UNIVERSITÀ DEGLI STUDI DELLA BASILICATA

Corso di Laurea in Scienze e Tecnologie Informatiche

Tesi di Laurea Triennale

Estrazione di informazioni dalla blockchain di BitCoin

Relatore:
Dott. Carlo Sartiani

Candidato:
Vincenzo Palazzo

12 dicembre 2019

All'amato me stesso

Citatemi dicendo che sono stato citato male.
Groucho Marx

Ringraziamenti

Grazie a Grazia, Graziella e la sorella.

Sommario

Piacere, sommario.

Indice

Elenco delle figure

Elenco delle tabelle

Lista dei codici

Capitolo 1

Blockchain

1.1 Introduzione

Il termine *blockchain* denota un particolare tipo di *distributed ledger* protetto mediante meccanismi crittografici. Il distributed ledger è un log distribuito, parzialmente o totalmente replicato a cui accedono un gruppo di peer. All'interno di una blockchain vengono memorizzate in modo permanente e immutabile una serie di informazioni verificate attraverso opportuni algoritmi di consenso.

1.2 Protocolli di consenso

Gli algoritmi di consenso assicurano che il nuovo blocco da pubblicare sulla blockchain sia completamente convalidato e protetto, assicurando anche che le transazioni siano valide; questi algoritmi svolgono un ruolo fondamentale nella risoluzione del problema della doppia spesa. Il protocollo introdotto dalla prima tecnologia blockchain, cioè Bitcoin[1], è l'algoritmo di Proof of Work (PoW) dove, per dimostrare la validità di un blocco e il lavoro svolto, i nodi nella rete (chiamati *miner*) usano potenza computazionale per convalidare le azioni e competono tra loro in una sfida crittografica per trovare una soluzione ad un problema imposto dal protocollo. Il vincitore ha diritto ad una ricompensa per la vincita e, se il protocollo lo prevede, a riscuotere le commissioni incluse nelle transazioni. Il miner vincitore della sfida creerà un nuovo blocco, includendo l'hash del blocco precedente, il timestamp e le transazioni.
Trasmettendo il nuovo blocco sulla rete, si può verificare un fenomeno di concorrenza: due o più nodi possono pubblicare un blocco con lo stesso hash di blocco precedente, ma contenuto completamente o parzialmente differente. Tale fenomeno viene aggirato scegliendo di lavorare con una catena di blocchi più lunga, perché in quel caso il miner avrà svolto maggior lavoro.
Come ogni protocollo, i protocolli di consenso presentano vulnerabilità: in teoria, il protocollo PoW può essere attaccato se un miner, da solo o in un gruppo, possiede più della

[1] Utilizzeremo per tutto il documento la convenzione sulla parola Bitcoin, scritta con la lettera maiuscola indicherà il protocollo Bitcoin, analogamente per bitcoin scritto con la lettera minuscola che indicherà la moneta.

metà della potenza di estrazione totale della rete. Questo è anche noto come *attacco del 51%* in cui gli aggressori creano la propria catena segreta riuscendo ad ottenere la fiducia di tutto il sistema, compromettendone quindi le funzionalità.

1.3 Blocchi

La blockchain può essere definita come una struttura ad albero con zero o al massimo un figlio; ogni blocco contiene il riferimento al blocco precedente, con cui si ottiene una relazione padre-figlio a tutti gli effetti. Concatenando tutti i blocchi con i relativi padri, si ottiene un unico percorso con cui attraversare la blockchain, che può anche essere vista come una lista di blocchi concatenati singolarmente. Il link che funge da collegamento tra i due blocchi è un puntatore crittografico chiamato comunemente *hash pointer*, generato da una funzione hash, applicata alla concatenazione delle informazioni del blocco in un preciso ordine. La Tabella 1.1 riporta una rappresentazione generale del blocco.

Block
Prova di lavoro (nonce)
Transazioni valide
Timestamp
Merkle Tree

Tabella 1.1: Rappresentazione generale del concetto di blocco in una blockchain.

- **Nonce**: Esso funge da contenitore del valore usato nella PoW e il suo valore è un numero casuale che rappresenta il valore di difficoltà con cui è stato costruito il blocco.
- **Timestamp**: Valore che indica la data di pubblicazione del blocco.

I Merkle tree furono inventati da Ralph Merkle nel 1988, nel tentativo di costruire migliori firme digitali; questa struttura in una blockchain ha lo scopo di diminuire il costo sulla verifica delle transazioni, senza dover accedere singolarmente ad esse, e di conseguenza fornisce un modo per verificare la correttezza dell'intera blockchain. La costruzione di questo albero inizia dalle foglie, dove ogni foglia contiene il valore hash calcolato tramite una funzione unidirezionale come MD5 oppure SHA-1, procedendo così alla creazione dei nodi interni che corrispondono alla funzione hash della concatenazione dei figli. Utilizzando la funzione hash nella costruzione della struttura ad albero, viene garantita l'integrità delle informazioni: ciò vuol dire che, se qualche malintenzionato cambiasse il valore all'interno di una foglia, tutto il cammino radice foglia sarebbe alterato, con un cambiamento inevitabile anche della radice. La proprietà di questa funzione vieta a chiunque di alterare le informazioni relative alle transazioni all'interno di un blocco una volta reso pubblico; ciò rende impossibile, ad esempio, aggiungere una transazione, rimuoverla o modificarla, come mostrato in Figura 1.1.

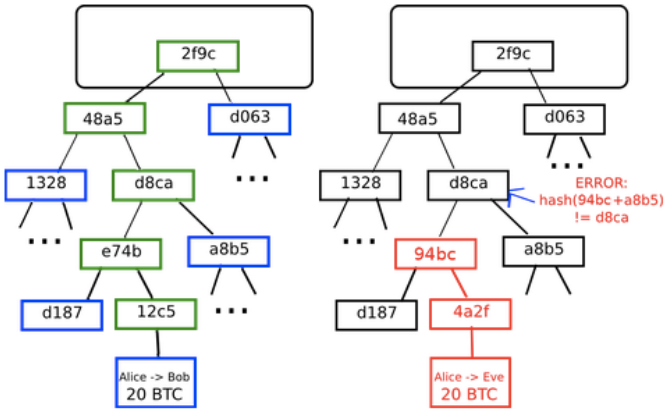

Figura 1.1: Esempio di stato non valido per un cammino radice foglia. [5]

Capitolo 2

Bitcoin

2.1 Introduzione

Bitcoin introdusse per primo il concetto di blockchain: ciò permise la creazione di un metodo di pagamento peer-to-peer esente da un intermediario, definito anche come criptovaluta decentralizzata. Introdotto da una persona o un gruppo di persone sotto lo pseudonimo di "Satoshi Nakamoto", Bitcoin viene descritto per la prima volta nel white paper intitolato "Bitcoin: A Peer-to-Peer Electronic Cash System" pubblicato nell'ottobre 2008 sulla mailing list di crittografia del sito https://www.metzdowd.com.
Il 3 gennaio 2009 venne rilasciata la versione 0.1 del codice sorgente sotto licenza MIT da Satoshi Nakamoto che smise di contribuire al progetto nel dicembre 2011. Si stima che ad oggi Satoshi Nakamoto possegga un milione di bitcoin[1], sparsi in vari wallet.

2.2 Transazioni

Le transazioni sono una parte fondamentale di Bitcoin. Infatti tutto il sistema è progettato per assicurare la creazione, propagazione e pubblicazione di transazioni su blockchain, ma queste ultime sono concettualmente differenti dalle transazioni a cui siamo abituati: ad esempio, una transazione di un database relazionale rappresenta un evento che innesca un cambio di stato all'interno della base di dati, dove in caso di malfunzionamenti la base di dati ritorna nella condizione precedente, prima che l'evento fosse scatenato. Le transazioni di Bitcoin sono concettualmente differenti.

Esempio 2.2.1. Consideriamo il seguente scenario con i seguenti personaggi:

- Alice e Bob sono due utenti che usano attivamente Bitcoin per effettuare i loro pagamenti.
- Vincent è un partecipante della rete di Bitcoin e si occupa di effettuare il mining (argomento trattato nella Sezione 2.4) per proteggere la rete e trarre ricavi in bitcoin.

Alice ha deciso di inviare 0.00700767 bitcoin a Bob; Alice dopo aver ottenuto l'indirizzo Bitcoin di Bob, che corrisponde a "33mMAc6nGyENdKMQTr5SrKoEkwNTeZQUx9", può inviare dei bitcoin a Bob, generando una nuova transazione con il seguente identificativo:

[1] Un milione di bitcoin equivale a 9297736092.00 di euro, nel 16/07/2019

“ddd587d54b693a9bc9bda2218c6f5e17979f6ac53755c5c1f668f3fa728e472d”.
Questa nuova transazione consuma una precedente transazione in possesso di Alice, chiamata UTXO (*unspent transaction output*), e crea un nuovo UTXO in possesso di Bob; la nuova transazione viene verificata da un componente (nodo) della rete, in questo caso Vincent.

Figura 2.1: La nuova transazione creata da Alice ricercata attraverso un explorer [2].

Come illustrato in Figura 2.1, l’operazione di Alice provoca lo spostamento di bitcoin da una precedente transazione con il seguente identificativo: “a57c2a427dfa591b1243-343c8413c249faac3e5df2fe4fa1fc93dca3d904f3c7” a due diversi indirizzi, cioè:

- 33mMAc6nGyENdKMQTr5SrKoEkwNTeZQUx9 (indirizzo Bitcoin di Bob);
- bc1qlctv5xhgw0yalp2vs47u342pegqm5r843c64dv (indirizzo Bitcoin di Alice): poiché gli UTXO sono indivisibili, essi devono venire consumati interamente. Nel momento in cui si genera una nuova transazione viene scelto l’UTXO più adatto con un valore di bitcoin uguale al valore da inviare a Bob; se questo non esiste, si seleziona l’UTXO *best fit*, con la conseguenza che il wallet genererà una nuova transazione verso se stesso per recuperare i bitcoin rimanenti. Questo processo è illustrato in Figura 2.2.

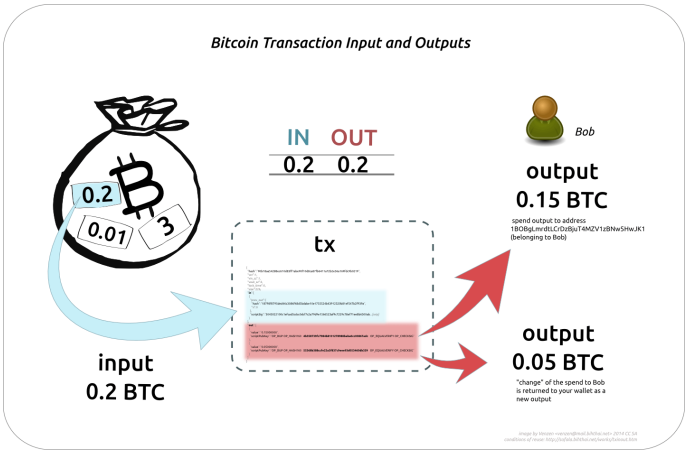

Figura 2.2: Gestione degli UTXO nella blockchain di Bitcoin [2].

La verifica della nuova transazione di Alice viene effettuata da Vincent, il quale inserisce la transazione in un nuovo blocco insieme a tutte le altre pubblicate sulla rete in quel

momento.

Nel momento in cui viene pubblicato il blocco il sistema crea una transazione verso Vincent come premio per il lavoro svolto; questa nuova transazione è conosciuta con il nome di transazione *coinbase* ed è mostrata in Figura 2.3.

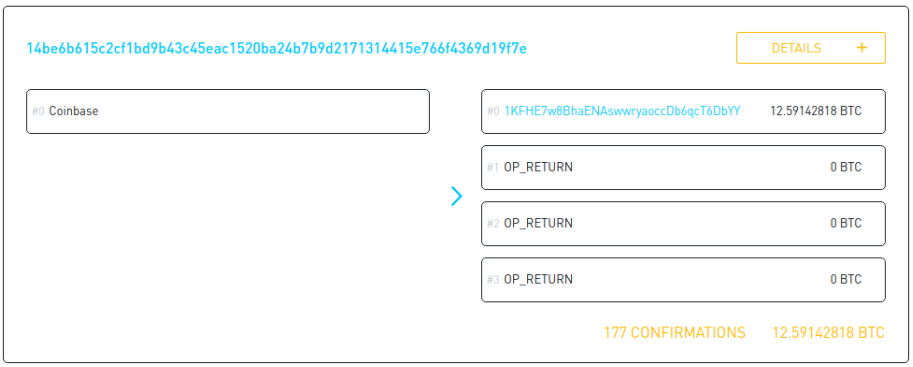

Figura 2.3: Transazione coinbase descritta [2].

L'esempio precedente mostra come in Bicoin si possano distinguere due tipi: la transazione *coinbase* e la transazione comune.

- La transazione coinbase è un evento speciale che si verifica sul sistema, il cui formato risulta non valido per la blockchain, poiché la sua struttura contiene zero input e un output. Questo evento rappresenta nel sistema un'emissione di nuovi bitcoin (argomento trattato nella Sezione 2.4), rendendo la transazione coinbase una transazione speciale per il sistema Bitcoin.
- Una transazione comune è un evento che punta a sbloccare una transazione esistente su blockchain, conosciuta come UTXO, allo scopo di creare un movimento di bitcoin.

Possono verificarsi principalmente due tipi di fallimenti durante la creazione di una transazione:

1. Una transazione non è valida per il sistema Bitcoin; in questo caso esso viene inoltrata alla rete come una transazione valida, ma essa verrà rigettata dal primo nodo completo[2] utile. Un esempio di transazione non valida è una transazione che contiene zero input ed un output.
2. Una transazione valida per il sistema ma non valida per sbloccare UTXO: essa viene accettata dal sistema Bitcoin perché valida, ma la transazione non è capace di sbloccare nessun UTXO. Un esempio è costituito da una transazione che prova a spendere due volte lo stesso UTXO (evento di doppia spesa).

Bitcoin pertanto descrive le transazioni come "consumabili", cioè una transazione di input consuma un UTXO e nello stesso istante produce un UTXO ex novo.

[2]Per nodo completo viene inteso un nodo che possiede l'intera copia della blockchain ed è abilitato per la validazione di transazioni; esso viene utilizzato comunemente dai miner.

Esempio 2.2.2. Consideriamo nuovament l'esempio precedente e supponiamo che Bob abbia deciso di inviare parte dei bitcoin ricevuti da Alice a Sara. Una volta ottenuto l'indirizzo Bitcoin di Sara, cioè "3CM8TuY8B3sCzVQN3FWYnd5skmzSF3uvWA", per effettuare il trasferimento Bob crea una nuova transazione di 0.00137927 bitcoin con il seguente identificativo "907538047d630d571298096028572087 05ef3baa4b573e6e845d5a3e6ea11464", raffigurata in Figura 2.4.

Figura 2.4: La transazione creata da Bob verso Sara [2].

Poiché gli UTXO sono contenuti all'interno di un'area in cui risiedono tutti gli UTXO, Bob deve fornire una prova che la transazione UTXO che vuole consumare sia effettivamente sua; inoltre in seguito deve utilizzare un modo per cui solo Sara sarà in grado di spendere il suo UTXO.

A tale scopo Bitcoin offre un sistema di blocco delle transazioni di output e un sistema di sblocco per le transazioni di input. Questo sistema si basa sull'impiego di script (argomento trattato nella Sezione 2.3) che utilizzano estensivamente tecniche crittografiche con cui è possibile verificare la proprietà di quell'UTXO. Nell'esempio in cui Alice crea una nuova transazione per Bob, per rendere la transazione sbloccabile solo da Bob, Alice ha inserito all'interno dello script di blocco dell'UTXO la chiave pubblica di Bob.
Bob per creare una transazione verso Sara deve sbloccare l'UTXO che gli appartiene fornendo all'interno dello script di sblocco (contenuto all'interno la transazione di input) la firma della sua chiave privata e in un secondo momento deve creare un UTXO sbloccabile solo da Sara inserendo all'interno lo script di blocco la chiave pubblica di Sara.
In Figura 2.5 viene rappresentato il flusso di eventi che descrive il blocco e sblocco di un UTXO.

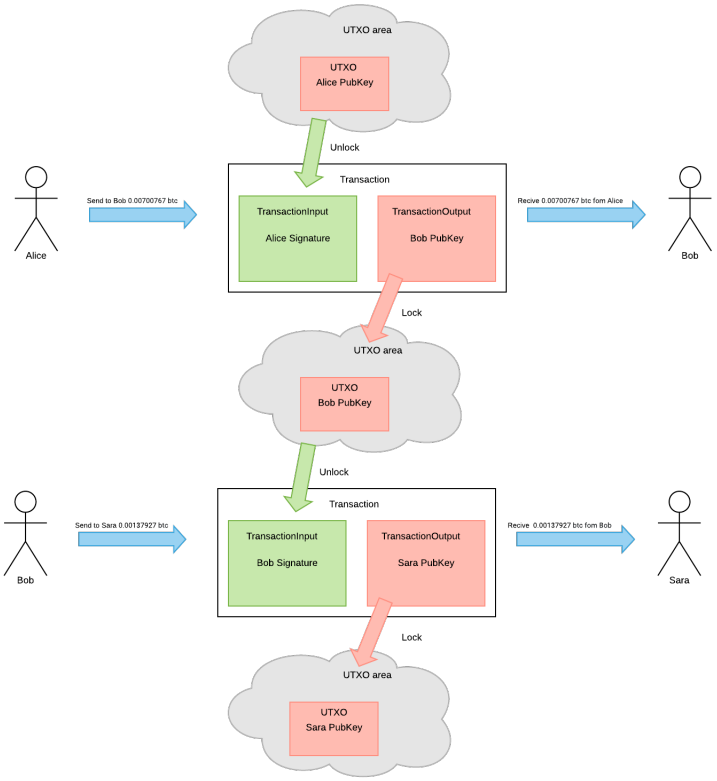

Figura 2.5: Rappresentazione del evento di blocco e sblocco di un UTXO.

Bitcoin è il primo tentativo di successo ad introdurre il concetto di transazione come struttura dati che definisce un trasferimento di valore da una o più fonti ad una o più destinazioni.

RawTransaction
Versione
Transazioni di input
Transazioni di output
Lock time

Tabella 2.1: Struttura delle transazioni in Bitcoin.

La struttura della transazione (Tabella 2.1) contiene dei campi che forniscono informazioni addizionali, ad esempio:

- **Versione**: indica le regole che strutturano la transazione.
- **LockTime**: definisce il primo istante in cui la transazione viene considerata valida e può essere trasmessa sulla rete Bitcoin; è rappresentato attraverso un valore intero compreso tra 0 e 500 milioni, assumendo significati diversi in base al valore assegnato, cioè:

- $LockTime = 0$: la transazione viene propagata ed eseguita all'istante di creazione.
- $LockTime \in (0, 500milioni]$: il valore viene interpretato come un'altezza di blocco, cioè la transazione sarà valida solo dopo essere stato pubblicato il blocco con altezza uguale al valore di lock time.
- $LockTime > 500milioni$: il valore viene interpretato come timestamp Unix e quindi la transazione sarà valida solo dopo la data rappresentata dal valore di lock time.

Gli input di transazione e gli output hanno anche loro una struttura che definisce il modo in cui si possono "spendere" i bitcoin associati a quella transazione; tale struttura contiene anche altre informazioni addizionali, ma nessuna di esse è relazionata direttamente con un wallet oppure un'identità.
La struttura delle transazioni di output (Figura 2.2) introduce un concetto fondamentale per Bitcoin, cioè le transazioni di output non spese, conosciute anche come UTXO. Quasi tutte le transazioni di output sono UTXO riconosciute da tutta la rete Bitcoin e sbloccabili attraverso uno script di sblocco (scriptSig).

Transazione di output
Importo
Script di blocco (ScriptPubKey)

Tabella 2.2: Rappresentazione struttura delle transazioni di output.

- **Valore**: valore di bitcoin espressi in satoshi che rappresentano una frazione di bitcoin come i centesimi per gli euro, $1\, bitcoin = 1 * 10^8\, satoshi$.
- **Script di blocco**: conosciuto come script *public key*, contiene le informazioni necessarie per bloccare l'output e non permettere a chiunque di spenderlo.

La struttura delle transazioni di input (Tabella 2.3) contiene due informazioni importanti: il riferimento alla transazione precedente e la condizione di sblocco della UTXO.

Transazione di input
Hash della transazione precedente
Index della transazione di output
Script di sblocco (ScriptSig)
Sequence

Tabella 2.3: Rappresentazione struttura delle transazione di input in Bitcoin.

2.3 Bitcoin Script

Gli script all'interno delle transazioni di Bitcoin vengono scritti in Bitcoin Script che è un linguaggio basato su stack e contiene molti operatori.
La mancanza di loop e funzionalità di controllo di flusso complesse rende Bitcoin Script molto limitato rispetto a linguaggi moderni come Solidity; questo cataloga Bitcoin Script come un linguaggio di programmazione non Turing completo e garantisce così l'impossibilità di eseguire loop infiniti oppure fenomeni di code-injection all'interno delle transazioni,

che potrebbero causare attacchi DoS (denial-of-service) sulla rete Bitcoin.
Gli script vengono eseguiti in modalità atomica, cioè senza possedere uno stato pre e post esecuzione. Bitcoin convalida le transazioni eseguendo gli script contenuti al suo interno: in particolare lo script di sblocco e lo script di blocco vengono eseguiti in sequenza e l'input risulta valido solo nel caso in cui lo script di sblocco soddisfi le condizioni dello script di blocco.

Nel client Bitcoin originale, gli script di sblocco e blocco venivano concatenati ed eseguiti in sequenza, ma per motivi di sicurezza questo è stato modificato nel 2010, a causa di una vulnerabilità che ha permesso a uno script di sblocco non valido di inviare dati nello stack e corrompere lo script di blocco. Nell'attuale implementazione, gli script vengono eseguiti separatamente, con lo stack trasferito tra le due esecuzioni, dove è possibile dividere la modalità di esecuzione in due passi:

1. Lo script di sblocco viene eseguito: se l'esecuzione non riporta errori, allora lo stack principale viene copiato e lo script di blocco viene eseguito.
2. Se, eseguendo lo script di blocco con i dati dello stack precedente, nel nuovo stack il risultato è "TRUE", allora lo script di sblocco è riuscito a risolvere le condizioni imposte dallo script di blocco e, pertanto, l'input è un'autorizzazione valida per spendere UTXO. Se dopo l'esecuzione dello script combinato rimangono risultati diversi da "TRUE", l'input non è valido perché non è riuscito a soddisfare le condizioni di spesa inserite in UTXO.

Durante lo sviluppo sono state rese possibili l'esecuzione di soli script di transazioni standard: ad oggi l'implementazione di riferimento ne contiene sette e sono definite in un tipo enumerazione all'interno del file `standard.h` (Codice 2.1) del client Bitcoin Core.

```
1  enum txnouttype
2  {
3      TX_NONSTANDARD,
4      TX_PUBKEY,
5      TX_PUBKEYHASH,
6      TX_SCRIPTHASH,
7      TX_MULTISIG,
8      TX_NULL_DATA,
9      TX_WITNESS_V0_SCRIPTHASH,
10     TX_WITNESS_V0_KEYHASH,
11     TX_WITNESS_UNKNOWN,
12 };
```

Codice 2.1: Porzione di codice che riporta il tipo enumerazione nel file standard.h di bitcoin core.

2.3.1 P2PKH

Lo script *pay-to-public-key-hash* fino a qualche anno fa era la tipologia di script più diffusa, perché lo script di blocco veniva composto dal hash della chiave pubblica e questo consentiva di non esporre la chiave pubblica del ricevente; l'output può essere sbloccato da uno script di sblocco contenente la chiave pubblica e la firma. Un esempio generalizzato potrebbe essere il seguente:

```
1  OP_DUP OP_HASH160 <A Public Key Hash> OP_EQUALVERIFY
2  OP_CHECKSIG
```

Codice 2.2: P2PKH Script di blocco.

```
1  <A Signature> <A Public Key>
```

Codice 2.3: P2PKH Script di sblocco.

```
1  <A Signature> <A Public Key>
2  OP_DUP OP_HASH160 <A Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG
```

Codice 2.4: Script completo.

Quando lo script viene eseguito lo stack è popolato con i valori dello script di sblocco come in Figura 2.6.

Figura 2.6: Stato iniziale dello stack.

Incontrato l'operatore OP_DUP, viene eseguita la copia dell'ultimo valore inserito all'interno dello stack, cioè <A pubkey> come in Figura 2.7.

Figura 2.7: Stato dello stack dopo l'esecuzione dell operatore OP_DUP.

Incontrando l'operatore OP_HASH160 viene calcolato l'hash dell'ultimo valore inserito ottenendo lo stato dello stack rappresentato in Figura 2.8.

Figura 2.8: Stato dello stack dopo l'esecuzione dell'operatore OP_HASH160.

Viene quindi inserito all'interno dello stack l'hash della chiave pubblica atteso, cioè <A Pub Key Hash>, ottenendo lo stack rappresentato in Figura 2.9.

Figura 2.9: Stato dello stack dopo l'inserimento della chiave pubblica attesa.

L 'operatore OP_EQUALVERIFY a questo punto verifica che le chiavi pubbliche siano uguali; se lo sono, l'esecuzione continua altrimenti, termina con un risultato diverso da TRUE. Se l'esecuzione prosegue, viene valutato l'ultimo OP code cioè OP_CHECKSIG, che ha il compito di verificare l'appartenenza della firma e della chiave pubblica alla medesima chiave privata A.

2.3.2 P2PK

Lo script *pay-to-public-key* è una forma di script più semplice del precedente, perché non viene applicato l'hash alla chiave pubblica e, quindi, l'indirizzo del ricevente è memorizzato nella transazione; per bloccare questo script si necessita solo della firma, il che fa sì che lo stack si semplifichi come segue:

```
1 <A Signature>
2 <A Public Key > OP_CHECKSIG
```

Codice 2.5: P2PK Script completo.

Quando il programma viene eseguito, lo stack è popolato con i valori dello script di sblocco come illustrato in Figura 2.10.

Figura 2.10: Stato dello stack dopo l'inserimento dello scriptSig.

Nel passo successivo la chiave pubblica viene inserita nello stack, come rappresentato dalla Figura 2.11.

Figura 2.11: Stato dello stack dopo l'inserimento del <public key>.

Avviene quindi la verifica tra firma e chiave pubblica con l'operatore OP_CHECKSIG e, se l'operatore ha come risultato TRUE, allora la transazione è sbloccabile con lo script di sblocco inserito.
Lo script P2PK, oltre ad essere uno script semplice, viene considerato dagli sviluppatori un errore di implementazione originale, perché non contiene un indirizzo Bitcoin ma contiene la reale chiave pubblica derivata dall'algoritmo di curva ellittica, motivo per cui nell'agosto 2019 da uno script P2PK non è più possibile ricavare un indirizzo Bitcoin tramite il client ufficiale. In Figura 2.12 viene mostrata la differenza del processo di creazione tra uno script P2PK e uno script P2PKH.

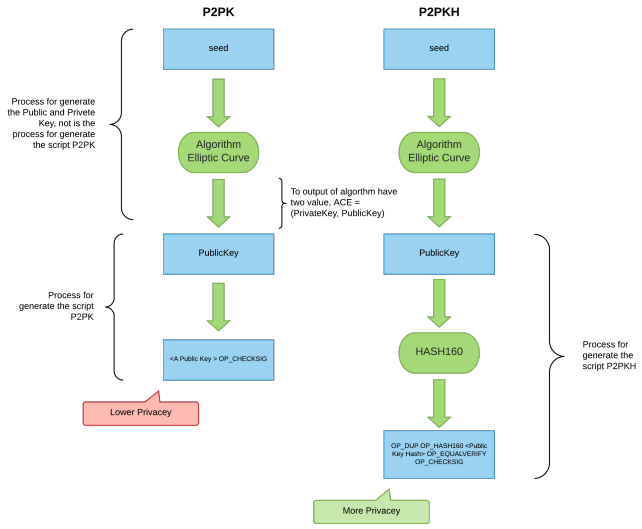

Figura 2.12: Rappresentazione del processo di creazione dello script P2PK e P2PKH.

2.3.3 P2MS

Gli script *pay-to-multisignature* definiscono una condizione M:N (molti a molti) dove M è il numero minimo di firme necessarie per verificare lo script di blocco e N è il numero totale di chiavi pubbliche. Il numero massimo di combinazioni ammesse per uno script P2MS è 15:15, ma solo le combinazioni rientranti nell'intervallo 3:3 sono considerate standard; tutte le restanti combinazioni verranno considerate come non standard. Un esempio di script P2MS 2:3 è il seguente:

```
1 OP_0 <A Signature> <B Signature>
2 OP_2 <Public key A> <Public key B> <Public key C> OP_3
    OP_CHECKMULTISIG
```

Codice 2.6: Script P2MS completo.

In questo esempio OP_0 funge da segnaposto per un bug nell'implementazione di OP_CHECKMULTISIG, il cui unico scopo è quello di aggirare un bug che è diventato accidentalmente una regola di consenso.

Lo stack inizialmente verrà popolato con i valori dello script di sblocco; la presenza dell'operatore OP_0 implica che lo stack sia vuoto quando lo si incontra. Lo stato dello stack è rappresentato dalla Figura 2.13.

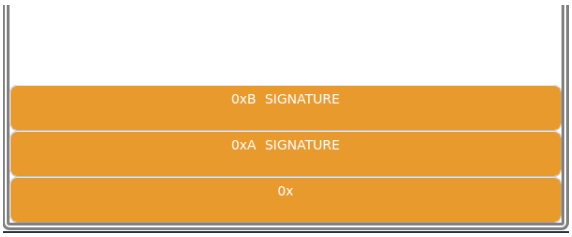

Figura 2.13: Stato dello stack dopo l'inserimento dello scriptSig nello stack.

L'operatore OP_2 verifica che nello stack ci siano 2 elementi, successivamente vengono inserite le tre chiavi pubbliche, ottenendo così uno stato come in Figura 2.14.

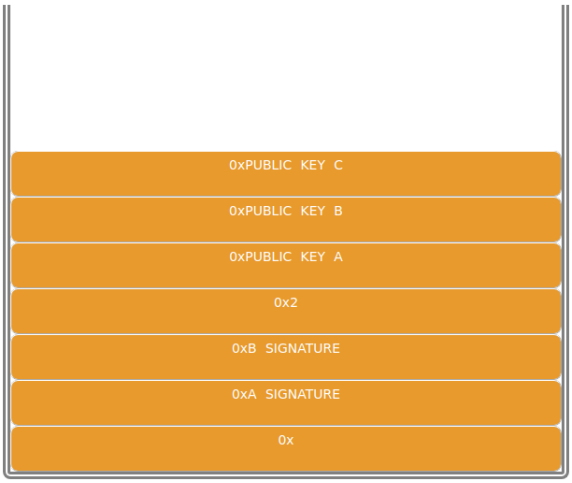

Figura 2.14: Stato dello stack dopo la verifica con OP_2 e l'inserimento nello stack delle chiavi pubbliche.

Infine le firme vengono verificate con le chiavi pubbliche mediante l'operatore OP_CHECK-MULTISIG in maniera iterativa, cioè la prima firma viene confrontata con tutte le chiavi pubbliche e l'azione si ripete per tutte le firme inserite, come in Figura 2.15.

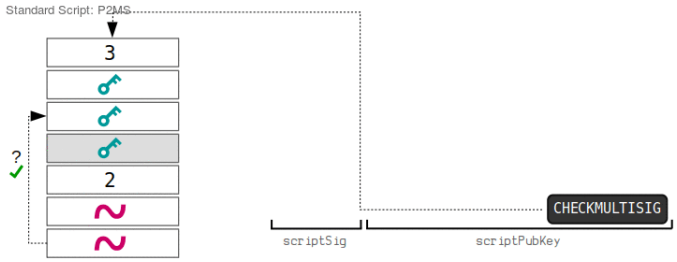

Figura 2.15: Esecuzione dell'operatore OP_CHECKMULTISIG per la verifica delle chiavi [6].

2.3.4 P2SH

Lo script *pay-to-script-hash* venne introdotto nel 2012, per semplificare l'uso degli script in transazioni complesse: infatti molto spesso accade di avere script molto grandi, come uno script P2MS 14:15, il che comporta un aumento della complessità dello script di sblocco; la motivazione dell'introduzione dello script P2SH è stata la semplificazione di quest'ultimo, come se fosse un normale pagamento ad un indirizzo Bitcoin. Questa nuova tipologia semplifica notevolmente lo script di blocco, perché sarà popolato solo dal suo hash, a discapito però dell'aggiunta di una copia dello script di blocco all'interno dello script di sblocco. Si consideri, ad esempio il seguente script P2SH:

```
1 OP_0 <A Signature> <B Signature> OP_2 <Public key A> <Public key B>
2 <Public key C> OP_3 OP_CHECKMULTISIG
3 OP_HASH160 <ScriptSig Hash> OP_EQUAL
```

Codice 2.7: Script P2SH completo.

In questo esempio lo script P2SH (Codice 2.7) completo viene composto dal seguente script di sblocco:

```
1 OP_0 <A Signature> <B Signature> OP_2 <Public key A> <Public key B>
2 <Public key C> OP_3 OP_CHECKMULTISIG
```

Codice 2.8: Script P2SH di sblocco.

Esso conterrà anche lo script di blocco, conosciuto, in questo caso, come script di riscatto (*redeemScript*). Quindi nello script di sblocco si possono distinguere:

- **<signature>**: Composto da <A Signature> <B Signature>.
- **<redeemScript>**: Composto da OP_2 <Public key A> <Public key B> <Public key C> OP_3 OP_CHECKMULTISIG.

Diversamente, lo script di blocco si semplifica come nell'esempio seguente:

```
1 OP_HASH160 <ScriptSig Hash> OP_EQUAL
```

Codice 2.9: Script P2SH di blocco.

L'esecuzione del Codice 2.7 si suddivide in due passaggi:

1. Viene valutato lo script di sblocco come descritto nella Sezione 2.3.3.
2. Viene verificato l'hash ricavato dallo script di riscatto con l'hash inserito all'interno dello script di blocco.

Il passo 1 dell'esecuzione consiste nella valutazione dello script di sblocco come un normale script P2MS, lasciando invariato lo stato di esecuzione dello stack. Il passaggio 2, invece, viene eseguito solo se l'esecuzione precedente restituisce TRUE; nel caso di script P2MS validi l'esecuzione prosegue con una copia dello stack all'interno di un nuovo stack popolato solo dal redeemScript.

Lo stato del nuovo stack viene popolato con i dati relativi allo script di riscatto come mostrato in Figura 2.16.

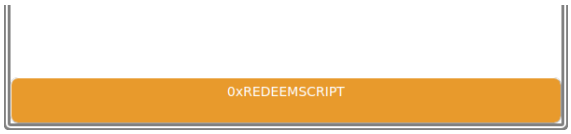

Figura 2.16: Stato dello stack dopo l'esecuzione dello ScriptSig come script P2MS.

Viene valutato l'operatore OP_HASH160, che esegue l'hash del redeemScript e porta lo stack nello stato riportato in Figura 2.17.

Figura 2.17: Stato dello stack dopo l'esecuzione dell'operatore OP_HASH160.

Come ultimo passaggio viene inserito l'hash atteso contenuto nello script di sblocco, ottenendo lo stato dello stack mostrato in Figura 2.18.

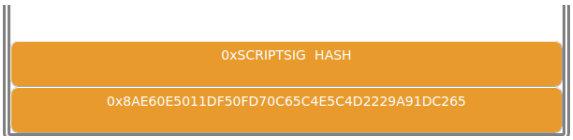

Figura 2.18: Stato dello stack dopo l'inserimento dell'hash atteso.

Infine, eseguendo l'operatore OP_EQUAL viene verificata l'uguaglianza degli hash; l'esito di tale confronto definisce il futuro della transazione.

Nell'implementazione del P2SH è stata aggiunta un'altra importante caratteristica cioè la possibilità di codificare un hash di script come un indirizzo Bitcoin attraverso l'utilizzo della codifica Base58; l'indirizzo ricavato è conosciuto anche come indirizzo P2SH, con l'unica differenza che quest'ultimo inizierà con il numero 3 per convenzione.

Un esempio di indirizzo Bitcoin P2SH:

```
1 3EBe7iCt2vhC9gqw9UdeaudjYkbfpq3b8c
```

Codice 2.10: Indirizzo Bitcoin P2SH.

Un'indirizzo primitivo Bitcoin:

```
1 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa
```

Codice 2.11: Indirizzo Bitcoin primitivo.

Attraverso l'indirizzo Bitcoin calcolato dallo script, si può riscrivere il Codice 2.9 come segue:

```
1 OP_HASH160 <P2SH key> OP_EQUAL
```

Codice 2.12: Script P2SH di blocco con Indirizzo Bitcoin P2SH.

Quindi lo script completo diventa:

```
1 OP_0 <A Signature> <B Signature> OP_2 <Public key A> <Public key B>
2 <Public key C> OP_3 OP_CHECKMULTISIG
3 OP_HASH160 <P2SH key> OP_EQUAL
```

Codice 2.13: Script P2SH completo con indirizzo Bitcoin P2SH.

L'esecuzione del Codice 2.13, che contiene un indirizzo Bitcoin P2SH al posto del hash dello script, non cambia, perché la chiave ricavata dallo script viene decodificata nel suo hash risultante, così da lasciare la sua esecuzione invariata. Bitcoin utilizza la convenzione dell'id wallet solo per una semplificazione per l'occhio umano, ma non necessita di queste informazioni per compiere le sue ordinarie operazioni; inoltre gli script P2SH e quindi gli script P2MS (in particolare script P2MS 2:2) vengono utilizzati estensivamente per la creazione e la gestione di canali all'interno della tecnologia *Lightning network*, che aumentano drasticamente la velocità delle transazioni con Bitcoin, utilizzando un protocollo *off-chain*.

2.3.5 Transazioni null data (OP_RETURN)

La blockchain di Bitcoin e, più in generale, le tecnologie blockchain hanno potenziali usi ben oltre i pagamenti. Molti sviluppatori hanno tentato di utilizzare Bitcoin script, per sfruttare la sicurezza e la resilienza del sistema, per applicazioni quali servizi notarili digitali, certificati azionari e *smart contract*.

I primi tentativi di utilizzare il linguaggio di script di Bitcoin per questi scopi hanno comportato la creazione di output di transazioni, che registravano dati sulla blockchain; nella versione 0.9 di Bitcoin core del 2014, è stato aggiunto un nuovo tipo di operatore chiamato OP_RETURN che marca la transazione come non correlata ad un movimento di bitcoin, ma ad un'archiviazione di dati. Questo operatore ha permesso ai miner di identificare la transazione e decidere se validarla o meno, considerando il problema che, archiviando una transazione con uno script non valido, oltre a creare un UTXO non spendibile, lo spazio richiesto della blockchain di bitcoin sarebbe aumentato, aumentando quindi anche le commissioni richieste dal miner. Per aggirare questo problema, il team di sviluppo decise di limitare la porzione di dati a 80 byte.

```
1 OP_RETURN <data>
```

Codice 2.14: Uso dell'operatore OP_RETURN.

Lo script precedente non fa altro che marcare il campo <data> come dati grezzi che non riguardano una transazione; infatti la semantica dell'operatore OP_RETURN marca la transazione come invalida, quindi i dati non vengono valutati.

2.4 Mining

Il processo di Mining si avvale dell'algoritmo di Proof of Work, per generare nuovi bitcoin e proteggere la rete da attacchi di malintenzionati, come ad esempio la pubblicazione di un blocco non valido o la modifica di una transazione all'interno di un blocco.
Bitcoin esegue la validazione delle transazioni in media ogni 10 minuti ed esse vengono

considerate valide solo quando il blocco che le contiene viene accodato alla catena ufficiale; ogni transazione può includere all'interno un valore di bitcoin che equivale ad una tassa indirizzata al miner per ricompensa del lavoro svolto, il quale può scegliere di dare precedenza a transazioni con una commissione maggiore di un quantitativo di bitcoin a loro scelta.
Bitcoin stabilisce un processo di creazione di nuova moneta decrescente e limitato: infatti impone un limite superiore massimo di $2.1 * 10^{15}$ bitcoin; il valore creato dai miner si dimezza ogni 210.000 blocchi, in media ogni 4 anni. La creazione di bitcoin iniziò con un ammontare pari a 50 bitcoin per blocco nel gennaio 2009 e si dimezzò a 25 bitcoin nel novembre 2012, stimando così l'emissione completa di tutti i bitcoin disponibili nel 2140; al termine dell'emissione, il guadagno di ogni miner sarà esclusivamente il valore della commissione. L'utilizzo dell protocollo Proof of Work, che è per sua natura *CPU-bound*, nel corso degli anni, con l'aumentare della competitività nel settore del mining di bitcoin, portò ad un aumento esponenziale della potenza di hashing, subendo così un cambiamento radicale della tecnologia utilizzata: si è passati da comuni CPU a componenti appositi per il calcolo della funzione hash del tipo FPGA (*mining and field programmable gate array*).[3] L'aumento della potenza di hashing (Figura 2.19) portò anche ad un aumento della difficoltà di estrazione di un blocco conosciuta come difficoltà metrica (Figura 2.20).

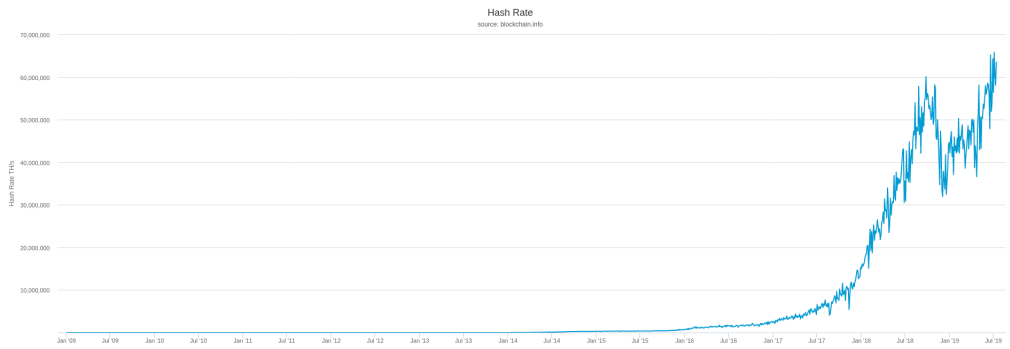

Figura 2.19: Grafico della crescita della potenza di hashing in data 28/09/2019. Fonte: blockchain.com.

[3]Nel 2014, l'energia consumata dalla rete Bitcoin era pari al consumo di elettricità dell'Irlanda (O'Dwyer e Malone, 2014)

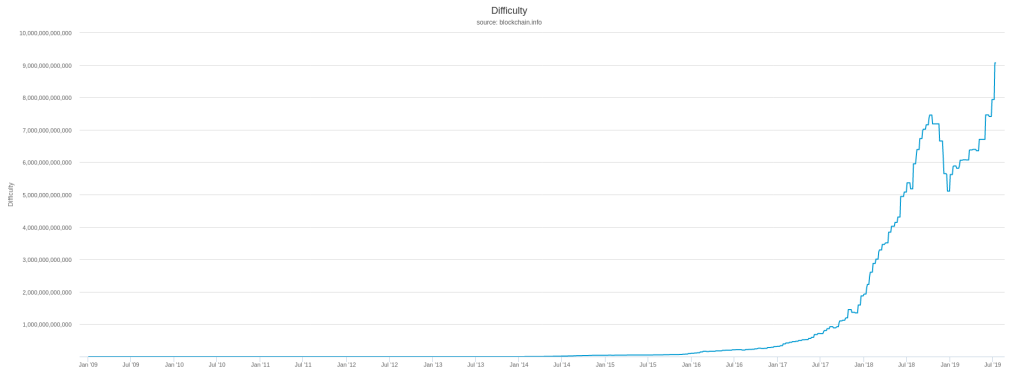

Figura 2.20: Grafico della crescita della difficoltà metrica in data 28/09/2019. Fonte: blockchain.com.

Capitolo 3

Bitcoin core

3.1 Introduzione

Bitcoin core è il successore della versione rilasciata da Satoshi Nakamoto ora conosciuta come "satoshi client"; attualmente è il client di riferimento del protocollo Bitcoin. Bitcoin core è un progetto open-source scritto in C++, il cui codice sorgente risiede attualmente su Github sotto licenza MIT e viene sviluppato da una comunità aperta di volontari; esso costituisce una riscrittura quasi completa della versione 0.1.0 del client rilasciato da Satoshi Nakamoto.

3.2 Blocchi

Così come le transazioni, il blocco viene definito attraverso una struttura dati: una volta creato e pubblicato un blocco, viene serializzato all'interno di un flat file, mantenendo i riferimenti necessari per eseguire la deserializzazione all'interno del database LevelDB di Google; le informazioni necessarie del blocco vengono contenute all'interno una struttura chiamata BlockHeader e la dimensione della struttura da deserializzare è pari a 80 byte, come rappresentato in Tabella 3.1

BlockHeader	
Type	Name
*int*32_*t*	nVersion
*uint*256	hashPrevBlock
*uint*256	hashMerkleRoot
*uint*32_*t*	nTime
*uint*32_*t*	nBits
*uint*32_*t*	nNonce

Tabella 3.1: Struttura di un block header in Bitcoin core.

- **nVersion**: Identifica le regole seguite dal blocco; fino ad ora si possono identificare 4 versioni, tutte introdotte attraverso dei soft-fork.
- **hashPrevBlock e hashMerkleRoot**: Appartengono ad un tipo di dato non primitivo del C++, ma ad un tipo definito da Bitcoin core. Rappresentano una struttura dati da 32 byte in cui vengono memorizzati l'hash del blocco precedente (in

`hashPrevBlock`) e la radice del Merkle Tree (in `hashMerkleRoot`); queste ultime garantiscono che il blocco non possa in nessun modo essere modificato, senza alterare l'intestazione del blocco.
- **nTime**: Valore che rappresenta un'epoca Unix e indica il momento in cui il miner ha iniziato ad eseguire la Proof of Work per convalidare il blocco.
- **nBits**: Un intero a 8 byte, che rappresenta il target di difficoltà dell'algoritmo di PoW del blocco.
- **nNonce**: Valore intero a 8 byte usato per contenere il valore generato dall'algoritmo di PoW.

Il blocco, oltre a contenere l'intestazione, contiene ulteriori informazioni riguardo la sua dimensione e la rete di appartenenza: infatti il client Bitcoin può essere eseguito anche in modalità testnet, in cui vengono testate le nuove versioni del software prima del rilascio ufficiale. La struttura finale del blocco, nella version attuale, può essere rappresentata nel modo indicato in Tabella 3.2

Block	
Type	Name
*int*32_*t*	magicNumber
*int*32_*t*	blockSize
BlockHeader	blockHeader
CompactSize	numberTx
vector<RawTransaction>	transactions

Tabella 3.2: Struttura di un blocco in Bitcoin core.

- **magicNumber**: Rappresenta una signature per la struttura dati e non è qualcosa di specifico per Bitcoin; il numero magico viene usato, infatti, nell'informatica per i file e i protocolli, per semplificare notevolmente il riconoscimento del file o della struttura dati: ad esempio il numero magico per un file png è 89504E470D0A1A0A, mentre il numero magico di bitcoin per la rete principale è rappresentato da 0xD9B4BEF9.

- **numberTx**: Il valore rappresenta il numero di transazioni contenute all'interno del blocco e viene serializzato usando un metodo simile alla tecnica di memorizzazione dei record a lunghezza variabile all'interno di un database relazionale; in Bitcoin core viene rappresentato attraverso un tipo di dato che porta il nome di VarInt (intero variabile) e questo valore occupa da 1 a 9 byte di spazio. Il client utilizza VarInt all'interno di LevelDB, ma utilizza il tipo di dato CompactSize rilasciato nella versione 0.1.0 di Satoshi per la serializzazione all'interno del file.

3.3 Transazioni

Come descritto nel Capitolo di 2.2, esse rappresentano l'elemento fulcro del sistema; nel corso degli anni, gli sviluppatori di Bitcoin core hanno dovuto confrontarsi con problemi nell'implementazione originale delle transazioni: uno di essi è conosciuto come problema di malleabilità. La malleabilità di una transizione consisteva nella possibilità di alterare l'identificativo della transizione (txId) durante il processo di verifica senza invalidarla.

La motivazione dell'alterazione era dovuta alla possibilità di modificare le firme nello script di sblocco (scriptSig) senza cambiare il loro significato; quindi per il modo in cui viene calcolato l'hash della transizione, questa alterazione comportava inevitabilmente l'alterazione dell'hash, portando all'interno del protocollo le seguenti problematiche:

- Il mittente non può più riconoscere la sua transazione dopo che essa è stata modificata.
- Le transazioni modificate sono effettivamente riconosciute come doppie spese, perché il mittente, non riuscendo più a riconoscere la sua transazione di input creata in precedenza, si ritroverà i bitcoin invariati e potrà spenderli una seconda volta. Questo problema non grava sul mittente, ma sull'intero sistema Bitcoin: ad esempio le transazioni che non vengono riconosciute dal mittente saranno sepolte all'interno della blockchain perché mai nessuno sarà in grado di spenderle.

Nel 2016 la community arrivò ad una soluzione tramite un soft-fork, che introdusse notevoli cambiamenti strutturali, risolvendo tutti i problemi relativi alla malleabilità di una transazione. Questo aggiornamento venne chiamato "Segregated Witness" e suddivise le transazioni in diverse parti, che possono essere gestite separatamente, sostituì le firme digitali con un segnaposto all'interno delle transazioni spostando le vere firme in una struttura dati differente. Questa soluzione non esclude però la possibilità che una transazione non sia malleabile, ma esclude una modifica dell'hash durante la verifica, poiché i dati utilizzati per il calcolo sono contenuti all'interno di uno spazio protetto; da qui nasce il nome Segregated Witness.

La struttura riportata in Figura 3.3 rappresenta la struttura delle transazioni:

RawTransaction	
Type	Name
*int*32_*t*	version
*uint*8_*t*	marker
*uint*8_*t*	flag
CompactSize	numberTxIn
vector<TransactionInput>	transactionsInput
CompactSize	numberTxOut
vector<TransactionOutput>	transactionsOutput
vector<TransactionWitness>	transactionsWitness

Tabella 3.3: Struttura della transazione dopo l'aggiornamento al Segregated witness.

- **marker e flag**: Sono valori interi a 1 byte che fungono da identificativo per una transazione conforme al formato Segregated Witness (SegWit); per i wallet non conformi al SegWit la transazione risulterà non valida, perché identifica una transazione con 0 input e 1 output.
- **transactionsWitness**: Questa lista non è preceduta da una dimensione perché la singola transazione witness ha senso solo se esiste un input correlato, quindi la dimensione di questa lista è uguale alla dimensione della lista delle transazioni di input.

Le transazioni di input sono definite tramite una struttura rappresentata dalla Tabella 3.4.

TransactionInput	
Type	Name
Outpoint	outpoint
CScript	scriptSig
*uint*32_*t*	nSequence

Tabella 3.4: La struttura della transazione input all'interno di Bitcoin core.

- **nSequence**: Il valore viene utilizzato per esprimere il timelock relativo a livello di transazione (argomento trattato nel Capitolo 3.4).
- **scriptSig**: Rappresenta la condizione di sblocco espressa dalla transazione sotto forma di script.

Il tipo di dato Outpoint utilizzato nella transazione di input contiene le informazioni necessarie per identificare la transazione di output contenuta in una transazione precedente. La Tabella 3.5 ne descrive la struttura.

Outpoint	
Type	Name
*uint*256	hash
*uint*32_*t*	index

Tabella 3.5: Struttura del tipo di dato Outpoint di Bitcoin core.

- **hash**: Rappresenta l'hash delle transazione precedente che contiene l'UTXO sbloccato dalla transazione di input che contiene outpoint.
- **index**: Il valore indica l'indice della lista in cui è memorizzato l'UTXO nella transazione precedente.

La struttura della transazione di output contiene esclusivamente le informazione dello script di blocco e il valore di bitcoin (in satoshi); in Tabella 3.6 viene illustrata la struttura dati corrispondente.

TransactionOutput	
Type	Name
*int*64_*t*	nValue
CScript	scriptPubKey

Tabella 3.6: La struttura della transazione di output di Bitcoin core.

- **nValue**: Rappresenta il valore di bitcoin (in satoshi) contenuti nella transazione.
- **scriptPubKey**: Rappresenta la condizione di blocco espressa dalla transazione sotto forma di script.

La transazione Witness rappresenta la serializzazione di tutti i dati del testimone; in Tabella 3.7 viene illustrata la struttura dati corrispondente.

TransactionWitness	
Type	Name
CompactSize	stackSize
CScript	stack

Tabella 3.7: La struttura della serializzazione dei dati del testimone; in Bitcoin core questa struttura è chiamata CScriptWitness.

- **stackSize**: Il valore rappresenta il numero di script contenuti all'interno della transazione.
- **stack**: Il valore contiene la lista di script per la singola transazione di input.

Il tipo di dato CScript è un tipo di dato contenente le informazioni necessarie per lo script; la Tabella 3.8 ne descrive la struttura.

CScript	
Type	Name
CompactSize	scriptSize
vector<unsigned char>	script

Tabella 3.8: Struttura del tipi di dato CScript di Bitcoin core.

- **scriptSize**: Rappresenta la lunghezza in byte dello script, espressa tramite il tipo di dato CompactSize.
- **script**: Rappresenta il vettore di byte dello script.

3.4 Bitcoin script

L'aggiornamento Segregated Witness comporta un notevole cambiamento anche sulla modalità di spesa degli UTXO; infatti tutti i tipi di transazione visti finora fanno riferimento allo script di sblocco, contenuto all'interno della transazione di input. Con il nuovo aggiornamento lo script di sblocco viene spostato all'interno di una struttura al di fuori delle transazioni di input, chiamata "Transazione Witness" e mostrata in Tabella 3.3. I dati relativi alle transazioni witness non contengono le informazioni di una reale transazione, bensì costituiscono uno spazio riservato per esprimere la condizione di sblocco. Per sbloccare un UTXO con il Segregated Witness bisogna far riferimento allo script contenuto all'interno delle transazioni witness, come illustrato in Tabella 3.7 (che chiameremo script witness). Questa modifica nella spesa degli UTXO ha costretto Bitcoin script ad un soft fork: ogni script d'ora in poi conterrà un numero di versione all'inizio che permette l'identificazione del tipo di transazione; lo script inizierà con un numero di versione uguale a zero per identificare uno script Witness, rendendo la transazione interpretabile anche da wallet non abilitati al Segregated Witness. Attraverso questa soluzione la transazione risulta essere spendibile da chiunque. Gli script P2PKH e P2SH con il testimone segregato si evolvono in P2WPKH e P2WSH

3.4.1 P2WPKH

Lo script *pay-to-witness-public-key-hash* si semplifica notevolmente rispetto allo script P2PKH: infatti lo script witness include la versione di verifica e l'hash della chiave pubblica come nello script P2PKH, ma in P2WPKH è chiamato "programma di controllo", composto da 20 byte.

Un esempio generalizzato:

```
1 <versione di controllo> <programma di controllo>
```

Codice 3.1: Esempio generale della struttura di uno script P2WPKH.

Un esempio reale:

```
1 0 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
```

Codice 3.2: Esempio reale di uno script P2WPKH.

3.4.2 P2WSH

Lo script *pay-to-witness-script-hash*, come lo script P2WPKH, semplifica notevolmente il predecessore e lo sostituisce interamente; un esempio di script P2WSH può essere il seguente:

```
1 0 a9b7b38d972cabc7961dbfbcb841ad4508d133c47ba87457b4a0e8aae86dbb89
```

Codice 3.3: Esempio di uno script P2WSH.

Esso è molto simile allo script P2WPKH, ma con l'unica differenza della dimensione del programma di controllo impostata a 32 byte; la dimensione del programma di controllo è l'unico modo per differenziare le due tipologie di script. La coesistenza delle transazioni tradizionali e delle transazioni con testimone segregato introduce un problema nella comunicazione tra wallet con versioni differenti, il quale è risolto dalla possibilità di costruire un indirizzo P2SH a partire da uno script witness; inoltre, come per gli script P2SH, è possibile codificare lo script witness in un indirizzo bitcoin utilizzando una codifica Base32 con checksum, rispetto alla codifica Base58 per gli indirizzi Bitcoin tradizionali; quest'ultimo sarà simile ai seguenti indirizzi.

Per la rete Mainet:

```
1 bc1qs0c2eqayqq7afzgy38u48ytwgkkvy8yae7uu6r
```

Codice 3.4: Address base32 della rete mainet.

Per la rete Testnet:

```
1 tb1qknj2fafudjfaa9nesf7rypc0hu38p04yp5ks6g
```

Codice 3.5: Address base32 della rete testnet.

Questi nuovi indirizzi vengono utilizzati estensivamente dalle tecnologie Lightning network; inoltre, essi introducono notevoli benefici per le funzioni del Bitcoin.

3.4.3 Transazioni non standard

Definire solo sette tipi di script tramite Bitcoin script potrebbe risultare restrittivo; infatti negli anni il linguaggio ha subito notevoli evoluzioni grazie alle quali si possono ottenere script complessi che eseguono operazioni non banali. Queste evoluzioni hanno sopratutto reso possibile lo sviluppo di tecnologie che lavorano a stretto contatto con la tecnologia Bitcoin e che puntano a migliorare le parti del protocollo che rendono Bitcoin inadatto per alcune applicazioni del mondo reale. Ad esempio, la velocità delle transazione risulta essere un punto debole di Bitcoin a causa dell'algoritmo di PoW; questo limite viene aggirato con la proposta di un layer addizionale, conosciuto come Lightning Network, il quale usa una caratteristica di Bitcoin script conosciuta come timelock relativo (introdotto nel Paragrafo 3.4.3) con cui si può utilizzare Bitcoin offchain.

In Figura 3.1 viene illustrato il confronto delle transazioni per secondo (tps) tra Bitcoin e la tecnologia Lightning Network.

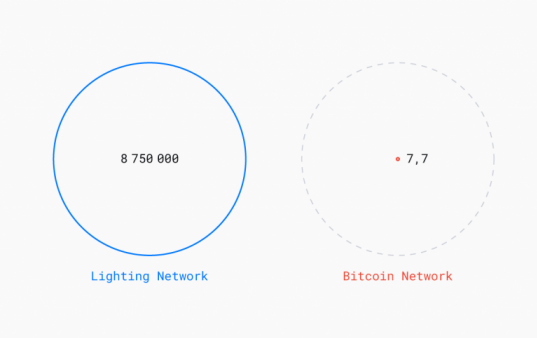

Figura 3.1: Confronto tra le tecnologie Lightning network e Bitcoin: transazioni per secondo (tps) [3].

Timelock

I timelock vengono introdotti nel 2016 e sono usati per esprimere restrizioni sulla modalità di spesa di UTXO al fine di consentire lo sblocco di questi ultimi solo dopo un certo evento. Questa definizione non è nuova nella tecnologia Bitcoin perché con il campo `nLockTime` della transazione è possibile esprimere una restrizione sulla modalità di spesa di una transazione (argomento trattato nel Capitolo 2.2); questo tipo di restrizione soffre di un problema illustrato nel seguente esempio (tratto dal libro [1]).

Esempio 3.4.1. Supponiamo che Alice firmi una transazione spedendo uno dei suoi UTXO all'indirizzo di Bob impostando la transazione con un nLocktime proiettato nel futuro di 3 mesi. Con questa transazione Alice e Bob sanno che:

- Bob non può trasmettere la transazione per riscattare i fondi fino a quando non sono trascorsi 3 mesi.
- Bob può trasmettere la transazione dopo 3 mesi.

Però:

- Alice può creare un'altra transazione, spendendo due volte gli stessi input senza un locktime. Pertanto, Alice può spendere lo stesso UTXO prima che siano trascorsi i 3 mesi.

- Bob non ha alcuna garanzia che Alice non lo farà.

Il concetto di timelock introduce un nuovo operatore in Bitcoin Script che prende il nome di OP_CHECKLOCKTIMEVERIFY (CLTV); questo operatore risolve il problema illustrato nell'esempio precedente poiché blocca la transazione a livello di script (utilizzando lo script di sblocco), ma l'operatore CLTV non punta a sostituire completamente il lavoro svolto dalla proprietà nLocktime; invece esso lavora in associazione con quest'ultima il cui valore deve essere maggiore o uguale al valore inserito nello script per rendere la transazione valida; se questa condizione viene a mancare, il sistema rifiuterà la transazione.

Sia nLockTime che CLTV sono tecniche di locking assolute in quanto specificano un quanto di tempo assoluto; con esse, quindi, non è possibile esprimere lassi di tempo relativi come, ad esempio, esprimere un lasso di tempo valido a partire dalla conferma della transazione (problema risolto dal timelock relativo, affrontato nella Sezione 3.4.3). Un esempio di script di blocco complesso estrapolato dal [4]:

```
1 OP_IF
2   <now + 3 months> OP_CHECKLOCKTIMEVERIFY OP_DROP
3   <C pubkey> OP_CHECKSIGVERIFY
4   1
5 OP_ELSE
6   2
7 OP_ENDIF
8  <A pubkey> <B pubkey> 2 OP_CHECKMULTISI
```

Codice 3.6: Script di Blocco che esprime una multipla condizione di spesa utilizzando OP_CHECKLOCKTIMEVERIFY.

L'UTXO creato con il precedente script di blocco può essere sbloccato in due modi:

- In qualsiasi momento da A e B con il seguente script:

```
1    0 <A signature> <B signature> 0
```

- Dopo tre mesi da A o B e C con il seguente script:

```
1    0 <A or B signature> <C signature> 1
```

Lo script di blocco precedente esprime una condizione utilizzando l'operatore CLTV che specifica un quanto di tempo pari a tre mesi. É tuttavia possibile utilizzare due notazioni diverse: infatti, si può esprimere il quantitativo di tempo tramite la dichiarazione del numero di blocchi successivi oppure con un timestamp Unix. Un esempio estrapolato dal libro [1] potrebbe essere:

- Altezza attuale + 12.960 (blocchi).
- Timestamp corrente + 7.760.000 (secondi).

Gli script di sblocco terminano entrambi con dei suffissi, cioè 0 e 1. Questi valori servono per la corretta esecuzione della clausola if-then-else, espressa in maniera differente da una condizione if-then-else di un linguaggio moderno tipo il C++. Infatti la clausola viene definita in maniera generale in Bitcoin script come nell'esempio seguente, utilizzando il numero 0 per esprimere FALSE e 1 per TRUE.

```
1 condizione
2 OP_IF condizione da eseguire per una condizione vera
3  codice
4 OP_ELSE da eseguire per una condizione falsa
5 codice
6 OP_ENDIF da eseguire in entrambi i casi
```

Codice 3.7: Clausola if-then-else in Bitcoin script.

Bitcoin core supporta anche timelock relativi, i quali sono utili per stabilire vincoli temporali rispetto al tempo di conferma di una transazione sulla blockchain, così da permettere di esprimere un lasso di tempo relativo che dipende (in questo caso) dall'istante in cui la transazione viene confermata. Come i timelock assoluti, i timelock relativi vengono espressi sia a livello di script che a livello di transazione; per fare ciò viene utilizzato il campo `nSequence` all'interno della transazione di input. Si possono distinguere attualmente due tipi di timelock relativi:

- Timelock relativo bastato su consenso con nSequence.
- Timelock relativo basato su OP_CHECKLOCKTIMEVERIFY(CLTV).

Timelock relativo bastato su consenso con nSequence

I timelock relativi possono essere impostati su ogni input di una transazione; l'introduzione in corso d'opera di questa nuova funzionalità senza produrre un hard-fork è stata resa possibile dall'esistenza del campo nSequence contenuto nella transazione di input, originariamente destinato ad una funzione (mai correttamente implementata) che consentiva la modifica della transazione durante la fase di creazione e propagazione di quest'ultima, cioè:

- nSequence != 0xFFFFFFFF: La transazione poteva subire modifiche (transazione non finalizzata), quindi essa veniva mantenuta in un'area di memoria in cui tutte le transazioni pubblicate ed in attesa di essere verificate risiedono, conosciuta anche sotto il nome di *mempool*. Fin quando la transazione conteneva un valore diverso da 0xFFFFFFFF, essa non veniva presa in considerazione dai miner.
- nSequence = 0xFFFFFFFF: La transazione veniva considerata come "finalizzata" e quindi presa in considerazione dati miner.

Questo tipo di timelock comporta alcune modifiche alle regole di consenso, le quali in base al bit più significativo interpretano il tipo di timelock applicato (regole di consenso elencate nel [4]).

Timelock relativo basato su OP_CHECKSEQUENCEVERIFY

Come nel timelock assoluto, anche per il timelock relativo è stato introdotto un nuovo operatore in Bitcoin script. Tale operatore prende il nome OP_CHECKSEQUENCEVERIFY (CSV) e lavora in associazione con il campo nSequence: esso verifica la distanza dal tempo in cui è stata accettata UTXO fino al momento di valutazione dell'operatore CSV. Vediamo un esempio estrapolato dalla demo di *miniscript* illustrato nel Codice 3.8:

```
1 <key_1> OP_CHECKSIG OP_SWAP <key_2> OP_CHECKSIG OP_ADD OP_SWAP <
    key_3>
```

```
2 OP_CHECKSIG OP_ADD OP_SWAP OP_DUP
3 OP_IF
4   <a032> OP_CHECKSEQUENCEVERIFY OP_VERIFY
5 OP_ENDIF
6 OP_ADD 3 OP_EQUAL
```

Codice 3.8: Uno script complesso che utilizza l'operatore OP_CHECKSEQUENCE-VERIFY.

L'esempio illustra come bloccare un input con uno script che esprime una condizione 3:3 e che solo dopo 90 giorni può essere sbloccato con una combinazione 2:3. L'evoluzione di Bitcoin script, oltre ad aumentare i campi di applicazione di Bitcoin, aumenta drasticamente anche la difficoltà di scrittura degli script complessi che possono ora essere definiti come *Smart contract*; infatti la scrittura di Smart contract e la gestione del ciclo di vita di esso risulta essere complesso. Per risolvere questi problema nell'agosto del 2019 è stato rilasciato il codice sorgente di un linguaggio per rappresentare Bitcoin script in modo strutturato, chiamato *miniscript*.

Miniscript

Miniscript è un linguaggio implementato da Pieter Wuille, cofondatore di Blockstream e attuale sviluppatore Bitcoin core, con l'obiettivo di semplificare la scrittura di smart contract. Miniscript è implementato in C++ ed è stato sviluppato per rappresentare un sottinsieme di Bitcoin script in modo strutturato. Ad esempio il Codice 3.8 può essere riscritto tramite il Codice 3.9 utilizzando miniscript:

```
1 thresh(3, pk(key_1), pk(key_2), pk(key_3), older(12960))
```

Codice 3.9: Un esempio di utilizzo di miniscript.

In questo esempio:

- thresh(x, y, . . . , k): esprime un'equazione del tipo x + y + . . . + n = k.
- pk(key): effettua il controllo sulla chiave presa in input.
- older(k): esprime un'espressione di base per miniscript la quale effettua il controllo con il valore preso in input; nel Codice 3.9 il valore deve essere maggiore di 12960 blocchi.

Appendice A

Appendice

Bibliografia

[1] Andreas M. Antonopoulos. *Mastering Bitcoin 2nd Edition - Programming the Open Blockchain*. O'Reilly, 2017.

[2] Blockstream. *Explora*. URL: https://github.com/Blockstream/esplora.

[3] DataLight. «Lightning Network Study. Will this technology become the new standard for Bitcoin transactions?» In: (). URL: https://datalight.me/blog/researches/lightning-network-study-will-this-technology-become-the-new-standart-for-bitcoin-transactions/.

[4] Nicolas Dorier kinoshitajona Mark Friedenbach BtcDrak. «Relative lock-time using consensus-enforced sequence numbers». In: (). URL: https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki.

[5] Ethereum Team. «A Next-Generation Smart Contract and Decentralized Application Platform». In: (). URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[6] Greg Walker. «Pay To Multisig». In: (). URL: https://learnmeabitcoin.com/glossary/p2ms.