il primo design di Rollup Ottimistico è stato presentato da John Adler e Mikerah Quintyne-Collins nel 2019 [1], in cui i blocchi vengono pubblicati su un'altra blockchain che ne definisce il consenso e l'ordinamento.

2.2 Optimism Bedrock

Bedrock [88] è versione più recente di Optimism. La versione precedente, la OVM (Optimistic Virtual Machine), richiedeva un compilatore ad hoc per compilare Solidity nel proprio bytecode: in contrasto, Bedrock è completamente equivalente alla EVM in quanto il motore di esecuzione segue le specifiche dell'Ethereum Yellow Paper [92].

2.2.1 Panoramica

Il dati di chiamata (detti *calldata*) di ogni transazione che avviene su L2 vengono pubblicati su L1 dai Sequenziatori sotto forma di batch. Gli utenti possono depositare transazioni sul Layer 2 utilizzando uno smart contract su L1: questo meccanismo viene principalmente utilizzato per spostare asset come token e NFT. Un nodo del Rollup può ricostruire interamente la catena del rollup leggendo i batch e le transazioni depositate su L1. Le transazioni vengono incluse in questa blockchain solo se valide, esattamente nello stesso modo in cui avviene per il Layer 1. Per poter permette l'invio di transazioni da L2 a L1, il Layer 1 ha bisogno di conoscere lo stato del Layer 2. Le radici di Merkle rappresentanti lo stato del Layer 2 e dell'insieme dei prelievi viene così salvato su Layer 1. La transazione viene inviata su un contratto del Layer 2, che viene finalizzata fornendo una prova di Merkle di inclusione nella radice. Ciò può avvenire soltanto al termine di un "periodo di disputa", durante il quale viene dato la possibilità di dimostrare l'invalidità delle radici utilizzando Cannon, il sistema di prove d'invalidità di Optimism.

2.2.2 Depositi

Gli utenti possono depositare transazioni tramite un contratto su Ethereum, l'Optimism Portal, chiamando la funzione `depositTransaction`.

```
function depositTransaction(
        address _to,
        uint256 _value,
        uint64 _gasLimit,
        bool _isCreation,
        bytes memory _data
    ) public payable metered(_gasLimit) {
        // Just to be safe, make sure that people specify address(0) as
            the target when doing
        // contract creations.
        if (_isCreation) {
            require(
                _to == address(0),
                "OptimismPortal: must send to address(0) when creating
                    a contract"
            );
```

```
        }

        // Transform the from-address to its alias if the caller is a
            contract.
        address from = msg.sender;
        if (msg.sender != tx.origin) {
            from = AddressAliasHelper.applyL1ToL2Alias(msg.sender);
        }

        bytes memory opaqueData = abi.encodePacked(
            msg.value,
            _value,
            _gasLimit,
            _isCreation,
            _data
        );

        // Emit a TransactionDeposited event so that the rollup node
            can derive a deposit
        // transaction for this deposit.
        emit TransactionDeposited(from, _to, DEPOSIT_VERSION,
            opaqueData);
    }
```

Quando una transazione viene eseguita viene emesso un evento `TransactionDeposited`, che ogni nodo del rollup ascolta per processare i depositi. Una transazione depositata è una transazione del L2 che viene derivata dal L1. Se il chiamante della funzione è un contratto (verificato con il confronto tra `msg.sender` e `tx.origin`), l'indirizzo viene trasformato aggiungendo `0x1111000000000000000000000000000000001111` ad esso. Ciò previene gli attacchi in cui un contratto su L1 ha lo stesso indirizzo di un contratto su L2, ma un codice diverso. Questo può avvenire quando vengono creati contratti usando l'opcode `CREATE`, che calcola l'indirizzo del contratto come `keccak256(senderAddress, nonce)`.

L'inclusione su L2 delle transazioni depositate viene garantita da specifica entro una *finestra di sequenziamento*.

Le transazioni depositate sono un nuovo tipo di transazione compatibile con EIP-2718 [98] con prefisso `0x7E`, in cui i campi codificati in rlp sono:

- `bytes32 sourceHash`: hash che identifica univocamente l'origine della transazione.
- `address from`: l'indirizzo del sender.
- `address to`: l'indirizzo del destinatario, o l'indirizzo zero se la transazione depositata è una creazione di contratto.
- `uint256 mint`: il valore in ETH da essere creato su L2.
- `uint256 value`: il valore in ETH da inviare al destinatario.

- `bytes data`: i dati di input.
- `bytes gasLimit`: il limite del gas della transazione.

Il `sourceHash` viene calcolato come l'hash keccak256 di `bytes32(uint256(0))`, `keccak256(l1BlockHash)` e `bytes32(uint256(l1LogIndex))` dove `l1BlockHash` e `l1LogIndex` identificano univocamente un evento `TransactionDeposited` in un blocco. Senza un `sourceHash` due diverse transazioni depositate possono avere lo stesso hash.

Una transazione depositata viene eseguita nei seguenti passi:

- il bilancio di `from` viene aumentato di `mint`.
- `CALLER` e `ORIGIN` vengono impostati a `from`.
- `context.calldata` viene impostato a `data`.
- `context.gas` viene impostato a `gasLimit`.
- `context.value` viene impostato a `value`.
- il nonce di `from` viene incrementato di 1.

Mercato del gas

Siccome le transazioni depositate vengono iniziate su L1 ma eseguite su L2, il sistema necessita di un meccanismo per pagare su L1 il gas speso su L2. Una soluzione è quella di inviare ETH tramite il Portal, ma questo implica che ogni chiamante (anche indiretto) deve essere marcato come `payable`, e questo non è possibile per molti progetti esistenti. L'alternativa è quella di bruciare il gas corrispondente su L1. Il gas g assegnato alle transazioni depositate viene chiamato *gas garantito*. Il prezzo del gas di L2 su L1 non viene automaticamente sincronizzato con l'effettivo di L2 ma viene stimato usando un meccanismo simile a EIP-1559 [20], aprendo la possibilità ad arbitraggio tra i due. La quantità massima di gas garantito per ogni blocco di Ethereum è 8 milioni, con un target di 2 milioni.

La quantità c di ETH richiesta per pagare per il gas su L2 è $c = g * b_{\text{L2}}$ dove b_{L2} è la basefee su L2. Il contratto su L1 brucia una quantità di gas pari a c/b_{L1}. Il gas speso per chiamare `depositTransaction` viene rimborsato su L2: se questa quantità è maggiore del gas garantito, nessun gas viene bruciato.

Deposito degli attributi di L1

La prima transazione di un blocco del rollup è una *transazione depositata degli attributi di L1*, usata per registrare su un predeploy di L2 gli attributi dei blocchi di Ethereum. L'address del predeploy è `0x4200000000000000000000000000000000000015` e l'account

che esegue i depositi è `0xdeaddeaddeaddeaddeaddeaddeaddeaddead0001`, un EOA di cui non si conosce la chiave privata e che viene ritornato dagli opcode `CALLER` e `ORIGIN` durante l'esecuzione di una transazione depositata degli attributi di L1.

Gli attributi che il predeploy dà accesso sono:

- Il numero del blocco di L1.
- Il timestamp del blocco di L1.
- La basefee del blocco di L1.
- L'hash del blocco di L1.
- Il numero di sequenza, ovvero il numero del blocco di L2 relativo al blocco L1 associato (chiamata anche *epoca*). Questo numero viene azzerato quando inizia una nuova epoca.

2.2.3 Sequenziamento

I nodi del rollup derivano la chain di Optimism interamente da Ethereum. Questa chain viene estesa ogni volta che vengono pubblicate nuove transazioni su L1 e i suoi blocchi vengono riorganizzati ogni volta che i blocchi di Ethereum vengono riorganizzati.

La blockchain del rollup è divisa in epoche. Per ogni numero di blocco n di Ethereum, esiste una corrispondente epoca n. Ogni epoca contiene almeno un blocco e ogni blocco in un'epoca contiene una transazione depositata degli attributi di L1. Il primo blocco di un'epoca contiene tutte le transazioni depositate tramite il Portal. I blocchi del Layer 2 posso contenere anche delle *transazioni sequenziate*, ovvero transazioni inviate direttamente al Sequenziatore.

Sequenziatore

Il Sequenziatore accetta transazioni dagli utenti e costruisce i blocchi. Per ogni blocco, costruisce un *batch* da pubblicare su Ethereum. Diversi batch possono essere pubblicati in maniera compressa, prendendo il nome di *canale*. Un canale può essere diviso in diversi *frame*, nel caso esso sia troppo grande per una singola transazione. Un canale viene identificato da un timestamp e un valore casuale in modo tale da poter essere pubblicati in ordine non sequenziale, come anche i loro frame.

I campi di un frame sono: `channel_id`, `random`, `timestamp`, `is_last`, `frame_data`, `frame_data_length` e `frame_number`. Un canale è definito come la compressione con ZLIB [26] di batch con codifica rlp. I campi di un batch sono: `epoch_number`, `epoch_hash`, `parent_hash`, `timestamp` e `tx_list`.

Derivazione

Una finestra di sequenziamento, identificata da un'epoca, contiene un numero w fissato di blocchi consecutivi di L1 che un passo di derivazione prende come input per costruire un numero variabile di blocchi di L2. Per l'epoca n, la finestra di sequenziamento n comprende i blocchi $[n, n + w)$. Ciò implica che l'ordinamento delle transazioni e dei blocchi di L2 all'interno di una finestra di sequenziamento non è fissato finché questa non termina. Una transazione del rollup è definita *sicura* se il batch che la contiene è stato confermato su L1.

I frame vengono letti dai blocchi di L1 per ricostruire i batch. L'implementazione attuale non consente di iniziare la decompressione di un canale prima di aver ricevuto tutti i frame corrispondenti. I batch non validi vengono ignorati.

Dai batch vengono ottenute le transazioni dei singoli blocchi, che vengono usate dal motore di esecuzione per applicare le transizioni di stato e ottenere lo stato del rollup.

2.2.4 Prelievi

Per poter processare i prelievi si implementa un sistema di messaging L2-to-L1.

Ethereum necessita di conoscere lo stato del L2 per accettare i prelievi, e questo viene fatto pubblicando sullo smart contract `L2OutputOracle` su L1 le radici degli stati di L2 per ogni blocco. Queste radici vengono accettate ottimisticamente come valide (o finalizzate) se durante il *periodo di disputa* non viene eseguita alcuna prova d'invalidità. Soltanto gli indirizzi designati come *Proponente* possono pubblicare delle radici di output. La validità delle radici di output viene incentivata facendo depositare ai Proponenti uno stake che viene sottratto se viene dimostrato che hanno proposto una radice invalida. Le transazioni vengono iniziate chiamando la funzione `initiateWithdrawal` sul predeploy all'indirizzo `0x4200000000000000000000000000000000000000` su L2 e poi finalizzate su L1 chiamando la funzione `finalizeWithdrawalTransaction` sull'Optimism Portal già menzionato in precedenza.

```
/**
 * @notice Sends a message from L2 to L1.
 *
 * @param _target   Address to call on L1 execution.
 * @param _gasLimit Minimum gas limit for executing the message on L1.
 * @param _data     Data to forward to L1 target.
 */
function initiateWithdrawal(
    address _target,
    uint256 _gasLimit,
    bytes memory _data
) public payable {
    bytes32 withdrawalHash = Hashing.hashWithdrawal(
        Types.WithdrawalTransaction({
```

```
            nonce: nonce,
            sender: msg.sender,
            target: _target,
            value: msg.value,
            gasLimit: _gasLimit,
            data: _data
        })
    );

    sentMessages[withdrawalHash] = true;

    emit WithdrawalInitiated(nonce, msg.sender, _target, msg.value,
       _gasLimit, _data);
    unchecked {
        ++nonce;
    }
}
```

Listing 2.1: Funzione del predeploy su L2 per iniziare un prelievo.

Per verificare e finalizzare un prelievo i seguenti input sono necessari:

- `nonce`: il nonce del messaggio.
- `sender`: l'indirizzo del sender su L2.
- `target`: l'indirizzo che viene chiamato su L1.
- `data`: i dati da inviare al target.
- `value`: la quantità di ETH da inviare al target.
- `gasLimit`: il gas da inoltrare al target.
- `timestamp`: il marcatore del tempo di L2 corrispondente alla radice di output.
- `outputRootProof`: 4 byte utilizzati per derivare la radice di output.
- `withdrawalProof`: prova di inclusione del prelievo.
- `l2BlockNumber`: il numero del blocco dove il prelievo è stato iniziato.

Viene ottenuta la radice di output corrispondente al `l2BlockNumber` dall'`L2OutputOracle`, si verifica che sia finalizzata, ovvero che sia passato il periodo di disputa, si verifica che l'`outputRootProof` corrisponda alla radice dell'oracolo, si verifica che l'hash del prelievo sia incluso in essa usando la `withdrawalProof`, che il prelievo non sia già stato finalizzato e poi si esegue la chiamata al `target`, inviando `gasLimit` quantità di gas, `value` quantità di Ether e i `data`.

```
function finalizeWithdrawalTransaction(
    Types.WithdrawalTransaction memory _tx,
    uint256 _l2BlockNumber,
    Types.OutputRootProof calldata _outputRootProof,
    bytes calldata _withdrawalProof
) external payable {
    require(
        l2Sender == DEFAULT_L2_SENDER,
        "OptimismPortal: can only trigger one withdrawal per
           transaction"
    );

    require(
        _tx.target != address(this),
        "OptimismPortal: you cannot send messages to the portal
           contract"
    );

    Types.OutputProposal memory proposal = L2_ORACLE.getL2Output(
       _l2BlockNumber);

    require(_isOutputFinalized(proposal), "OptimismPortal: proposal is
       not yet finalized");

    require(
        proposal.outputRoot == Hashing.hashOutputRootProof(
           _outputRootProof),
        "OptimismPortal: invalid output root proof"
    );

    bytes32 withdrawalHash = Hashing.hashWithdrawal(_tx);

    require(
        _verifyWithdrawalInclusion(
            withdrawalHash,
            _outputRootProof.withdrawerStorageRoot,
            _withdrawalProof\node{r}
    child { node {a} }
    child { node {b} };
        ),
        "OptimismPortal: invalid withdrawal inclusion proof"
    );

    require(
        finalizedWithdrawals[withdrawalHash] == false,
        "OptimismPortal: withdrawal has already been finalized"
    );
```

```
    finalizedWithdrawals[withdrawalHash] = true;

    require(
        gasleft() >= _tx.gasLimit + FINALIZE_GAS_BUFFER,
        "OptimismPortal: insufficient gas to finalize withdrawal"
    );

    l2Sender = _tx.sender;

    (bool success, ) = ExcessivelySafeCall.excessivelySafeCall(
        _tx.target,
        _tx.gasLimit,
        _tx.value,
        0,
        _tx.data
    );

    l2Sender = DEFAULT_L2_SENDER;

    emit WithdrawalFinalized(withdrawalHash, success);
}
```

Listing 2.2: Finalizzazione di un prelievo.

2.2.5 Cannon: il sistema di prove d'invalidità

Se un validatore del rollup, eseguendo localmente i batch e le transazioni depositate, scopre che lo stato del Layer 2 non corrisponde alla radice dello stato pubblicato on-chain dal Sequenziatore, può eseguire una prova d'invalidità su L1 per dimostrare che il risultato della transizione di stato del blocco è sbagliato. A causa dell'overhead elaborare un intero blocco del Rollup su L1 è troppo costoso. La soluzione è quella di eseguire on-chain soltanto la prima istruzione di `minigeth` in cui ci trova in disaccordo, compilandolo in una architettura MIPS che viene eseguite su un interprete on-chain (di soli 400 righe) e pubblicato su L1. `minigeth` è una versione semplificata di geth in cui è stato rimosso il PoW, l'RPC e il database.

Per trovare l'istruzione di disaccordo si svolge una ricerca binaria interattiva tra colui che esegue la la prova d'invalidità e chi ha pubblicato la radice di output. Quando la disputa inizia, entrambe le parti pubblicano la radice dello stato della memoria di MIPS a metà dell'esecuzione del blocco sul contratto `Challenge`: se l'hash coincide significa che entrambe le parti sono in accordo sulla prima metà dell'esecuzione pubblicando così la radice della metà della seconda parte, altrimenti si pubblica la metà della prima parte. Così facendo si raggiunge la prima singola istruzione di disaccordo in tempo logaritmico. Se uno dei due smette di interagire, al termine del periodo di disputa l'altro partecipante vince automaticamente.

Accesso alla memoria

L'interprete MIPS per elaborare questa istruzione ha bisogno di accedere alla sua memoria: siccome si ha la radice, le celle necessarie possono essere pubblicate dimostrandone l'inclusione. Per accedere allo stato della EVM si fa uso dell'*Oracolo delle Preimmagini*: dato l'hash di un blocco, esso ritorna il block header, da cui si può ottenere l'hash del blocco precedente e tornare indietro nella chain, oppure ottenere l'hash dello stato e dei log da cui si può ottenere la preimmagine. L'oracolo viene implementato da `minigeth` e sostituisce il database. Per ottenere le preimmagini vengono effettuate delle query ad altri nodi.

Capitolo 3

Rollup di Validità

L'obiettivo di un Rollup di Validità è quello di dimostrare formalmente la validità di una transizione di stato data la sequenza di transazioni per permetterne la verifica in un tempo minore di quello che si spenderebbe processandole tutte interamente.

Essi si differenziano in base al loro grado di compatibilità con la EVM [19]:

- **Tipo 1**: il Rollup è totalmente equivalente ad Ethereum e non fa alcuna modifica per facilitare la generazione delle prove. Il vantaggio è che ogni strumento è immediatamente riutilizzabile, ma siccome Ethereum non è pensato per essere dimostrato da prove di validità dimostrare un solo blocco può richiedere ore. Attualmente un solo team di ricerca sta cercando di implementare un Rollup di Validità di Tipo 1 [71].

- **Tipo 2**: il Rollup è totalmente equivalente alla EVM, ma presenta differenze rispetto ad Ethereum come alcune strutture dati, ad esempio la struttura dei blocchi o l'albero dello stato. I client di Ethereum non sono riutilizzabili senza modifiche ma la maggior parte delle applicazioni continuerebbe a funzionare. Il vantaggio è che il tempo di dimostrazione diminuisce rispetto al Tipo 1, ma la maggior parte della complessità è causata dalla EVM stessa. Scroll [80] e Polygon Hermez [73] stanno costruendo questo tipo di Rollup.

- **Tipo 3**: il Rollup è quasi equivalente alla EVM con qualche differenza per aumentare la velocità del dimostratore. Attualmente nessuno sta costruendo un Rollup di Tipo 3, ma è possibile che Scroll e Hermez vengano rilasciati inizialmente in questo modo per poi raggiungere l'equivalenza nel tempo.

- **Tipo 4**: il Rollup utilizza una macchina virtuale differente dalla EVM, ma è possibile compilare un linguaggio della EVM in un linguaggio per questa macchina virtuale. In questo modo si ottiene la massima velocità di dimostrazione a discapito della compatibilità con gli strumenti per Ethereum. ZKSync [74] e StarkNet [86] ne sono due esempi.

3.1 Preliminari

In generale, data una funzione, un input e un output si vuole creare un certificato che dimostri che l'output si ottiene calcolando la funzione sull'input e che la verifica del certificato costi meno del costo della funzione. Questi certificati prendono il nome di *prove di integrità computazionale* [13]. La proprietà di una soluzione di essere facilmente verificabile ma difficilmente calcolabile si conosce bene nello studio della complessità, come accade per i problemi NP-completi. Con le prove di integrità computazionale si cerca di sfruttare questa asimmetria anche per computazioni arbitrarie: nel 1991 venne dimostrato il teorema PCP [1], che afferma che per qualsiasi dimostrazione in NP, se il dimostratore esegue soltanto una quantità polinomiale di lavoro extra, un verificatore può validare la prova in tempo poli-logaritmico [8]. Le PCP vengono generalizzate alle *prove interattive ad oracolo* per problemi oltre la classe NP [5].

3.1.1 Crittografia omomorfica

La crittografia omomorfica permette di cifrare un valore a cui è comunque possibile applicare operazioni aritmetiche.

Un gruppo $\mathbb{G}$ è un insieme di elementi e un'operazione binaria (denotata qui con $\cdot$) che ha le seguenti quattro proprietà:

- **Chiusura**: $\forall a, b \in \mathbb{G} : a \cdot b \in \mathbb{G}$.
- **Associatività**: $\forall a, b, c \in \mathbb{G} : a \cdot (b \cdot c) = (a \cdot b) \cdot c$.
- **Identità**: esiste un elemento denotato $1_{\mathbb{G}}$ tale che $\forall a \in \mathbb{G} : 1_{\mathbb{G}} \cdot a = a \cdot 1_{\mathbb{G}} = a$.
- **Invertibilità**: $\forall a \in \mathbb{G}, \exists b \in \mathbb{G} : a \cdot b = 1_{\mathbb{G}}$. Questo elemento è denotato con a^{-1}.

Se l'operazione del gruppo è anche commutativa, il gruppo è detto *abeliano*. Un gruppo è detto *ciclico* se esiste un elemento del gruppo g, detto *generatore*, tale che tutti gli elementi del gruppo possono essere scritti come g^i per qualche intero positivo i. [2] Ogni gruppo ciclico è abeliano. Un sottogruppo $\mathbb{H}$ di $\mathbb{G}$ è un sottoinsieme di $\mathbb{G}$ che forma un gruppo sotto la stessa operazione di $\mathbb{G}$. La cardinalità $|\mathbb{G}|$ è detto *ordine* di $\mathbb{G}$. Per il Teorema di Lagrange, l'ordine di un sottogruppo $\mathbb{H}$ di $\mathbb{G}$ divide l'ordine di $\mathbb{G}$.

Dato un gruppo $\mathbb{G}$, il problema del logaritmo discreto prende due elementi del gruppo a e b e ritorna un intero positivo i tale che $a^i = b$. Se $\mathbb{G}$ è ciclico è garantito che questo i esista. Si crede che trovare il logaritmo discreto sia computazionalmente intrattabile, ma è stato dimostrato che un computer quantistico può risolvere il problema in tempo

[1] da Probabilistic Checkable Proof.

[2] dove g^i denota $\underbrace{g \cdot g \cdot \cdots \cdot g}_{i \text{ copie di } g}$.

polinomiale utilizzando l'algoritmo di Shor [82]. L'algoritmo classico usato in pratica più veloce che si conosce opera in tempo $O(\sqrt{|\mathbb{G}|})$ [72].

In crittografia, i gruppi usati sono tipicamente sottogruppi ciclici di gruppi definiti usando curve ellittiche su campi finiti, o gruppi moltiplicativi di interi modulo un grande numero primo.

Dato un generatore g di un gruppo ciclico, è quindi possibile cifrare un valore a calcolando $h = g^a$. Dato questo schema, è possibile moltiplicare un valore cifrato per un valore noto b calcolando h^b, mantenendo il risultato cifrato:

$$h^b = (g^a)^b = g^{a \cdot b}$$

Inoltre è possibile sommare due valori cifrati usando la moltiplicazione:

$$g^a \cdot g^b = g^{a+b}$$

e similmente sottrarre due valori cifrati con la divisione:

$$\frac{g^a}{g^b} = g^{a-b}$$

Non è possibile moltiplicare o dividere due valori cifrati tra di loro e non è possibile calcolare la potenza di valore cifrato.

3.1.2 Dimostrazioni probabilistiche

Le dimostrazioni probabilistiche possono essere usate per migliorare drasticamente l'efficienza di alcuni algoritmi. Ad esempio, Freivalds [36] dimostrò come verificare che una matrice C sia il prodotto di due matrici A e B di dimensione $n \times n$ in tempo $O(n^2)$ (soltanto una costante in più rispetto a leggere le matrici), mentre l'algoritmo più veloce che si conosce per moltiplicare due matrici viene eseguito in circa tempo $O(n^{2.37286})$ [60] [4].

La tecnica utilizzata si basa sul convertire l'informazione in un vettore di lunghezza n, interpretare gli elementi come coefficienti di un polinomio, codificare il vettore come valutazioni del polinomio in un campo $\mathbb{F}_p$ dove $p \gg n$ per ogni $r \in \mathbb{F}_p$ e sfruttare il fatto che due polinomi diversi di grado n si uguagliano su al massimo $n-1$ punti, permettendo di verificare probabilisticamente la loro uguaglianza usando del campionamento. Questa proprietà può essere estesa ai polinomi a più variabili per ridurre il grado del polinomio [79]. Siccome le codifiche vengono interpretate come valutazioni, non è necessario costruire l'intero vettore. Questa tecnica prende il nome di codifica di Reed-Solomon [90], in cui si sfrutta la proprietà di amplificazione della distanza. La codifica può essere alternativamente costruita interpretando gli elementi del vettore originale come le valutazioni e ottenere il polinomio utilizzando tecniche di interpolazione polinomiale come l'interpolazione di Lagrange, con il vantaggio che il vettore originale è un sotto-vettore di quello esteso.

Esempio 1: Confronto probabilistico di due stringhe

Alice e Bob vogliono determinare se i file che possiedono sono uguali minimizzando la quantità di informazione scambiata durante la comunicazione. I file consistono in una sequenza di n caratteri ASCII, quindi $m = 128$ caratteri possibili. Alice posside il file $(a_1, \ldots, a_n)$, mentre Bob il file $(b_1, \ldots, b_n)$. La soluzione banale è inviare tutti gli n caratteri, ma ciò non è possibile se n è molto grande. Non esiste nessuna procedura deterministica che invia meno informazione [53], quindi decidono di usare una procedura probabilistica. Si fissa un numero primo $p \geq \max\{m, n^2\}$ per il campo $\mathbb{F}_p$. Per il resto dell'esempio si assume che tutte le operazioni sono svolte in questo campo. Si definisce la famiglia di funzioni di hash $\mathcal{H} = \{h_r : r \in \mathbb{F}_p\}$ dove $h_r(a_1, \ldots, a_n) = \sum_{i=1}^{n} a_i \cdot r^{i-1}$, ovvero il risultato della valutazione del polinomio di grado $n-1$ su r ottenuto interpretando i caratteri come coefficienti. Alice sceglie un elemento casuale $r \in \mathbb{F}_p$, calcola $v = h_r(a)$ e invia r e v a Bob. Bob verifica se $v = h_r(b)$, in caso positivo dà in output UGUALI altrimenti NON-UGUALI.

Completezza e correttezza

Banalmente, se $\forall i \in \{1, \ldots, n\} : a_i = b_i$, allora Bob dà in output UGUALI per ogni scelta di r. Se c'è almeno un i per cui $a_i \neq b_i$, allora Bob dà in output NON-UGUALI con probabilità almeno $1 - (n-1)/p$, ovvero almeno $1 - 1/n$ per $p \geq n^2$. Per dimostrare ciò, sia $p_a(x) = \sum_{i=1}^{n} a_i \cdot x^{i-1}$ e similmente $p_b(x) = \sum_{i=1}^{n} b_i \cdot x^{i-1}$: se c'è almeno un $a_i \neq b_i$, allora ci sono al massimo $n-1$ valori di r tali che $p_a(r) = p_b(r)$. Siccome r è scelto casualmente da $\mathbb{F}_p$, la probabilità che Alice estragga un tale r è al massimo $(n-1)/p$, perciò la probabilità che egli dia in output NON-UGUALI è almeno $1 - (n-1)/p$.

Costo del protocollo

La procedura deterministica ha un costo di $n \log m$ bit scambiati. Nella procedura probabilistica, vengono scambiati soltanto due elementi di $\mathbb{F}_p$, ovvero v e r: assumendo $p \leq n^c$ per qualche costante c, il costo è $O(\log n)$.

3.1.3 Dimostrazioni interattive

Le dimostrazioni interattive vennero introdotte formalmente la prima volta da Babai [7].

Data una funzione $f : \{0,1\}^n \to R$ dove R è un intervallo finito, un *sistema di dimostrazione interattivo a k messaggi* per f consiste un algoritmo probabilistico V, detto *Verificatore* che viene eseguito in tempo polinomiale e un algoritmo deterministico P detto *Dimostratore* (o Prover). Sia a V che a P viene dato un input comune $x \in$

$\{0,1\}^n$ e all'inizio del protocollo P produce un valore y che afferma essere uguale a $f(x)$. Successivamente, P e V si scambiano una serie di k messaggi in maniera alternata: al termine il Verificatore risponderà con 1 o 0 in base a se accetta l'affermazione del Dimostratore per cui $y = f(x)$ oppure no. Il Verificatore può essere reso deterministico fissando a priori la fonte di casualità interna. Si denota con $\text{out}(V, x, r, P) \in \{0,1\}$ l'output del Verificatore V su input x che interagisce con P con valore casuale interno r.

Si dice che un sistema di dimostrazione interattivo possiede errore di completezza δ_C e errore di correttezza (soundness) δ_S se valgono le seguenti proprietà:

1. **Completezza**: per ogni $x \in \{0,1\}^n$ e P onesto,

$$\mathbb{P}[\text{out}(V, x, r, P) = 1] \geq 1 - \delta_C$$

2. **Correttezza**: per ogni $x \in \{0,1\}^n$ e per ogni $y \neq f(x)$ inviato da P' all'inizio del protocollo,

$$\mathbb{P}[\text{out}(V, x, r, P') = 1] \leq \delta_S$$

Per convenzione, si dice che un sistema di dimostrazione interattivo è valido se $\delta_C, \delta_S \leq 1/3$. Oltre al costo in tempo, determinante è il costo il spazio, il numero di bit comunicati, il numero di messaggi inviati e se r è un valore reso pubblico o no.

Le dimostrazioni interattive illustrate in seguito avranno completezza perfetta, ovvero $\delta_C = 0$, mentre l'errore di correttezza sarà proporzionale a $1/|\mathbb{F}|$ dove $\mathbb{F}$ è il campo sul quale la dimostrazione è definita. A livello pratico verrà scelto un campo tale per cui l'errore di correttezza è estremamente piccolo ($\leq 2^{-128}$). Questo errore può essere ulteriormente ridotto a δ_S^k ripetendo il protocollo k volte.

Le dimostrazioni interattive possono essere adattate anche ai linguaggi, richiedendo che V accetti con alta probabilità una parola del linguaggio e rifiuti con alta probabilità una parola non contenuta nel linguaggio. La differenza principale ad usare linguaggi invece che funzioni è che non viene richiesto una dimostrazione di non inclusione per parole non incluse. Dato una funzione f, una dimostrazione interattiva per f è equivalente ad un dimostrazione interattiva per il linguaggio $\mathcal{L}_f \coloneqq \{(x, y) : y = f(x)\}$.

IP = PSPACE

Sia IP la classe dei linguaggi la cui appartenenza è dimostrabile usando un sistema di dimostrazione interattivo con un Verificatore che opera in tempo polinomiale. La classe NP è un sottoinsieme di IP in quanto può essere vista come una sua restrizione in cui le dimostrazioni sono deterministiche e non interattive, ovvero che hanno errore di completezza e correttezza pari a zero. Adi Shamir [81] riuscì a caratterizzare la classe IP dimostrandone l'equivalenza a PSPACE, che si crede essere molto più grande di NP.

3.1.4 Prove a conoscenza zero

Le prove a conoscenza zero [3] sono un tipo di dimostrazione probabilistica tra un Dimostratore e un Verificatore introdotte per la prima volta da Goldwasser et al. [40]. Data una proposizione x, può esistere una *witness* w che dimostra una relazione R tra x e w ad un Verificatore V. Per esempio, x può essere un hash, e w la preimmagine tale che vale la relazione $x = H(w)$, dove H è una funzione hash. Si dice che x è vera se e solo se esiste w tale che $R(x, w)$ vale. Il Dimostratore P vuole dimostrare a V che x è vera.

Le tre proprietà di una prova a conoscenza zero sono:

- **Completezza**: se $R(x, w)$ è vera, allora $P(x, w) \iff V(x)$: se P possiede una witness che dimostra la proposizione, allora sarà in grado di convincere V che la proposizione è vera. Importante notare che il Verificatore non prende la witness come input.
- **Correttezza**: se x è falso, $V(x)$ rifiuterà (con alta probabilità) per ogni P.
  - **Correttezza della conoscenza**: se P fa accettare $V(x)$, allora esiste un modo per estrarre w tale che $R(x, w)$. Spesso viene richiesta questa forma più forte di correttezza, in quanto esistono delle proposizioni che sono banalmente sempre vere, come l'esistenza di una preimmagine di un hash a 256 bit per SHA256.
- **Conoscenza zero**: se $R(x, w)$ vale, allora possiamo simulare l'esecuzione della prova usando soltanto x. Questo dimostra che il Verificatore non è in grado di estrarre nessuna informazione su w.

Esempio 2: Protocollo di Schnorr

Il protocollo di Schnorr [78] è un utilizzo molto semplice delle prove a conoscenza zero in quanto vengono applicate ad un problema specifico e la tecnica non è generale. Sia p un numero primo e sia g un generatore pubblicamente noto di un gruppo ciclico $\mathbb{G}$ di ordine primo q. Per generare una coppia di chiave, Alice:

- Sceglie un numero casuale a da 1 a q.
- Calcola la chiave pubblica $\mathrm{PK}_A = g^a \mod p$.
- Salva la chiave segreta $\mathrm{SK}_A = a$.

Alice vuole dimostrare a Bob che conosce la chiave segreta (la witness w) che corrisponde alla chiave privata usando un protocollo interattivo:

[3] in inglese *zero-knowledge proof*.

1. Alice sceglie un k casuale da 1 a q e calcola $h = g^k \mod p$ e lo invia a Bob.
2. Bob sceglie un c casuale e lo invia ad Alice.
3. Alice risponde con $s = ac + k \mod q$.
4. Bob controlla che $g^s \equiv \mathrm{PK}_A^c \cdot h \mod p$.

Il protocollo è una prova a conoscenza zero in quanto rispetta le tre proprietà prima elencate:

- **Completezza**:

$$\begin{aligned} g^s &\equiv PK_A^c \cdot h \mod p \\ g^{ac+k} &\equiv (g^a)^c \cdot g^k \mod p \\ g^{ac+k} &\equiv g^{ac+k} \mod p \end{aligned}$$

- **Correttezza della conoscenza**: per dimostrare che Alice conosce effettivamente la chiave segreta a, si utilizza un tipo speciale di Verifier chiamato *estrattore di conoscenza*, in grado di estrarre la chiave dalla dimostrazione. Ciò non contraddice la proprietà di conoscenza zero in quanto l'estrattore non è ammesso se la prova è svolta correttamente. L'estrattore, dopo il terzo step, fa ripartire il protocollo dal passo 2, facendo usare ad Alice lo stesso blinding factor k. Con questa strategia è possibile estrarre SK_A:

$$\begin{aligned} & (s_1 - s_2)/(c_1 - c_2) && \mod q \\ = & (ac_1 + k - ac_2 - k)/(c_1 - c_2) && \mod q \\ = & a(c_1 - c_2)/(c_1 - c_2) && \mod q \\ = & a \end{aligned}$$

- **Conoscenza zero**: l'obiettivo è dimostrare che si conosce il segreto a per qualche chiave pubblica $g^a \mod p$ senza conoscere a, utilizzando un tipo speciale di Prover chiamato *Simulator*. Alice invia a Bob g^{k_1} per scoprire quale c casuale Bob sceglie. Dopodiché riavvolge il protocollo, scegliendo questa volta $g^{k_2} = g^z * g^{a(-c)}$ come messaggio iniziale. Quando Bob invia nuovamente la challenge c, Alice risponde con z. Siccome dal punto di vista del Verificatore questo protocollo è identico a quello reale, essa non svela alcuna informazione su a. Ciò non è in contrasto con la proprietà di correttezza in quanto il Simulator non è ammesso se la prova è svolta correttamente.

3.1.5 SNARK

Una prova SNARK (Succint Non-interactive ARgument of Knowledge) è una categoria di dimostrazione probabilistica per computazione arbitraria. Le caratteristiche principali sono la dimensione costante delle prove e il tempo costante di verifica. La loro non interattività è data da una inizializzazione fidata per generare la fonte di casualità.

Per costruire una SNARK i passi da effettuare sono:

1. Conversione della computazione in un circuito aritmetico.
2. Conversione del circuito in un Rank-1 Constraint System (R1CS).
3. Costruzione del Quadratic Arithmetic Program (QAP).
4. Costruzione prova SNARK.

A scopo illustrativo, si suppone di dover dimostrare la corretta esecuzione del seguente programma per qualche x:

```
def f(x):
    y = x**3
    return y + 8
```

che calcola il risultato del polinomio $x^3 + 8$.

Circuiti aritmetici

Un circuito aritmetico prende in input dei segnali numerici e applica ad essi delle moltiplicazioni e addizioni in un campo finito primo. Un circuito aritmetico può avere multipli segnali intermedi e un segnale di output. Uno dei linguaggi più utilizzati per definire circuiti aritmetici è Circom.

Il codice preso in esame deve essere convertire in uno che utilizza esclusivamente l'assegnazione e operazioni della forma $x = y@z$ dove $@ \in (+, -, *, /)$ in un processo chiamato *appiattimento*. Il codice appiattito corrispondente è quindi:

```
def f(x):
    n = x*x
    m = n*x
    out = m + 8
```

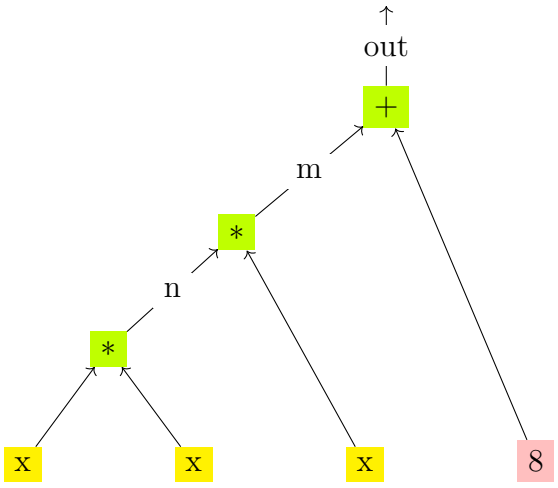


Figura 3.1: Il programma rappresentato con un circuito aritmetico.

R1CS

I gate di un circuito aritmetico vengono rappresentati come un insieme di equazioni di un Rank-1 Constraint System (R1CS). Un R1CS è una sequenza di gruppi di tre vettori (a_i, b_i, c_i) chiamati vincoli e una soluzione è un vettore s tale che $\forall i : s \cdot a_i * s \cdot b_i - s \cdot c_i = 0$. La lunghezza dei vettori è data dal numeri di variabili del sistema e in più il numero 1 per esprimere le costanti e una variabile `out` rappresentate l'output. Ogni gate del circuito viene rappresentato da un vincolo.

Il vettore s corrispondente al programma in esame è:

$$s = \begin{bmatrix} 1 \\ x \\ n \\ m \\ \text{out} \end{bmatrix}$$

Per ogni linea del programma appiattito viene poi definito un vincolo. Per la riga 2 si definisce il seguente gruppo: $a_1 = [0\ 1\ 0\ 0\ 0],\ b_1 = [0\ 1\ 0\ 0\ 0],\ c_1 = [0\ 0\ 1\ 0\ 0]$. Da notare che il vincolo $s \cdot a_1 * s \cdot b_1 - s \cdot c_1 = 0$ è rispettato.

Gli altri vincoli sono:

- Riga 3: $a_2 = [0\ 0\ 1\ 0\ 0],\ b_2 = [0\ 1\ 0\ 0\ 0],\ c_2 = [0\ 0\ 0\ 1\ 0]$.

- Riga 4: $a_3 = [8\ 0\ 0\ 1\ 0],\ b_3 = [1\ 0\ 0\ 0\ 0],\ c_3 = [0\ 0\ 0\ 0\ 1]$.

Eseguire il programma è equivalente a trovare un s tale che rispetta il R1CS. Un esempio è $s = [1\ 3\ 9\ 27\ 35]$.

QAP

Il passaggio successivo è convertire i vettori in polinomi $A_i(x)$, $B_i(x)$ e $C_i(x)$ per $i \in [1, N]$ dove N è il numero di elementi nei vincoli. Questi polinomi vengono costruiti richiedendo che $A_i(n) = a_{n,i}$, $B_i(n) = b_{n,i}$ e $C_i(n) = c_{n,i}$ e usando l'interpolazione di Lagrange. Nel nostro caso si passa da 3 gruppi di 3 vettori di lunghezza 5 a 5 gruppi di 3 polinomi di grado 2.

Ad esempio, per A_1 si ha $A_1(1) = 0$, $A_1(2) = 0$, $A_1(3) = 8$ e interpolando si ottiene $A_1(x) = 4x^2 - 12x + 8$.

Similmente per gli altri polinomi:

- $A_2(x) = \frac{1}{2}x^2 - \frac{5}{2}x + 3$
- $A_3(x) = -x^2 + 4x - 3$
- $A_4(x) = \frac{1}{2}x^2 - \frac{3}{2}x + 1$
- $A_5(x) = 0$
- $B_1(x) = \frac{1}{2}x^2 - \frac{3}{2}x + 1$
- $B_2(x) = -\frac{1}{2}x^2 + \frac{3}{2}x$
- $B_3(x) = 0$
- $B_4(x) = 0$
- $B_5(x) = 0$
- $C_1(x) = 0$
- $C_2(x) = 0$
- $C_3(x) = \frac{1}{2}x^2 - \frac{5}{2}x + 3$
- $C_4(x) = -x^2 + 4x - 3$
- $C_5(x) = \frac{1}{2}x^2 - \frac{3}{2}x + 1$

Calcolando per $x = 1$ si ottiene il primo gruppo di vincoli, per $x = 2$ il secondo gruppo e così via. Ad esempio, calcolando per $x = 3$ si ottiene:

$$[A_1(3)\ A_2(3)\ A_3(3)\ A_4(3)\ A_5(3)] = [8\ 0\ 0\ 1\ 0] = a_3$$

$$[B_1(3)\ B_2(3)\ B_3(3)\ B_4(3)\ B_5(3)] = [1\ 0\ 0\ 0\ 0] = b_3$$

$$[C_1(3)\ C_2(3)\ C_3(3)\ C_4(3)\ C_5(3)] = [0\ 0\ 0\ 0\ 1] = c_3$$

A questo punto il R1CS può essere espresso come una sola equazione:

$$A(x) * B(x) - C(x) = H(x) * Z(x)$$

dove $A(x)$ è il prodotto scalare tra s e $[A_1(x)\ A_2(x)\ A_3(x)\ A_4(x)\ A_5(x)]$ e similmente per $B(x)$ e $C(x)$. Siccome $\forall x \in [1, 3] : P(x) = A(x) * B(x) - C(x) = 0$, il polinomio è un multiplo di $Z(x) = \prod_{i=1}^{3}(x - i)$. $H(x)$ viene calcolato come $P(x)/Z(x)$.

Nell'esempio presentato:

$$P(x) = -10x^4 + 54x^3 - 74x^2 - 6x + 36$$
$$Z(x) = x^3 - 6x^2 + 11x - 6$$
$$H(x) = -10x - 6$$

Se s non è una soluzione del R1CS allora $P(x)$ non è divisibile per $Z(x)$.

Usando questa proprietà si sviluppa il seguente protocollo naive:

1. Il Verificatore genera una valore casuale r, e conoscendo il numero di vincoli calcola $Z(r)$ e invia r al Dimostratore.

2. Il Dimostratore calcola il polinomio $H(x)$, valuta $P(r)$ e $H(r)$ e li invia al Verificatore.

3. Il Verificatore controlla che $P(r) = H(r) * Z(r)$. Siccome due polinomi diversi di grado n si eguagliano su al massimo $n - 1$ punti, se l'uguaglianza vale si ha un'alta probabilità che il Dimostratore conosca effettivamente $P(x)$.

Un problema è che il Dimostratore potrebbe usare un h casuale e calcolare un p tale che $p = h * Z(r)$ senza effettivamente conoscere $P(x)$. Inoltre non viene effettuato nessun controllo sul grado del polinomio.

Una soluzione parziale è nascondere il valore di r usando la crittografia omomorfica:

1. Il Verificatore genera un valore casuale r, valuta $Z(r)$ e calcola i valori cifrati delle potenze di r necessarie per il polinomio, nel nostro caso g^{r^0}, g^{r^1}, g^{r^2}, g^{r^3} e g^{r^4} e le invia al Dimostratore. [4]

[4] perché, come spiegato in precedenza, non è possibile calcolare la potenza di un valore cifrato.

2. Il Dimostratore calcola $H(x)$, usando i valori cifrati calcola $g^{P(r)}$ e similmente $g^{H(r)}$ e li invia al Verificatore.

3. Il Verificatore controlla che $g^{P(r)} = \left(g^{H(r)}\right)^{Z(r)} = g^{H(r)\cdot Z(r)}$.

In questa maniera il Dimostratore viene ristretto ad usare la selezione delle potenze di r che gli viene consegnata, ma questa restrizione non viene imposta: ad esempio, dato un valore casuale s, il Dimostratore può calcolare $z_h = g^s$ e $z_p = \left(g^{Z(r)}\right)^s$. Il Verificatore verificando

$$\left(g^{Z(r)}\right)^s = (g^s)^{Z(r)}$$

accetterebbe erroneamente la prova.

Si ottiene il risultato desiderato utilizzando la Knowledge-of-Exponent Assumption (KEA1) [12] introdotta per la prima volta da Ivan Damgard [24]: informalmente, sia g un generatore, dati g e g^a, l'unico modo per calcolare una coppia (C, Y) dove $Y = C^a$ è calcolare, scegliendo un qualche c, $C = g^c$ e $Y = (g^a)^c$.

Questa assunzione può essere utilizzata per chiedere al Dimostratore di calcolare i risultati su g^r e su un valore *shiftato* $g^{\alpha r}$, usato come "checksum":

1. Il Verificatore invia al Dimostratore i valori g^{r^0}, g^{r^1}, ..., g^{r^d} e i loro shift $g^{\alpha r^0}$, $g^{\alpha r^1}$, ..., $g^{\alpha r^d}$.

2. Il Dimostratore usando questi valori calcola $g^p = g^{P(r)}$ e $g^{p'} = g^{\alpha P(r)}$.

3. Il Verificatore verifica che $(g^p)^\alpha = g^{p'}$.

Questo approccio è stato introdotto per la prima volta da Jens Groth [44], ideatore del sistema di dimostrazione Groth16 [43].

Conoscenza zero

La proprietà di conoscenza zero può essere attaccata usando un attacco a forza bruta sui coefficienti del polinomio: il protocollo deve essere sicuro anche se vi è un solo coefficiente ed esso è 1.

I controlli che il Verificatore esegue sono:

$$g^p = \left(g^{H(r)}\right)^{Z(r)} \qquad \text{(verifica delle radici del polinomio)}$$
$$(g^p)^\alpha = g^{p'} \qquad \text{(verifica dell'uso corretto del polinomio)}$$

La proprietà di conoscenza zero può essere conservata per attacchi di forza bruta applicando nuovamente uno shift sui valori di un valore δ. Per estrarre delle informazioni, il Verificatore dovrebbe trovare questo valore, che è considerato computazionalmente infattibile. Inoltre la randomizzazione è statisticamente indistinguibile dal caso.

Le operazioni del Verificatore così diventano:

$$(g^p)^\delta = \left(\left(g^{H(r)}\right)^\delta\right)^{Z(r)}$$
$$\left((g^p)^\delta\right)^\alpha = \left(g^{p'}\right)^\delta$$

Non interattività e inizializzazione fidata

L'unico che può essere certo della validità di una prova a conoscenza zero interattiva è il Verificatore. Da un punto di visto di un osservatore esterno, il Verificatore potrebbe aver colluso con il Dimostratore comunicandogli i valori segreti r e α. Ciò è utile in alcune applicazioni in cui non si vuole permettere di replicare la prova ad altri [48], ma nel caso dei sistemi distribuiti come la blockchain è inefficiente ricreare la prova per ognuno.

I parametri che si desidera mantenere segreti sono $Z(r)$ e α. Si potrebbe usare lo stesso metodo utilizzato per cifrare le potenze di r, ma come già menzionato la crittografia omomorfica non permette la moltiplicazione di due valore cifrati.

La soluzione è l'utilizzo di mappe bilineari crittografiche [27], ovvero una funzione $e(g^*, g^*)$ che dati due input cifrati g^a e g^b produce deterministicamente, utilizzando una mappa, la loro rappresentazione moltiplicata $e(g^a, g^b) = e(g, g)^{ab}$. Siccome la funzione utilizza come dominio e codominio due gruppi diversi, non è possibile moltiplicare il risultato per un altro valore cifrato. Le proprietà principali, ottenute utilizzando curve ellittiche, sono esprimibili come le seguenti equazioni:

$$e(g^a, g^b) = e(g^b, g^a) = e(g^{ab}, g^1) = e(g^1, g^{ab}) = e(g^1, g^a)^b = e(g^1, g^1)^{ab}$$

Si assume ci sia un partecipante fidato che generi i segreti r e α e che dopo aver calcolato le potenze cifrate e i loro α-shift decida di eliminare i valori in chiaro. Questi parametri prendono il nome di "stringa di riferimento comune" (CRS [5]). Essi si dividono in due gruppi:

- **Chiave di dimostrazione**: $\forall i \in \{0, \ldots, d\} : (g^{r^i}, g^{\alpha r^i})$
- **Chiave di verifica**: $(g^{Z(r)}, g^\alpha)$

Utilizzando la chiave di verifica e avendo ottenuto $g^p, g^{p'}$ e $g^{H(r)}$ dal Dimostratore, il Verificatore controlla:

- $e(g^p, g^1) = e(g^{Z(r)}, g^{H(r)})$
- $e(g^p, g^\alpha) = e(g^{p'}, g^1)$

[5] da Common Reference String.

Il problema di questo approccio è che bisogna fidarsi che i valori segreti, che prendono il nome di "rifiuti tossici", vengano eliminati. Eli Ben-Sasson et al. descrissero nel 2015 un metodo per minimizzare questa assunzione utilizzando una computazione multipartitica (MPC) [15], in cui un numero arbitrario di partecipanti contribuiscono ed è necessario che almeno uno di questi elimini i proprio valori generati per rendere la procedura sicura. ZCash, una blockchain anonima basata su SNARK [47], organizzò la "cerimonia delle potenze di tau" per generare la propria stringa di riferimento comune.

3.1.6 STARK

Il sistema di dimostrazione STARK è stato introdotto nel 2018 come caso speciale di SNARK [14]. Esso utilizza funzioni di hash come unica assunzione crittografica e la proprietà di conoscenza zero è opzionale. La complessità in tempo del Dimostratore è $O(T \cdot \text{poly}\log(T))$ rispetto al limite di tempo T della computazione originale, mentre in SNARK dipende dall'implementazione. La complessità in tempo per il Verificatore è $O(\text{poly}\log(T))$. I due grandi vantaggi di STARK in alternativa a SNARK sono l'assenza di un'inizializzazione fidata e la resistenza ad attacchi quantistici, in quanto non si fa utilizzo del problema del logaritmo discreto e delle curve ellittiche per rendere sicuro il sistema.

Le dimostrazioni sono rese non interattive utilizzando la trasformata di Fiat-Shamir [34]: per generare i valori casuali del Verifier viene utilizzata una funzione di hash che prende come input la trascrizione del protocollo fino al punto in cui questi valori sono necessari, in modo da impedire che esso scelga degli input che calcolano un hash adatto per generare prove valide ma incorrette.

Approfondimenti

Martin Furer et al. [37] dimostrarono che è possibile trasformare ogni sistema di dimostrazione interattivo con errore di completezza $\delta_C \leq 1/3$ in uno con completezza perfetta con blowup polinomiale del Verifier.

Goldwasser e Sipser [41] dimostrarono che è possibile convertire ogni sistema di dimostrazione interattivo a casualità privata ad uno a casualità pubblica con blowup polinomiale per il Verifier.

3.2 StarkNet

3.2.1 Panoramica

StarkNet è un Rollup di Validità sviluppato da StarkWare che utilizza il sistema di dimostrazione STARK per validare il proprio stato su Ethereum. Per facilitare la costruzione

delle prove di validità, si utilizza una macchina virtuale differente dalla EVM, il cui linguaggio ad alto livello è Cairo. Ad esempio, al posto di hash `keccak256` si usano, dove possibile, Pedersen hash.

3.2.2 Depositi

Gli utenti possono depositare transazioni tramite un contratto su Ethereum chiamando la funzione `sendMessageToL2`.

```
function sendMessageToL2(
    uint256 toAddress,
    uint256 selector,
    uint256[] calldata payload
) external override returns (bytes32) {
    uint256 nonce = l1ToL2MessageNonce();
    NamedStorage.setUintValue(L1L2_MESSAGE_NONCE_TAG, nonce + 1);
    emit LogMessageToL2(msg.sender, toAddress, selector, payload, nonce
        );
    bytes32 msgHash = getL1ToL2MsgHash(toAddress, selector, payload,
        nonce);
    l1ToL2Messages()[msgHash] += 1;

    return msgHash;
}
```

Il messaggio viene registrato nel contratto calcolandone l'hash e incrementandone un contatore. I Sequenziatori ascoltano l'evento `LogMessageToL2` e codificano le informazioni in una transazione di StarkNet che chiama una funzione di un contratto che possiede il decoratore `l1_handler`. Al termine dell'esecuzione, quando viene prodotta la prova della transizione di stato, viene allegato ad essa la consumazione del messaggio che viene cancellato decrementandone il contatore.

Di seguito un esempio di `l1_handler` per depositare token fungibili:

```
@l1_handler
func deposit{
    syscall_ptr : felt*,
    pedersen_ptr : HashBuiltin*,
    range_check_ptr,
}(from_address : felt, user : felt, amount : felt):
    # Make sure the message was sent by the intended L1 contract.
    assert from_address = L1_CONTRACT_ADDRESS

    # Read the current balance.
    let (res) = balance.read(user=user)

    # Compute and update the new balance.
    tempvar new_balance = res + amount
```

```
    balance.write(user, new_balance)

    return ()
end
```

Un contratto può avere più `l1_handler` che si differenziano tramite un selettore, ottenibile utilizzando la libreria Python di StarkNet:

```
from starkware.starknet.compiler.compile import get_selector_from_name

print(get_selector_from_name('deposit'))
```

Mercato del gas

L'inclusione delle transazioni depositate non è richiesta dalla specifica di StarkNet, per questo è necessario un mercato del gas che incentivi i Sequenziatori a pubblicarle su L2. Nella versione attuale, siccome il Sequenziatore è centralizzato e gestito da StarkWare, il costo delle transazioni depositate è solamente determinato dal costo dell'esecuzione del deposito. Dalla versione di StarkNet 0.10.0, il costo di esecuzione su L2 viene pagato inviando ETH a `sendMessageToL2` che diventa `payable`, e l'evento `LogMessageToL2` aggiunge un campo `uint256 fee`. Questi ETH rimangono bloccati su L1 e vengono trasferiti al Sequenziatore, sempre su L1, quando la transazione depositata viene inclusa in una transizione di stato. La quantità di ETH inviata, se la transazione depositata viene inclusa, viene interamente spesa, indipendentemente dalla quantità di gas consumata su L2.

Deposito degli attributi di L1

StarkNet non possiede un sistema che rende disponibili gli attributi dei blocchi di L1 automaticamente. In alternativa, Fossil è un protocollo sviluppato da Oiler Network che permette, dato un l'hash di un blocco, di ottenere qualsiasi informazione da Ethereum pubblicando le preimmagini [70].

3.2.3 Sequenziamento

Lo stato corrente di StarkNet può essere interamente derivato da Ethereum. Ogni differenza di stato tra una transizione e l'altra viene pubblicato su L1 come calldata.

Sequenziatore

Le differenze sono pubblicate per ogni contratto e sono salvate come `uint256[]` e seguono la seguente codifica:

- Numero di campi riguardanti pubblicazioni di contratti.

- Per ogni contratto pubblicato:
  - `contract_address`: l'address del contratto pubblicato.
  - `contract_hash`: l'hash del contratto pubblicato.
  - `len(constructor_call_data)`: il numero di argomenti del costruttore del contratto.
  - `constructor_call_data`: lista degli argomenti del costruttore.
- Numero di contratti il cui storage è stato modificato.
- Per ogni contratto modificato:
  - `contract_address`: l'address del contratto modificato.
  - `num_of_storage_updates`: numero di modifiche dello storage.
  - `key, value`: coppia dell'address dello storage del contratto il cui valore è stato modificato e il nuovo valore.

Derivazione

Le differenze di stato sono pubblicate in ordine, quindi è sufficiente leggerle in sequenza per ricostruire lo stato. La derivazione è implementata da Pathfinder, un client full node di StarkNet.

3.2.4 Prelievi

Per inviare un messaggio da L2 a L1 si utilizza la syscall `send_message_to_L1`. Il messaggio viene pubblicato su L1 aumentandone il contatore dell'hash insieme alla prova e finalizzato chiamando la funzione `consumeMessageFromL2` su L1, che decrementa il contatore.

```
function consumeMessageFromL2(uint256 fromAddress, uint256[] calldata
    payload)
    external
    override
    returns (bytes32)
{
    bytes32 msgHash = keccak256(
        abi.encodePacked(fromAddress, uint256(msg.sender), payload.
            length, payload)
    );

    require(l2ToL1Messages()[msgHash] > 0, "INVALID_MESSAGE_TO_CONSUME"
        );
    emit ConsumedMessageToL1(fromAddress, msg.sender, payload);
```

```
    l2ToL1Messages()[msgHash] -= 1;
    return msgHash;
}
```

Il bridge ufficiale di StarkNet è il StarkGate, ed utilizza così il sistema di messaging per trasferire ETH da L2 a L1:

```
function withdraw(uint256 amount, address recipient) public override {
    consumeMessage(amount, recipient);
    // Make sure we don't accidentally burn funds.
    require(recipient != address(0x0), "INVALID_RECIPIENT");
    require(address(this).balance - amount <= address(this).balance, "
        UNDERFLOW");
    recipient.performEthTransfer(amount);
}
```

```
function consumeMessage(uint256 amount, address recipient) internal {
    emit LogWithdrawal(recipient, amount);

    uint256[] memory payload = new uint256[](4);
    payload[0] = TRANSFER_FROM_STARKNET;
    payload[1] = uint256(recipient);
    payload[2] = amount & (UINT256_PART_SIZE - 1);
    payload[3] = amount >> UINT256_PART_SIZE_BITS;

    messagingContract().consumeMessageFromL2(l2TokenBridge(), payload);
}
```

Siccome nella chiamata di `consumeMessageFromL2` il chiamante è il contratto StarkGate, ognuno può finalizzare qualsiasi prelievo.

3.2.5 Dimostrazioni di validità

La Cairo Virtual Machine [39] è pensata per facilitare la costruzione di prove STARK. Il linguaggio Cairo permette di descrivere la computazione come un programma e non direttamente come un circuito [6]. Ciò viene realizzato utilizzando un sistema di equazioni polinomiali (AIR) che rappresentano una singola computazione: il ciclo FDE di un'architettura di von Neumann. Il numero di vincoli è così fissato e indipendente dal tipo di computazione, permettendo di avere un solo Verificatore per ogni programma.

L'insieme di istruzioni viene definito come un *Algebraic RISC*, in cui le operazioni di addizione e moltiplicazione vengono effettuate su un campo finito primo, la verifica dell'uguaglianza di due valori è supportata ma non il confronto se un valore è minore dell'altro. Questo trade-off nasce da una valutazione attorno al fatto che aggiungere

[6] Come avviene ad esempio con il linguaggio Circom.

un'istruzione in più all'insieme di istruzioni aumenta la complessità dell'esecuzione di un singolo step ma può ridurre il numero di step di un programma.

L'architettura possiede tre registri:

- **ap**: `allocation pointer`, punta alla prima cella di memoria non utilizzata.
- **fp**: `frame pointer`, punta al frame della funzione corrente. L'indirizzo in memoria degli argomenti della funzione e le variabili locali sono relativi a questo valore.
- **pc**: `program counter`, punta all'istruzione corrente.

Per aumentare l'efficienza dei programmi viene fatto uso della *programmazione nondeterministica*. Supponiamo di voler calcolare la radice quadrata di un certo numero x: l'approccio deterministico consiste nell'usare un algoritmo che calcola $y = \sqrt{x}$ e di conseguenza includere la computazione nella prova; nell'approccio nondeterministico il Prover calcola $y = \sqrt{x}$ con lo stesso algoritmo ma senza includerlo nella prova, includendo però la verifica di $y^2 = x$, che può essere svolta con una singola istruzione. Dal punto di vista del Verifier, il valore di y è "indovinato" e l'unico calcolo che viene effettuato è la sua verifica. In questo specifico caso, anche $-y$ è un valore accettato dal Verifier, nonostante la versione deterministica ritorni soltanto y. Queste istruzioni nondeterministiche vengono chiamate *hint*, e siccome sono eseguite soltanto dal Prover possono essere scritte in qualsiasi linguaggio. L'attuale implementazione del linguaggio Cairo usa Python per scrivere gli hint.

La memoria è read-only e nondeterministica: il Prover sceglie tutti i valori della memoria, che non possono essere modificati. Un altro requisito, per motivi di efficienza, è l'essere contigua. Per calcolare la radice quadrata di 25, in Cairo si può scrivere:

```
[ap] = 25; ap++
[ap - 1] = [ap] * [ap]; ap++
```

ovvero: la cella puntata da `ap` deve contenere 25, e la cella successiva deve contenere un valore x che soddisfa $25 = x * x$. Il Prover per generare la prova, deve assegnare i valori alla memoria in modo tale che questi vincoli vengano soddisfatti. Questo può essere fatto utilizzando gli hint:

```
[ap] = 25; ap++
%{
    import math
    memory[ap] = int(math.sqrt(memory[ap - 1]))
%}
[ap - 1] = [ap] * [ap]; ap++
```

L'input di un programma è la sua witness, mentre l'output sono i dati da condividere con il Verifier. Nella versione attuale viene rappresentata come un file JSON. Supponendo di avere un programma che dato n calcola l'n-esimo numero di Fibonacci y, se l'output

comprende sia n che y, allora si sta dimostrando al Verifier che y è l'n-esimo numero di Fibonacci, se l'output è soltanto y, si dimostra al Verifier che si conosce un n tale che l'n-esimo numero di Fibonacci è y, mentre se l'output è soltanto n si dimostra di aver calcolato l'n-esimo numero di Fibonacci senza mostrarne il risultato.

Formalmente si definiscono due macchine Cairo, una deterministica, utilizzata dal Prover, e una non deterministica utilizzata dal Verifier. Si fissi un campo primo $\mathbb{F}_P = \mathbb{Z}/P$ e un'estensione finita $\mathbb{F}$ di esso:

Definizione 1 *La macchina Cairo è una funzione che riceve i seguenti input:*

- *un numero di step* $T \in \mathbb{N}$
- *la funzione della memoria* $m : \mathbb{F} \to \mathbb{F}$
- *una sequenza* S *di* $T+1$ *stati* $S_i = (pc_i, ap_i, fp_i) \in \mathbb{F}^3$ *per* $i \in [0, T]$

e l'output è "accetta" o "rifiuta". La macchina accetta se e solo se per ogni i *la transizione di stato dallo stato* i *allo stato* $i+1$ *è valida.*

La decisione se una singola transizione è valida o no dipende solo dai due stati S_i e S_{i+1} in esame. La funzione della memoria m, siccome nella pratica $|\mathbb{F}|$ è molto grande, può essere vista come una funzione sparsa in cui quasi tutti i valori sono zero.

Definizione 2 *La macchina Cairo non deterministica riceve i seguenti input:*

- *un numero di step* $T \in \mathbb{N}$
- *la funzione parziale della memoria* $m^* : A^* \to \mathbb{F}$*, dove* $A^* \subseteq \mathbb{F}_P$
- *valori iniziali e finali dei registri* `pc`*,* `ap` *e* `fp`*.*

e l'output è "accetta" o "rifiuta". La macchina accetta se e solo se esiste una funzione della memoria $m : \mathbb{F} \to \mathbb{F}$ *che estende* m^* *e una lista di stati* $S_i \in \mathbb{F}^3$ *per* $i \in [0, T]$ *tale che lo stato iniziale e finale corrispondono ai valori in input e che una macchina Cairo deterministica accetta.*

Calcolare se la versione deterministica accetta o no un input può essere calcolato in tempo polinomiale da una macchina deterministica, mentre la versione non deterministica necessita una macchina non deterministica per essere calcolato in tempo polinomiale.

Il bytecode di un programma è una sequenza di elementi del campo $b = (b_0, \ldots, b_{|b|-1})$ e due indici $\text{prog}_{\text{start}}$, $\text{prog}_{\text{end}} \in [0, |b|]$. Per eseguire il programma, si sceglie un indice $\text{prog}_{\text{base}} \in \mathbb{F}$, si imposta la funzione parziale della memoria m^* tale che $\forall i \in [0, |b|] : m^*(\text{prog}_{\text{base}} + i) = b_i$ e si assegna il valore iniziale di `pc` a $\text{prog}_{\text{base}} + \text{prog}_{\text{start}}$ e quello finale

a $\text{prog}_{\text{base}} + \text{prog}_{\text{end}}$. Inoltre la funzione parziale potrebbe contenere altre assegnazioni che aggiungono ulteriori vincoli, come ad esempio gli argomenti in input.

L'esecutore Cairo è responsabile dell'esecuzione di questi programmi. La differenza principale tra eseguire un normale programma e un programma in Cairo è che quest'ultimo, come visto in precedenza, permette istruzioni non deterministiche. L'esecutore Cairo utilizza gli hint per dedurre questi valori. L'output dell'esecutore consiste in:

- un input che la macchina Cairo non deterministica accetta:

$$(T, m^*, \texttt{pc}_I, \texttt{pc}_F, \texttt{ap}_I, \texttt{ap}_F)$$

  dove m^* contiene il bytecode del programma, informazioni aggiuntive come gli hint, $\texttt{pc}_I = \text{prog}_{\text{base}} + \text{prog}_{\text{start}}$ e $\texttt{pc}_F = \text{prog}_{\text{base}} + \text{prog}_{\text{end}}$.

- un input che la macchina Cairo deterministica accetta:

$$(T, m, S)$$

L'esecutore fallisce se l'esecuzione risulta in una contraddizione o se non riesce a calcolare alcuni valori a causa di hint insufficienti. Il dimostratore STARK utilizza l'input deterministico per generare una dimostrazione che l'input non deterministico viene accettato dalla macchina Cairo non deterministica.

StarkNet aggrega più transazioni in un'unica prova STARK utilizzando un dimostratore condiviso che prende il nome di SHARP. Le prove vengono inviate ad uno smart contract su Ethereum, che ne verifica la validità e aggiorna la radice di Merkle corrispondente al nuovo stato. Il costo sub-lineare di verificare una prova di validità permette di ammortizzarne il costo su più transazioni.

Capitolo 4

Confronto

4.1 Tempo di prelievo

L'aspetto più importante che distingue i Rollup Ottimistici dai Rollup di Validità è il tempo che trascorre tra l'inizializzazione di un prelievo e la sua finalizzazione.

In entrambi i casi i prelievi vengono inizializzati su L2 e finalizzati su L1. Su StarkNet la finalizzazione è possibile appena la prova di validità della radice dello stato viene accettata su Ethereum: a livello teorico, è possibile prelevare i fondi nel primo blocco di L1 successivo all'inizializzazione, in seguito alla verifica della validità. In pratica, la frequenza dell'invio di prove di validità su Ethereum è un compromesso tra la velocità di finalizzazione dei blocchi e aggregazione delle prove. Attualmente StarkNet fornisce le prove di validità per la verifica ogni 10 ore [33], ma questo intervallo verrà diminuito all'aumentare delle transazioni.

Su Optimism Bedrock è possibile finalizzare il prelievo soltanto al termine del periodo di disputa (attualmente di 7 giorni), al termine del quale una radice viene considerata automaticamente valida. La lunghezza di questo periodo è determinata dal fatto che la prova d'invalidità può venire censurata su Ethereum fino al suo termine. La probabilità successo di questo tipo di attacco decresce esponenzialmente all'aumentare del tempo:

$$\mathbb{E}[\text{valore sottratto}] = V * p^n$$

dove n è il numero di blocchi in un intervallo, V è la quantità di fondi che è possibile sottrarre pubblicando una radice invalida e p è la probabilità di effettuare con successo un attacco censura in un dato blocco. Supponiamo che questa probabilità sia 99%, che il valore racchiuso nel Rollup sia un milione di Ether e che i blocchi in un intervallo siano 1800 (6 ore di blocchi con intervallo di 12 secondi): il valore atteso è circa 0.01391 Ether. Il sistema viene reso sicuro chiedendo ai propositori di nuove radici di mettere in stake una quantità di Ether molto maggiore rispetto al valore atteso.

Winzer et al. mostrano come realizzare un attacco censura utilizzando un semplice smart contract che assicura che delle certe aree di memoria nello stato non cambino [91].

Modellando l'attacco come un gioco di Markov, il paper dimostra che censurare è la strategia dominante per un produttore di blocchi razionale se egli riceve una compensazione maggiore rispetto ad includere la transazione che modifica la memoria. Il valore p discusso poco sopra può essere visto come la percentuale di produttori di blocchi razionali nella rete, dove per "razionale" si intende che segue delle strategie di profitto sul breve termine senza tenere però conto di esternalità possibilmente penalizzanti, come ad esempio una minor fiducia nella blockchain che ne diminuisce il prezzo della criptovaluta.

```
function claimBribe(bytes memory storageProof) external {
    require(!claimed[block.number], "bribe already claimed");
    OutputProposal memory current = storageOracle.getStorage(L2_ORACLE,
        block.number, SLOT, storageProof);
    require(invalidOutputRoot == current.outputRoot, "attack failed");
    (bool sent, ) = block.coinbase.call{value: address(this).balance /
       period}("");
    require(sent, "failed to send ether");
}
```

Listing 4.1: Esempio di contratto che incentiva un attacco censura a Bedrock.

La lunghezza del periodo di disputa deve anche tenere conto del fatto che la prova d'invalidità è una dimostrazione interattiva e perciò deve essere garantito abbastanza tempo ai partecipanti per interagire e che ogni interazione può venire censurata. Se l'ultima mossa avviene in un momento molto vicino al termine del periodo di disputa, il costo della censura è in maniera significativa minore.

Nonostante censurare sia la strategia dominante, la probabilità di successo diminuisce in quanto i nodi che censurano sono vulnerabili ad attacchi Denial of Service (DoS): un attaccante può generare transazioni molto complesse che terminano con la pubblicazione di una prova d'invalidità (facendo fallire l'attacco censura) a costo zero, in quanto nessuna commissione verrebbe pagata [46]. A causa del problema della fermata non esiste nessuno strumento che può determinare il risultato di una transazione senza eseguirla.

In casi estremi, un lungo periodo di disputa permette di coordinarsi in caso di un attacco censura di successo per organizzare un soft fork ed escludere i produttori di blocchi attaccanti. Questa difesa ha più efficacia utilizzando un algoritmo di consenso PoS che consente di effettuare uno *slash* sullo stake dei propositori di blocchi.

Un altro subdolo attacco, da cui StarkNet e gli altri Rollup di Validità sono immuni, è pubblicare più proposte di radici di stato di quelle che i disputatori riescono a verificare. Ciò può essere evitato utilizzando un limite di frequenza.

4.1.1 Prelievi ottimistici veloci

Siccome la validità di un Rollup Ottimistico può essere verificata in qualunque momento da chiunque, un oracolo fidato può essere utilizzato per conoscere su L1 se il prelievo

può essere finalizzato in sicurezza. Questo meccanismo è stato proposta la prima volta da Maker [61]: un oracolo verifica il prelievo, pubblica il risultato su L1 sul quale viene assegnato all'utente interessato un prestito con interessi che viene automaticamente chiuso al termine dei 7 giorni, ovvero quando il prelievo può essere effettivamente finalizzato. Questa soluzione introduce un'assunzione di fiducia, ma nel caso di Maker viene minimizzata in quanto l'operatore dell'oracolo è gestito dalla stessa organizzazione che si assume il rischio fornendo il prestito.

4.2 Ricorsione: L3 e oltre

Ad alto livello, una dimostrazione di validità può essere vista come:

1. Il Dimostratore genera la prova:

$$P(A, x, y, w, T) = \pi$$

   dove A è un programma, x è l'input, y è l'output, w è la traccia dell'esecuzione del programma A su input x con limite di tempo T. Nel caso di STARK, come visto in 3.1.6, il tempo per generare π è $O(T \cdot \text{poly}\log(T))$.

2. Il Verifier verifica la prova:

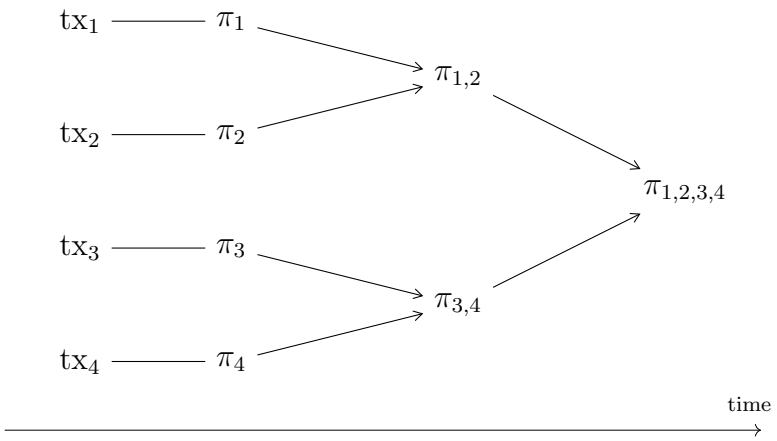
$$V(A, x, y, T, \pi) \in \{\text{VERO}, \text{FALSO}\}$$

   nel caso di STARK, il tempo di verifica è $O(\text{poly}\log(T))$.

Siccome è possibile generare prove per computazioni di programmi arbitrari e la verifica di una prova è una di queste, è possibile generare ricorsivamente la prova della verifica di una prova:

$$P(V, (A, x, y, T, \pi), \text{VERO}, w_1, T_1)$$

SHARP è in grado di aggregare centinaia di migliaia di transazioni in una singola prova, dove il collo di bottiglia è determinato dalle risorse computazionali, come la memoria. In più, se si vogliono aggregare un alto numero di transazioni bisogna attendere l'arrivo dell'ultima transazione per iniziare la generazione della prova. Un sistema di prova ricorsivo è in grado di risolvere questi problemi costruendo un albero binario di prove: le transazioni sono dimostrabili in parallelo, evitando l'attesa dell'ultima e il consumo eccessivo di risorse su un singolo dimostratore, e il tempo per ottenere la prova finale è strettamente minore del metodo standard.



Sia t il tempo utilizzato per dimostrare una transazione e k il numero di transazioni, il sistema di dimostrazione standard impiega circa kt tempo per creare una dimostrazione aggregata. Asintoticamente, il tempo per creare una prova della verifica delle transazioni è minore del tempo delle prove originali:

$$O(T \cdot \text{poly}\log(T)) \supset O(2 \cdot \text{poly}\log(T) \cdot \text{poly}\log(2 \cdot \text{poly}\log(T)))$$

e ciò è vero anche nelle specifiche istanze del problema fino al raggiungimento del limite imposto dall'overhead [1]. Supponendo che la ricorsione si fermi prima di questo limite, il tempo di prova t_i utilizzato al passo i è maggiore di quello al passo $i+1$:

$$t_1 > t_2 > t_3 > \cdots > t_n$$

da cui segue che

$$n \cdot t_1 > t_1 + t_2 + t_3 + \cdots + t_n$$

ovvero $n \cdot t_1$ è un limite superiore del tempo totale di prova. Il numero di passi della ricorsione è al massimo l'altezza dell'albero binario: date k transazioni esso è circa $\log(k)$. Siccome $t_1 = t$, per $k > 0$ e $t \geq 0$ vale:

$$\log(k) \cdot t \leq kt$$

Utilizzando questo meccanismo il Verificatore finale deve verificare soltanto prove ricorsive invece di programmi arbitrari, riducendone la complessità.

Sviluppando un Verificatore SHARP in Cairo si può sfruttare questo meccanismo per la costruzione di un L3, ovvero un ulteriore Rollup che si appoggia a StarkNet. La latenza della finalizzazione su Ethereum non viene incrementata significativamente, e la pubblicazione dei dati e la dimostrazione delle prove è meno costoso in quanto si interagisce con L2. Il vantaggio principale di un L3 è la possibilità di costruire Rollup

[1] l'applicazione ricorsiva di un logaritmo sommato ad una costante converge.

per applicazioni specifiche o con tecnologie sperimentali, senza la necessità di un elevato numero di transazioni per ammortizzare i costi.

A differenza dei Rollup di Validità, costruire un Rollup Ottimistico al di sopra di un altro Rollup Ottimistico non porta a molti vantaggi: il periodo di finalizzazione di un L3 si somma a quello di L2, nel caso di una istanza Optimism Bedrock basata sulla principale diventerebbe di 14 giorni. Siccome le chiamate di funzioni vengono pubblicate in versione compressa su Ethereum e un eventuale L3 dovrebbe fare lo stesso su L2, non si ottiene nessun vantaggio ad applicare due volte la compressione sulle stringhe.

4.3 Costo delle transazioni

Il costo delle transazioni su L2 è per la maggior parte determinato dall'interazione con L1 mentre il costo di ciò che viene consumato off-chain è perlopiù trascurabile: ad esempio il prezzo del gas su Optimism è stabile nell'ordine del millesimo di Gwei [57]. StarkNet pubblica su Ethereum sotto forma di calldata ogni cambiamento dello storage di L2 ed esegue le verifiche delle dimostrazioni di validità, mentre Optimism pubblica sotto forma di calldata compresso tutti i calldata delle transazioni del Rollup e molto raramente esegue le prove di invalidità [2]. In entrambe le soluzioni il costo della computazione è molto economico in quanto eseguito interamente off-chain, ma per StarkNet lo storage usato dalle transazioni è la risorsa con il costo più elevato, mentre per Optimism lo è il calldata.

Come definito nel Yellow Paper di Ethereum e in EIP-2028 [3], il costo del calldata su Ethereum per un byte rappresentante lo zero è 4 gas e 16 per byte non zero. Il calcolo del costo in gas della scrittura in storage su Ethereum dipende se la cella ha già avuto un accesso (`SLOAD`) oppure no, come riportato di seguito:

OPCODE	**con accesso**	**senza accesso**
`SSTORE` da zero a non-zero	22100	20000
`SSTORE` da non-zero a non-zero	5000	2900
`SSTORE` a byte zero	costo val. precedente + rimborso	costo val. precedente + rimborso
`SSTORE` di un val. modificato	100	100

Tabella 4.1: Costo della modifica di una cella dello storage. Il calcolo del rimborso è oltre lo scopo di questa sezione.

[2] se il modello economico funziona correttamente e tutti i partecipanti sono razionali, non si ha mai il bisogno di pubblicarle.

StarkNet pubblica i dati delle differenze di storage nel formato descritto in 3.2.3. L'esatto costo del calldata rappresentate le differenze dipende dai valori particolari delle celle e dai valori che vengono scritti. Supponendo di non pubblicare alcun contratto e di modificare su StarkNet 10 celle non precedentemente accedute nella maniera descritta in Appendice 5 si calcola un costo di 9240 gas rispetto ai 221000 se lo storage fosse stato modificato su Ethereum, ovvero circa il 4.18% [3]. Se una cella viene sovrascritta n volte tra due pubblicazioni dei dati, il costo di ogni scrittura sarà $1/n$ rispetto al costo di una singola in quanto viene pubblicata soltanto l'ultima modifica. Il costo può essere ulteriormente minimizzato comprimendo valori utilizzati di frequente. Il costo della verifica della prova di validità viene diviso tra le transazioni a cui fa riferimento: ad esempio, il blocco 4779 di StarkNet contiene 200 transazioni e la sua prova di validità consuma 267830 gas, ovvero 1339.15 gas per ognuna.

Optimism pubblica i dati del calldata delle transazioni di L2 su L1 nel formato descritto in 2.2.3. Al momento non è ancora attiva una rete che utilizza la versione Bedrock, ma grazie all'equivalenza con la EVM è possibile testare la compressione utilizzando i blocchi di L1: supponendo di utilizzare le transazioni nel blocco di Ethereum ad altezza 13100000 (29904951 gas) per costruire un batch, la compressione con ZLIB riduce di 51.69% la dimensione in byte del calldata e del 79.74% il costo in gas (da 1323384 a 1055548). La differenza tra i due rapporti avviene a causa delle lunghe sequenze di zeri. Il rapporto di compressione può essere ulteriormente migliorato comprimendo più batch insieme: utilizzando i blocchi da 13100000 a 13100009 si ottiene una compressione in byte del 51.02% e in gas del 77.10% [4]. Supponendo il prezzo del gas su Optimism a 0.001 Gwei e il prezzo del gas su Ethereum a 10 Gwei, il costo per elaborare il blocco 13100000 su Optimism è circa il 3.5% rispetto a se fosse eseguito su Ethereum.

4.3.1 Ottimizzare il calldata: contratto cache

Quando si progetta un contratto su Optimism bisogna tenere in mente il fatto che il calldata è la risorsa più costosa. Di seguito viene presentato uno smart contract che implementa una cache per gli indirizzi che sfrutta il fatto che lo storage e l'esecuzione sono risorse molto meno costose insieme ad un contratto `Friends` che ne dimostra l'utilizzo. Quest'ultimo tiene traccia degli "amici" di un indirizzo che possono essere registrati chiamando la funzione `addFriend`. Se un address è già stato utilizzato almeno una volta può essere aggiunto chiamando la funzione `addFriendWithCache`: gli indici della cache sono interi a 4 byte mentre gli indirizzi sono rappresentati da 120 byte, perciò si ottiene un risparmio di circa 96.66% sull'argomento della funzione. La stessa logica può essere utilizzata per altri tipi di dati come interi o più in generale byte.

[3] il codice utilizzato per la stima di StarkNet è visitabile al link `https://github.com/lucadonnoh/starknet-data-availability-cost`.

[4] il codice utilizzato per la stima di Optimism è visitabile al link `https://github.com/lucadonnoh/optimism-data-availability-cost`.

```
contract AddressCache {
    mapping(address => uint32) public address2key;
    address[] public key2address;

    function cacheWrite(address _address) internal returns (uint32) {
        require(key2address.length < type(uint32).max, "AddressCache:
            cache is full");
        require(address2key[_address] == 0, "AddressCache: address
            already cached");
        // keys must start from 1 because 0 means "not found"
        uint32 key = uint32(key2address.length + 1);
        address2key[_address] = key;
        key2address.push(_address);
        return key;
    }

    function cacheRead(uint32 _key) public view returns (address) {
        require(_key <= key2address.length && _key > 0, "AddressCache:
            key not found");
        return key2address[_key - 1];
    }
}
```

Listing 4.2: Contratto cache per gli indirizzi.

```
contract Friends is AddressCache {
    mapping(address => address[]) public friends;

    function addFriend(address _friend) public {
        friends[msg.sender].push(_friend);
        cacheWrite(_friend);
    }

    function addFriendWithCache(uint32 _friendKey) public {
        friends[msg.sender].push(cacheRead(_friendKey));
    }

    function getFriends() public view returns (address[] memory) {
        return friends[msg.sender];
    }
}
```

Listing 4.3: Esempio di contratto che eredita la cache di indirizzi.

Il contratto supporta in cache circa 4 miliardi di indirizzi (2^{32}) e aggiungendo un solo byte se ne ottengono circa mille miliardi (2^{40}).

4.3.2 Ottimizzare lo storage: filtri di Bloom

Su StarkNet ci sono diverse tecniche per minimizzare l'utilizzo dello storage. Se non è necessario garantire la disponibilità del dato originale allora è sufficiente salvare l'hash on-chain: questo è il meccanismo usuale utilizzato per salvare i dati di un ERC-721 (NFT) [29], ovvero un link IPFS che, se disponibile, risolve l'hash del dato. Per dati che vengono salvati più volte è possibile utilizzare una tabella di look-up simile al sistema di caching introdotto per Optimism richiedendo però di salvare almeno una volta tutti i valori. Per alcuni applicazioni ciò può essere evitato utilizzando un filtro di Bloom [17] [23] [2], ovvero una struttura dati probabilistica che permette di sapere con certezza se un elemento non appartiene ad un insieme ma ammette una piccola ma non trascurabile probabilità di falsi positivi.

Un filtro di Bloom viene inizializzato come un array di m bit a zero. Per aggiungere un elemento si utilizzano k funzioni di hash con una distribuzione casuale uniforme, ognuna che mappa ad un bit dell'array che viene impostato ad 1. Per verificare se un elemento appartiene all'insieme si eseguono le k funzioni di hash e si verificano che i k bit siano impostati a 1. In un filtro semplice di Bloom non c'è nessun modo per distinguere se un elemento appartiene effettivamente all'insieme o è un falso positivo, probabilità che cresce all'aumentare degli inserimenti. Dopo aver inserito n elementi:

$$\mathbb{P}[\text{falso positivo}] = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

assumendo l'indipendenza della probabilità di ogni bit impostato. Se ci si aspetta di inserire n elementi (di dimensione arbitraria!) e la probabilità di un falso positivo tollerata è p, la dimensione dell'array può essere calcolata come:

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

mentre il numero ottimale di funzioni di hash è:

$$k = \frac{m}{n} \ln 2$$

Se si suppone di inserire 1000 elementi con una tolleranza del 1% la dimensione dell'array è 9585 bit con $k = 6$, mentre per una tolleranza del 0.1% diventa 14377 con $k = 9$. Se ci si aspetta di inserire un milione di elementi, la dimensione dell'array diventa circa 1170 kB per 1% e 1775 kB per 0.1%, con gli stessi valori di k, in quanto dipende solo da p [87].

In un gioco in cui si implementa un sistema in cui si vuole che ai giocatori non venga assegnato un avversario già sfidato, invece di salvare in storage per ogni giocatore la lista di avversari passati si può utilizzare un filtro di Bloom. Il rischio di non scontrare qualche giocatore è spesso accettabile e il filtro può essere resettato periodicamente. Una implementazione di un filtro di Bloom in Cairo è stata sviluppata da Sam Barnes [10].

4.4 Compatibilità con Ethereum

Il vantaggio principale dell'essere compatibile con la EVM ed Ethereum è il riutilizzo di tutti gli strumenti. Gli smart contract di Ethereum possono essere pubblicati su Optimism senza alcuna modifica e nuovi audit: sono infatti già attive le maggiori applicazioni come Synthetix, Uniswap, Aave e Curve. I wallet rimangono compatibili, gli strumenti di sviluppo e di analisi statica, gli strumenti di analisi, di indicizzazione e gli oracoli. Ethereum e Solidity hanno una lunga storia di vulnerabilità ben studiate, come attacchi di rientranza, overflow e underflow, flash loan e manipolazioni di oracoli che son stati causa di attacchi dove decine o migliaia di milioni di dollari sono stati rubati [42] [52]. Grazie a ciò Optimism è riuscito a catturare una grande quantità di valore in poco tempo.

Scegliere di adottare una differente macchina virtuale implica dover ricostruire un intero ecosistema ma con il vantaggio di avere una maggiore libertà implementativa. StarkNet implementa nativamente l'*account abstraction*, ovvero un meccanismo per cui ogni account è uno smart contract che può implementare una logica arbitraria a patto che rispetti un'interfaccia (da qui il termine *abstraction*): ciò permette di utilizzare diversi schemi di firma digitale, di poter cambiare la chiave privata utilizzando lo stesso indirizzo o utilizzare un multisig. La comunità di Ethereum ha proposto l'introduzione di questo meccanismo con EIP-2938 nel 2020 ma la proposta è rimasta indiscussa da più di un anno in quanto altri aggiornamenti hanno avuto più priorità [21].

Un'altra importante vantaggio ottenuto dalla compatibilità è il riutilizzo dei client esistenti: Optimism utilizza una versione di geth per il proprio nodo con sole 800 righe di differenza, il quale è sviluppato, testato e mantenuto dal 2014. Avere un client robusto è cruciale in quanto esso definisce ciò che viene accettato come valido oppure no nella rete. Un bug nell'implementazione del sistema di prove d'invalidità potrebbe far accettare come corretta una prova d'invalidità non corretta o come non corretta una prova d'invalidità per un blocco non valido, compromettendo il sistema. La probabilità di questo tipo di attacco può essere limitata con una più ampia diversità di client: Optimism può riutilizzare oltre a geth gli altri client di Ethereum già mantenuti ed è già in corso lo sviluppo di un altro client basato su Erigon. Nel 2016 un problema nella gestione della memoria di geth è stato sfruttato per un attacco DoS e la prima linea di difesa è stata consigliare l'uso di Parity, il secondo client più utilizzato al momento. StarkNet presenta lo stesso problema con le prove di validità, ma i client devono essere scritti da zero e il sistema di dimostrazione è molto più complesso e di conseguenza è anche molto più complesso garantirne la correttezza. Attualmente StarkNet possiede soltanto un client e non sono in corso lo sviluppo di altri.

4.5 Licenza d'uso

StarkNet è stato spesso al centro dell'attenzione a causa della licenza restrittiva. Il linguaggio e toolchain Cairo adottano la Cairo Toolchain License [84]: essa permette l'uso di Cairo per la scrittura e compilazione di programmi Cairo e altri usi non commerciali come ricerca accademica e scientifica. La toolchain è modificabile soltanto per il fix di errori e non per introdurre nuove funzionalità e non è autorizzata la copia e la distribuzione del codice (che è pubblicamente disponibile).

Il codice del Prover, attualmente closed source, verrà rilasciato con licenza StarkWare Polaris Prover License [85] che permette l'utilizzo commerciale soltanto per generare prove che vengono inviate ad un Polaris Verifier, ovvero un Verifier approvato da StarkWare. La lista di Polaris Verifier approvati può solo essere estesa, perciò un Verifier approvato non può essere revocato. Ciò previene in maniera effettiva la creazione di fork indipendenti da StarkWare.

Optimism, al contrario, è completamente open source e utilizza una licenza MIT per tutti gli strumenti utilizzati. Ciò ha portato alla creazione di due fork, Metis Andromeda [76] e Boba Network [69] che non hanno però compromesso il successo della versione originale. Ciò è già stato visto con Bitcoin e soprattutto Ethereum, che presentano numerosi fork che non hanno fatto altro che legittimare la tecnologia e portato più sviluppatori a verificare ed espandere il codice originale. Attualmente la EVM è la macchina virtuale più utilizzata per le blockchain e Solidity il linguaggio più diffuso per la scrittura di smart contract, e Optimism punta nella stessa maniera a diventare uno standard di fatto per la costruzione di Rollup.

Capitolo 5

Conclusione

I Rollup sono la soluzione più promettente a disposizione oggi per risolvere il problema della scalabilità nelle blockchain decentralizzate, dando inizio all'era delle blockchain modulari in opposizione alle blockchain monolitiche.

La scelta di sviluppare un Rollup Ottimistico o un Rollup di Validità si mostra principalmente come un trade-off tra complessità e agilità. StarkNet presenta numerosi vantaggi come velocità di prelievo potenzialmente istantanea, impossibilità strutturale di avere transizioni di stato invalide, ricorsione e minor costo delle transazioni a discapito di un più lungo periodo di sviluppo e incompatibilità con la EVM, dove invece Optimism ha sfruttato l'economia di rete per ottenere velocemente una importante fetta del mercato.

Optimism Bedrock possiede però un design modulare che gli permette in futuro di diventare un Rollup di Validità: Cannon attualmente utilizza `minigeth` compilato a MIPS per la disputa delle prove d'invalidità, ma la stessa architettura può essere usata per ottenerne un circuito e produrre prove di validità. Compilare una macchina complessa come la EVM per una microarchitettura permette di ottenere un circuito più semplice e che non necessita di modifiche e nuove verifiche in caso di aggiornamenti. RISC Zero è una microarchitettura verificabile con prove STARK già in sviluppo basata su RISC-V che può essere usata a questo scopo in alternativa a MIPS [77].

Un aspetto da non sottovalutare è la complessità nel capire il funzionamento della tecnologia. Un punto di forza delle blockchain tradizionali è il riuscire a verificare lo stato della blockchain senza fidarsi di nessun ente terzo, ma nel caso di StarkNet anche se si utilizza un proprio nodo bisogna fidarsi dell'implementazione se non si è grado di verificare i vari componenti basati su crittografia e matematica avanzata. Ciò può inizialmente creare frizione per l'adozione della tecnologia, ma con l'avanzare degli strumenti e dell'adozione delle prove di integrità anche al di fuori del campo delle blockchain questo problema verrà risolto.

Appendice

Valore calldata di StarkNet

[0, 1, 780129873670784982447369675874412763760142061544058579488225814081041044107 21,
10, 494378874472551056171993858879801295902990434109063998972743396866643805749 60,
816131961448629539307552844120134857538259427258882219150120796517921101038 08,
778698456722451216622375469368981950776859707744005289457906347504863999862 45,
855582862946510181192823559337725237995657897574864694368702337412006017209 03,
907454391127999952806739589633198098410919025736309032946556089529112375106 38,
49, 720637046056882137158723760715143116893166152703846623748271754214828801251 80,
390479362961554678915233061147509724108989888105591289887439267463348393892 54,
898212066715393192799951976954292641231754933983198048425751997281811152525 99,
99, 474757530469111647376719505791720754231873361106537491064972192816565443668 08,
29, 305944998118728275451532579931741471777461630038346286452396079853598431082 05,
16, 212300457440899191952618616610204169448481949565279986808809530290668972194 08,
279415550595590981415673486269881650988863094755754947109990322361781143175 93,
835497333184104796148204451663912820867505262407909175550623545005458693802 30,
17, 701999795741901033933259737975669288854606559067092933787131003382076281380 06,
370220555333743621864823051105821363111032967027114647104947901850273177159 2]

ovvero:

- Celle dedicate alla pubblicazione di contratti: 0
- Numero di contratti con celle modificate: 1
- Indirizzo del contratto
- Numero di celle modificate: 10
- Lista delle celle modificate (chiave, valore), di cui 5 con valore piccoli (≤ 100) e 5 con valori alti (fino a $2^{256} - 1$).

Bibliografia

[1] John Adler e Mikerah Quintyne-Collins. «Building scalable decentralized payment systems». In: *arXiv preprint arXiv:1904.06441* (2019).

[2] Sachin Agarwal e Ari Trachtenberg. «Approximating the number of differences between remote sets». In: *2006 IEEE Information Theory Workshop-ITW'06 Punta del Este*. IEEE. 2006, pp. 217–221.

[3] Alexey Akhunov et al. *EIP-2028: Transaction data gas cost reduction*. 2019. URL: `https://eips.ethereum.org/EIPS/eip-2028`.

[4] Josh Alman e Virginia Vassilevska Williams. «A refined laser method and faster matrix multiplication». In: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM. 2021, pp. 522–539.

[5] Gal Arnon, Alessandro Chiesa e Eylon Yogev. «A PCP Theorem for Interactive Proofs and Applications». In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2022, pp. 64–94.

[6] Georgia Avarikioti et al. «Towards secure and efficient payment channels». In: *arXiv preprint arXiv:1811.12740* (2018).

[7] László Babai. «Trading group theory for randomness». In: *Proceedings of the seventeenth annual ACM symposium on Theory of computing*. 1985, pp. 421–429.

[8] László Babai, Lance Fortnow e Carsten Lund. «Non-deterministic exponential time has two-prover interactive protocols». In: *Computational complexity* 1.1 (1991), pp. 3–40.

[9] Johann Barbie. *Why Smart Contracts are NOT feasible on Plasma*. 2018. URL: `https://ethresear.ch/t/why-smart-contracts-are-not-feasible-on-plasma/2598`.

[10] Sam Barnes. *cairo-bloom*. 2022. URL: `https://github.com/sambarnes/cairo-bloom`.

[11] Mustafa Al-Bassam, Alberto Sonnino e Vitalik Buterin. «Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities». In: *arXiv preprint arXiv:1809.09044* 160 (2018).

[12] Mihir Bellare e Adriana Palacio. «The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols». In: *Annual International Cryptology Conference*. Springer. 2004, pp. 273–289.

[13] Eli Ben-Sasson et al. «Computational integrity with a public random string from quasi-linear PCPs». In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2017, pp. 551–579.

[14] Eli Ben-Sasson et al. «Scalable, transparent, and post-quantum secure computational integrity». In: *Cryptology ePrint Archive* (2018).

[15] Eli Ben-Sasson et al. «Secure sampling of public parameters for succinct zero knowledge proofs». In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 287–304.

[16] Blockchair. *Bitcoin TPS*. URL: `https://blockchair.com/bitcoin/charts/transactions-per-second`.

[17] Burton H Bloom. «Space/time trade-offs in hash coding with allowable errors». In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[18] Vitalik Buterin. *The Cost of Internet of Money*. 2022. URL: `https://twitter.com/vitalikbuterin/status/1477402773998247944`.

[19] Vitalik Buterin. *The different types of ZK-EVMs*. 2022. URL: `https://vitalik.ca/general/2022/08/04/zkevm.html`.

[20] Vitalik Buterin et al. *EIP-1559: Fee market change for ETH 1.0 chain*. 2019.

[21] Vitalik Buterin et al. *EIP-2938: Account Abstraction*. 2020. URL: `https://eips.ethereum.org/EIPS/eip-2938`.

[22] Usman W Chohan. «The double spending problem and cryptocurrencies». In: *Available at SSRN 3090174* (2021).

[23] Ken Christensen, Allen Roginsky e Miguel Jimeno. «A new analysis of the false positive rate of a bloom filter». In: *Information Processing Letters* 110.21 (2010), pp. 944–949.

[24] Ivan Damgård. «Towards practical public key systems secure against chosen ciphertext attacks». In: *Annual International Cryptology Conference*. Springer. 1991, pp. 445–456.

[25] Sergi Delgado-Segura et al. «Analysis of the bitcoin utxo set». In: *International Conference on Financial Cryptography and Data Security*. Springer. 2018, pp. 78–91.

[26] P. Deutsch e J-L. Gailly. *ZLIB Compressed Data Format Specification version 3.3*. RFC 1950. RFC Editor, mag. 1996, pp. 1–10. URL: `https://www.rfc-editor.org/rfc/rfc1950.html`.

[27] Ratna Dutta, Rana Barua e Palash Sarkar. «Pairing-based cryptographic protocols: A survey». In: *Cryptology ePrint Archive* (2004).

[28] Peter Elias. «Error-free coding». In: (1954).

[29] William Entriken et al. *EIP-721: Non-Fungible Token Standard*. 2018. URL: https://eips.ethereum.org/EIPS/eip-721.

[30] Etherscan. *Average Daily Transaction Fee*. 2022. URL: https://etherscan.io/chart/avg-txfee-usd.

[31] Etherscan. *Ethereum Gas Price Chart*. 2022. URL: https://etherscan.io/chart/gasprice.

[32] Etherscan. *Ethereum Network Utilization Chart*. 2022. URL: https://etherscan.io/chart/networkutilization.

[33] Etherscan. *StarkNet Core Contract*. 2022. URL: https://etherscan.io/address/0xc662c410C0ECf747543f5bA90660f6ABeBD9C8c4.

[34] Amos Fiat e Adi Shamir. «How to prove yourself: Practical solutions to identification and signature problems». In: *Conference on the theory and application of cryptographic techniques*. Springer. 1986, pp. 186–194.

[35] Solana Foundation. *Solana Validator Requirements*. 2022. URL: https://docs.solana.com/running-validator/validator-reqs#rpc-node-recommendations.

[36] Rusins Freivalds. «Probabilistic Machines Can Use Less Running Time.» In: *IFIP congress*. Vol. 839. 1977, p. 842.

[37] Martin Furer, Oded Goldreich e Yishay Mansour. «On completeness and soundness in interactive proof systems». In: (1989).

[38] Evangelos Georgiadis. «How many transactions per second can bitcoin really handle? Theoretically.» In: *Cryptology ePrint Archive* (2019).

[39] Lior Goldberg, Shahar Papini e Michael Riabzev. «Cairo–a Turing-complete STARK-friendly CPU architecture». In: *Cryptology ePrint Archive* (2021).

[40] Shafi Goldwasser, Silvio Micali e Charles Rackoff. «The knowledge complexity of interactive proof systems». In: *SIAM Journal on computing* 18.1 (1989), pp. 186–208.

[41] Shafi Goldwasser e Michael Sipser. «Private coins versus public coins in interactive proof systems». In: *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. 1986, pp. 59–68.

[42] Emily Graffeo. *DeFi Protocol Cream Finance Loses $130 Million in Latest Crypto Hack*. 2021. URL: https://www.bloomberg.com/news/articles/2021-10-27/defi-protocol-cream-finance-loses-130-million-in-latest-hack.

[43] Jens Groth. «On the size of pairing-based non-interactive arguments». In: *Annual international conference on the theory and applications of cryptographic techniques*. Springer. 2016, pp. 305–326.

[44] Jens Groth. «Short pairing-based non-interactive zero-knowledge arguments». In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2010, pp. 321–340.

[45] Abdelatif Hafid, Abdelhakim Senhaji Hafid e Mustapha Samih. «Scaling blockchains: A comprehensive survey». In: *IEEE Access* 8 (2020), pp. 125244–125262.

[46] Tjaden Hess, River Keefer e Emin Gin Sirer. *Ethereum's DAO Wars Soft Fork is a Potential DoS Vector*. 2016. URL: https://hackingdistributed.com/2016/06/28/ethereum-soft-fork-dos-vector/.

[47] Daira Hopwood et al. «Zcash protocol specification». In: *GitHub: San Francisco, CA, USA* (2016), p. 1.

[48] Markus Jakobsson, Kazue Sako e Russell Impagliazzo. «Designated verifier proofs and their applications». In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 1996, pp. 143–154.

[49] Sandra Johnson, Peter Robinson e John Brainard. «Sidechains and interoperability». In: *arXiv preprint arXiv:1903.04077* (2019).

[50] jshess.eth. *Token Swap during UNI Launch*. 2022. URL: https://etherscan.io/tx/0x6059ffe5a440d09a4ce055cdd9a3cd35470b7771510163a37273eaa754cebe0d.

[51] Georgios Konstantopoulos. «Plasma cash: towards more efficient plasma constructions». In: *arXiv preprint arXiv:1911.12095* (2019).

[52] Daniel Kuhn e Kevin Reynolds. *DeFi Protocol Pickle Finance Token Loses Almost Half Its Value After $19.7M Hack*. 2020. URL: https://www.nasdaq.com/articles/defi-protocol-pickle-finance-token-loses-almost-half-its-value-after-%5C%2419.7m-hack-2020-11.

[53] Eyal Kushilevitz. «Communication complexity». In: *Advances in Computers*. Vol. 44. Elsevier, 1997, pp. 331–360.

[54] L2Beat. *L2 Value Locked*. 2022. URL: https://web.archive.org/web/20220902092817/https://l2beat.com/scaling/tvl/.

[55] Fuel Labs. *Fuel Labs Github Organization*. 2022. URL: https://github.com/FuelLabs.

[56] Off-chain Labs. *Arbitrum Nitro Github Repository*. 2022. URL: https://github.com/OffchainLabs/nitro.

[57] Optimism Labs. *Optimism Public Grafana*. 2022. URL: https://public-grafana.optimism.io/d/9hkhMxn7z/public-dashboard?orgId=1&refresh=5m.

[58] Uniswap Labs. *Introducing UNI*. 2020. URL: https://uniswap.org/blog/uni.

[59] Yuga Labs. *Otherside Website*. URL: https://otherside.xyz.

[60] François Le Gall. «Powers of tensors and fast matrix multiplication». In: *Proceedings of the 39th international symposium on symbolic and algebraic computation*. 2014, pp. 296–303.

[61] Sam MacPherson. *Announcing the Optimism Dai Bridge with Fast Withdrawals*. 2021. URL: https://forum.makerdao.com/t/announcing-the-optimism-dai-bridge-with-fast-withdrawals/6938.

[62] Oleksandr Marukhnenko e Gennady Khalimov. «The Overview of Decentralized Systems Scaling Methods». In: *COMPUTER AND INFORMATION SYSTEMS AND TECHNOLOGIES* (2021).

[63] Patrick McCorry et al. «Towards bitcoin payment networks». In: *Australasian Conference on Information Security and Privacy*. Springer. 2016, pp. 57–76.

[64] Ralph C Merkle. «A digital signature based on a conventional encryption function». In: *Conference on the theory and application of cryptographic techniques*. Springer. 1987, pp. 369–378.

[65] Gianmaria Del Monte, Diego Pennino e Maurizio Pizzonia. «Scaling blockchains without giving up decentralization and security: A solution to the blockchain scalability trilemma». In: *Proceedings of the 3rd Workshop on Cryptocurrencies and Blockchains for Distributed Systems*. 2020, pp. 71–76.

[66] Donald R Morrison. «PATRICIA—practical algorithm to retrieve information coded in alphanumeric». In: *Journal of the ACM (JACM)* 15.4 (1968), pp. 514–534.

[67] Satoshi Nakamoto. «Bitcoin whitepaper». In: *URL: https://bitcoin.org/bitcoin.pdf-(: 17.07. 2019)* (2008).

[68] Lydia D Negka e Georgios P Spathoulas. «Blockchain state channels: A state of the art». In: *IEEE Access* (2021).

[69] Boba Network. *Boba*. 2022. URL: https://github.com/bobanetwork/boba.

[70] Oiler Network. *Fossil*. 2022. URL: https://github.com/OilerNetwork/fossil.

[71] Carlos Pérez et al. *Circuits for zkEVM*. 2022. URL: https://github.com/privacy-scaling-explorations/zkevm-circuits.

[72] John M Pollard. «Monte Carlo methods for index computation». In: *Mathematics of computation* 32.143 (1978), pp. 918–924.

[73] *Polygon Hermez Github Organization*. 2022. URL: https://github.com/0xpolygonhermez.

[74] *Polygon Hermez Github Repository*. 2022. URL: https://github.com/matter-labs/zksync.

[75] Joseph Poon e Vitalik Buterin. «Plasma: Scalable autonomous smart contracts». In: *White paper* (2017), pp. 1–47.

[76] Metis Protocol. *Metis Andromeda*. 2022. URL: https://github.com/MetisProtocol/mvm.

[77] risc0. *Risc Zero*. 2022. URL: https://github.com/risc0/risc0.

[78] Claus-Peter Schnorr. «Efficient identification and signatures for smart cards». In: *Conference on the Theory and Application of Cryptology*. Springer. 1989, pp. 239–252.

[79] Jacob T Schwartz. «Fast probabilistic algorithms for verification of polynomial identities». In: *Journal of the ACM (JACM)* 27.4 (1980), pp. 701–717.

[80] *Scroll Github Organization*. 2022. URL: https://github.com/scroll-tech.

[81] Adi Shamir. «Ip= pspace». In: *Journal of the ACM (JACM)* 39.4 (1992), pp. 869–877.

[82] Peter W Shor. «Algorithms for quantum computation: discrete logarithms and factoring». In: *Proceedings 35th annual symposium on foundations of computer science*. Ieee. 1994, pp. 124–134.

[83] Corwin Smith, Emmanuel Awosika e Sam Richards. *Volitions and Validium*. 2022. URL: https://ethereum.org/en/developers/docs/scaling/validium/#volitions-and-validium.

[84] StarkWare. *Cairo Toolchain License*. 2022. URL: https://github.com/starkware-libs/cairo-lang/blob/master/LICENSE.txt.

[85] StarkWare. *StarkWare Polaris Prover License*. 2021. URL: https://starkware.co/starkware-polaris-prover-license/.

[86] *Starkware Github Organization*. 2022. URL: https://github.com/starkware-libs.

[87] David Starobinski, Ari Trachtenberg e Sachin Agarwal. «Efficient PDA synchronization». In: *IEEE Transactions on Mobile Computing* 2.1 (2003), pp. 40–51.

[88] Mark Tyneway et al. *optimism: Bedrock*. Ver. v0.2.0. Lug. 2022. DOI: 10.5281/zenodo.6894644. URL: https://doi.org/10.5281/zenodo.6894644.

[89] Fabian Vogelsteller e Vitalik Buterin. *EIP-20: Token Standard*. 2015.

[90] Stephen B Wicker e Vijay K Bhargava. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[91] Fredrik Winzer, Benjamin Herd e Sebastian Faust. «Temporary censorship attacks in the presence of rational miners». In: *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE. 2019, pp. 357–366.

[92] Gavin Wood et al. «Ethereum: A secure decentralised generalised transaction ledger». In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[93] xteesy.eth. *Otherside Land Mint Transaction*. 2022. URL: https://etherscan.io/tx/0xf0228cb01218dbfcb42b6c131782ff83c1c28cfd1458fd08a01007df7ae9baa7.

[94] Anatoly Yakovenko. «Solana: A new architecture for a high performance blockchain v0. 8.13». In: *Whitepaper* (2018).

[95] zhenxiang. *BSC TPS*. 2022. URL: https://dune.com/queries/1038929/1791946.

[96] zhenxiang. *Ethereum TPS*. 2022. URL: https://dune.com/queries/1038637/1791423.

[97] zhenxiang. *Solana TPS*. 2022. URL: https://dune.com/queries/1106410/1889494.

[98] Micah Zoltu. *EIP-2718: Typed Transaction Envelope*. 2020.