



UNIVERSITÀ DEGLI STUDI DELLA BASILICATA

DIPARTIMENTO DI MATEMATICA, INFORMATICA ED ECONOMIA

Corso di Laurea Triennale in Scienze e tecnologie informatiche

Estrazione di Informazioni dalla Blockchain di Bitcoin

Relatore:
Dott. Carlo Sartiani

Candidato:
Vincenzo Palazzo

Anno Accademico 2018-2019

Non è tanto chi sono ma quello
che faccio che mi qualifica.

Bruce Wayne

Sommario

Piacere, sommario.

Indice

Sommario	II
1 Blockchain	1
1.1 Introduzione	1
1.2 Protocolli di consenso	1
1.3 Blocchi	2
2 Bitcoin	4
2.1 Introduzione	4
2.2 Transazioni	4
2.3 Bitcoin Script	9
2.3.1 P2PKH	10
2.3.2 P2PK	11
2.3.3 P2MS	13
2.3.4 P2SH	14
2.3.5 Transazioni null data (OP_RETURN)	17
2.4 Mining	17
3 Bitcoin core	19
3.1 Introduzione	19
3.2 Blocchi	19
3.3 Transazioni	20
3.4 Bitcoin script	23
3.4.1 P2WPKH	24
3.4.2 P2WSH	24
3.4.3 Transazioni non standard	25
4 Problema	29
4.1 Introduzione	29
4.2 Grafo delle transazioni	30
4.3 Grafo degli address	31

5	Stato dell'Arte	35
5.1	BlockSci	35
5.2	BiVA	36
5.3	Bitcoin Transaction Visualization	36
6	Tecnologie Utilizzate	38
6.1	Bitcoin-Cryptography-Library	38
6.2	RapidJSON	38
6.3	Bitcoin-api-cpp	38
6.4	JSON-RPC 1.0	38
6.5	Zlib	39
6.6	Bitcoin core library	39
6.7	OpenMP	39
6.8	Ngraph	39
7	Soluzione Proposta	41
7.1	Introduzione	41
7.2	SpyCBlock	41
7.3	SpyCBlockRPC	42
7.4	Serializzazione della blockchain in formato JSON	42
7.5	Grafo delle transazioni	44
7.6	Grafo degli address	46
7.7	Risultati Sperimentali	52
7.8	DEMO	53
7.8.1	AnalyticsPyBlock	53
7.8.2	SpyJSBlocks	54
8	Conclusioni	57
	Ringraziamenti	58
A	Appendice	1
A.1	Decodifica Transazione con Segregated Witness	1
A.2	Frammento di codice estratto dalla serializzazione in JSON di SpyCBlock	2
A.3	Processo per la creazione dell'hash rappresentato in un test di unità	2
A.4	Processo per la creazione dell'address primitivo da una chiave pubblica	4
	Riferimenti bibliografici	1

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 Bitcoin¹, 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 vittoria 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

¹Utilizzeremo 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.

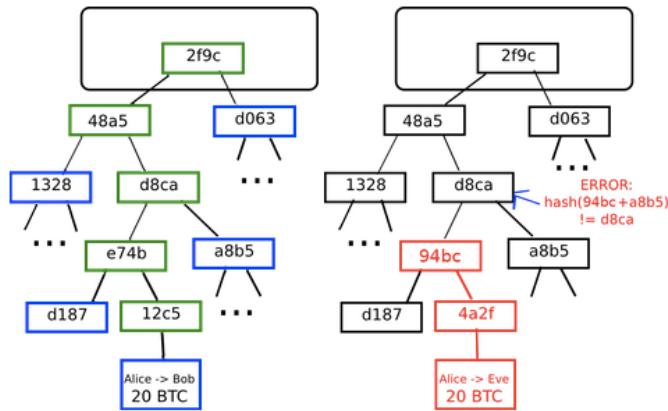


Figura 1.1: Esempio di stato non valido per un cammino radice foglia. [27]

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 possieda un milione di bitcoin¹, 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:

¹Un 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 [5].

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 (conosciuta come *exchange transaction*) verso se stesso per recuperare i bitcoin rimanenti. Questo processo è illustrato in Figura 2.2.

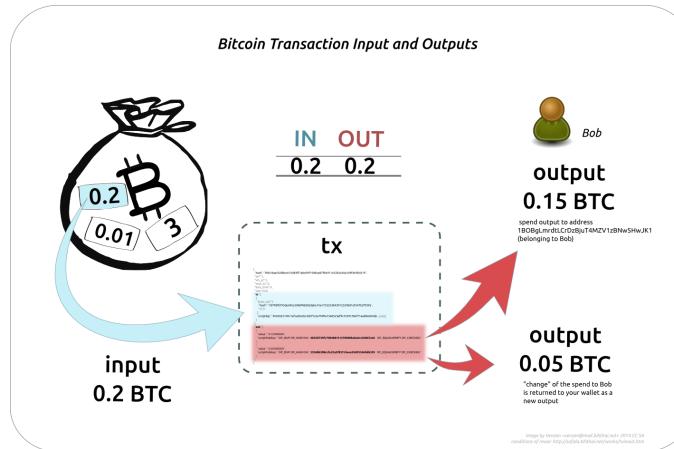


Figura 2.2: Gestione degli UTXO nella blockchain di Bitcoin [5].

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 [5].

L'esempio precedente mostra come in Bitcoin 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 completo² 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).

²Per 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.

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

Esempio 2.2.2. Consideriamo nuovamente 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 “907538047d630d57129809602857208705ef3baa4b573e6e845d5a3e6ea11464”, raffigurata in Figura 2.4.



Figura 2.4: La transazione creata da Bob verso Sara [5].

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.

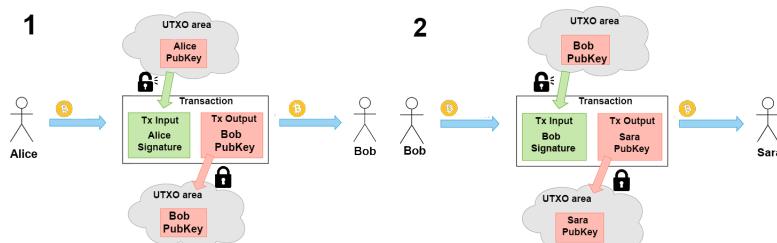


Figura 2.5: Rappresentazione del evento di blocco e sblocco di un UTXO.

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.

Raw Transaction
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.
 - $LockTime \in (0, 500 \text{ milioni}]$: 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.
 - $LockTime > 500 \text{ milioni}$: 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, $1 \text{ bitcoin} = 1 * 10^8 \text{ satoshi}$.

- **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.

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

```
1 enum txouttype
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};
```

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:

Codice 2.2: P2PKH Script di blocco.

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

Codice 2.3: P2PKH Script di sblocco.

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

Codice 2.4: Script completo.

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

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:

Codice 2.5: P2PK Script completo.

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

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.

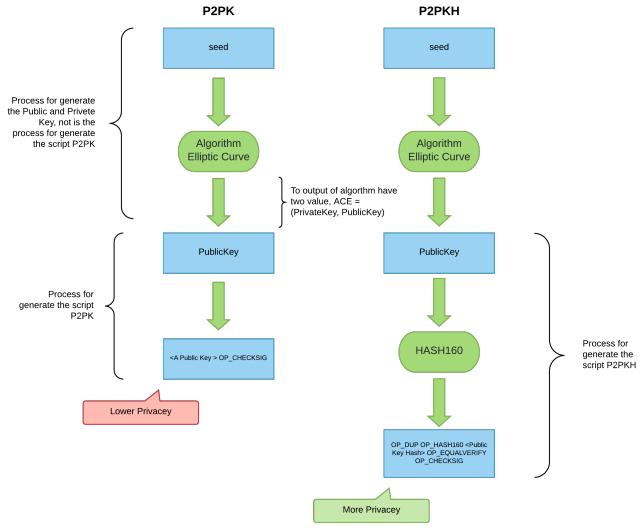


Figura 2.12: Rappresentazione del processo di creazione dello script P2PK e P2PKH.

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:

Codice 2.6: Script P2MS completo.

```

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

```

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_CHECKMULTISIG 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.

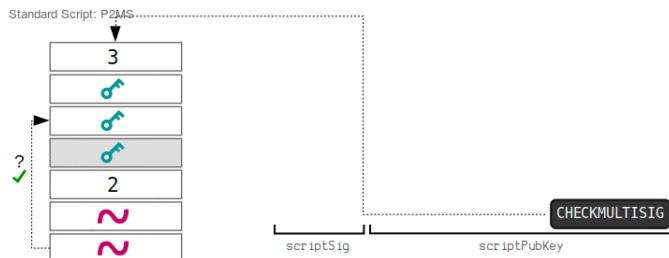


Figura 2.15: Esecuzione dell'operatore OP_CHECKMULTISIG per la verifica delle chiavi [32].

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:

Codice 2.7: Script P2SH completo.

```

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

```

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

Codice 2.8: Script P2SH 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

```

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:

Codice 2.9: Script P2SH di blocco.

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

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:

Codice 2.10: Indirizzo Bitcoin P2SH.

¹ 3EB^e7iCt2vhC9gqw9Udea^dYkbfpq3b8c

Un'indirizzo primitivo Bitcoin:

Codice 2.11: Indirizzo Bitcoin primitivo.

¹ 1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa

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

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

¹ OP_HASH160 <P2SH key> OP_EQUAL

Quindi lo script completo diventa:

Codice 2.13: Script P2SH completo con indirizzo Bitcoin 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 <P2SH key> OP_EQUAL`

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.

Codice 2.14: Uso dell'operatore OP_RETURN.

¹ `OP_RETURN <data>`

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 del 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).

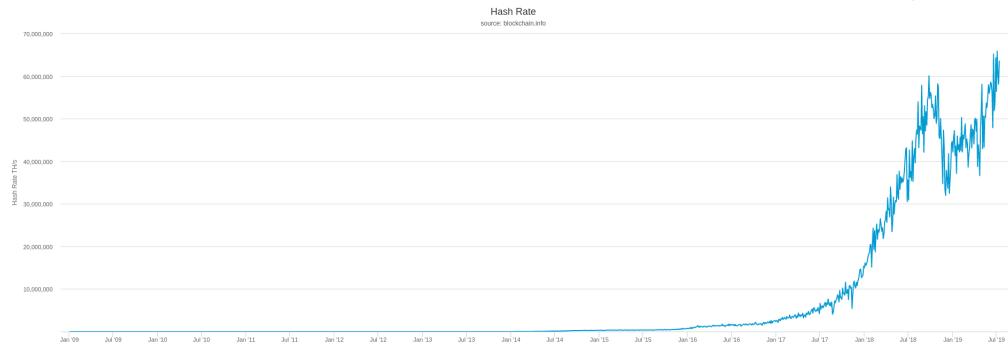


Figura 2.19: Grafico della crescita della potenza di hashing in data 28/09/2019. Fonte: blockchain.com.

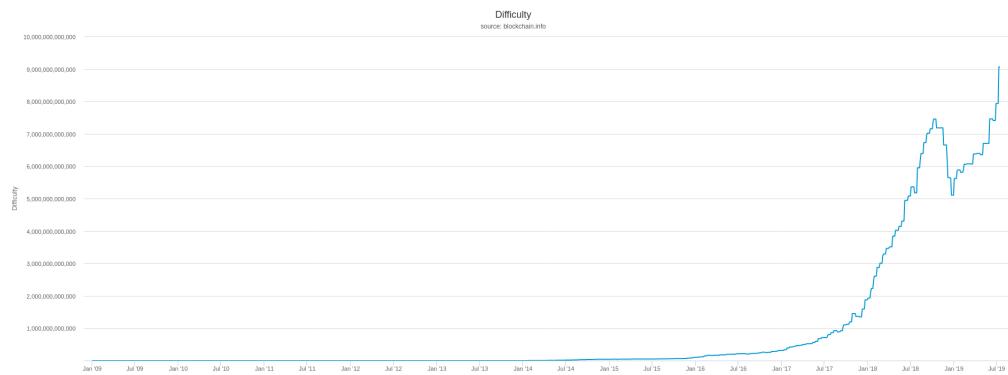


Figura 2.20: Grafico della crescita della difficoltà metrica in data 28/09/2019. Fonte: blockchain.com.

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

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
<i>int32_t</i>	nVersion
<i>uint256</i>	hashPrevBlock
<i>uint256</i>	hashMerkleRoot
<i>uint32_t</i>	nTime
<i>uint32_t</i>	nBits
<i>uint32_t</i>	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 versione attuale, può essere rappresentata nel modo indicato in Tabella 3.2

Block	
Type	Name
<code>int32_t</code>	<code>magicNumber</code>
<code>int32_t</code>	<code>blockSize</code>
<code>BlockHeader</code>	<code>blockHeader</code>
<code>CompactSize</code>	<code>numberTx</code>
<code>vector<RawTransaction></code>	<code>transactions</code>

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
<i>int32_t</i>	version
<i>uint8_t</i>	marker
<i>uint8_t</i>	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
<i>uint32_t</i>	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
<i>uint256</i>	hash
<i>uint32_t</i>	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
<i>int64_t</i>	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 tipo 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:

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

¹ <versione di controllo> <programma di controllo>

Un esempio reale:

Codice 3.2: Esempio reale di uno script P2WPKH.

¹ 0 ab68025513c3dbd2f7b92a94e0581f5d50f654e7

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:

Codice 3.3: Esempio di uno script P2WSH.

¹ 0 a9b7b38d972cabcb7961dbfbcb841ad4508d133c47ba87457b4a0e8aae86dbb89

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:

Codice 3.4: Address base32 della rete mainet.

¹ bc1qs0c2eqayqq7afzgy38u48ytwgkkvy8ya7uu6r

Per la rete Testnet:

Codice 3.5: Address base32 della rete testnet.

¹ tb1qknj2fafudjf9aa9nesf7rypc0hu38p04yp5ks6g

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 soprattutto 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.

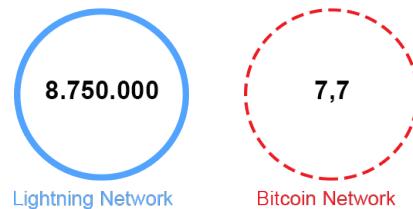


Figura 3.1: Confronto tra le tecnologie Lightning network e Bitcoin: transazioni per secondo (tps) [7].

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 [3]).

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 [18]:

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

```
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_CHECKMULTISIG
```

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. È 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 [3] 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.

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

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

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(CLT)**.

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 [18]).

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:

Codice 3.8: Uno script complesso che utilizza l'operatore **OP_CHECKSEQUENCEVERIFY**.

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
```

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 problemi 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:

Codice 3.9: Un esempio di utilizzo di miniscript.

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

In questo esempio:

- `thresh(x, y, ..., k)`: esprime un'equazione del tipo $x + y + \dots + n = k$.
- `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.

Capitolo 4

Problema

4.1 Introduzione

Bitcoin è a tutti gli effetti una criptovaluta pseudo-anonima. Fin dalla sua introduzione, questa caratteristica ha suscitato particolare interesse all'interno del *Dark Web*; ciò ha reso i bitcoin, a partire dal 2011, lo strumento di pagamento elettronico principale per attività correlate al mercato nero di droghe, farmaci e armi, nonché uno dei principali strumenti per il riciclaggio di denaro.

Il concetto di distributed ledger, associato all'utilizzo estensivo di Bitcoin per attività illecite, ha dato vita a numerosi studi di analisi forense per tracciare il flusso dei bitcoin. Queste analisi vengono di solito effettuate a partire da rappresentazioni a grafo della blockchain di Bitcoin.

I grafi più comunemente utilizzati sono il *grafo delle transazioni* e il *grafo degli address*. L'esempio 4.1.1 rappresenta una problematica riguardante l'utilizzo di diversi tipi di address in Bitcoin; in cui risulta molto utile la rappresentazione a grado degli address.

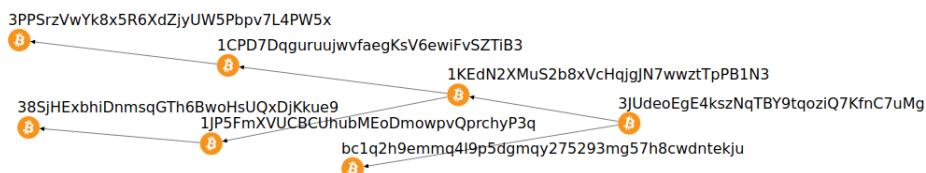


Figura 4.1: Frammento del grafo delle transazioni relativo alla transazione tra Alice e Bob descritta nell'Esempio 2.2.1.

Esempio 4.1.1. Considerando il frammento del grafo di address illustrato in Figura 4.1 dove possiamo osservare la spedizione di bitcoin dall'address “3JUdeoEgE4kszNqTBY-9tqoziQ7KfnC7uMg” verso due nuovi address:

- L'address “1KEdN2XMuS2b8xVcHqjgJN7wwztTpPB1N3” che rappresenta il destinatario dei bitcoin.
- L'address “bc1q2h9emmq4l9p5dgmqy275293mg57h8 cwdntekju” che rappresenta l'address assegnato all'exchange transaction appartenente al medesimo wallet di origine.

Dall'address “1KEdN2XMuS2b8xVcHqjgJN7wwztTpPB1N3” si genera uno spostamento di bitcoin con un ammontare minore del valore rappresentato dal precedente UTXO ricevuto, questo genera due nuove transazioni verso indirizzi distinti, cioè:

- L'address “1CPD7DqguruujwvfaegKsV6ewiFvSZTiB3” che rappresenta il destinatario dei bitcoin.
- L'address “1JP5FmXVUCBCUhubMEoDmowpvQprchyP3q” che rappresenta l'address assegnato all'exchange transaction.

Dall'address “1CPD7DqguruujwvfaegKsV6ewiFvSZTiB3” avviene uno spostamento di bitcoin verso l'indirizzo “3PPSzVwYk8x5R6XdZjyUW5Pbpv7L4PW5x” e dall'address “1JP5FmXVUCBCUhubMEoDmowpvQprchyP3q” avviene uno spostamento di bitcoin verso l'indirizzo “38SjHExbhiDnmsqGTh6BwoHsUQxDjKkue9”, si noti inoltre che gli UTXO vengono “consumati” interamente perchè non viene generata nessun exchange transaction.

Questo esempio lascia immaginare che il flusso di bitcoin originato da “3JUdeoEg-E4kszNqTBY9tqoziQ7KfnC7uMg” si sia diretto verso due destinatari diversi: cioè “3PPSzVwYk8x5R6XdZjyUW5Pbpv7L4PW5x” e “38SjHExbhiDnmsqGTh6BwoHsUQxDjKkue9”, ma questi due address appartengono allo stesso wallet.

Attraverso questa osservazione viene fornita una dimostrazione del problema relativo alla frammentazione degl'address (descritti in dettaglio nella Sezione 4.3), i quali se utilizzati nel modo giusto possono rappresentare all'interno del grafo un entità distinta. Inoltre ad oggi sulla rete bitcoin gli address originati da script sono in rapida crescita, anche perchè alcuni software generano solo address derivati da script gestendo internamente le chiavi pubbliche, nascondendo quest'ultime anche al proprietario del wallet. Questo rende gli address primitivi catalogabili come address *legacy*.

4.2 Grafo delle transazioni

Il grafo delle transazioni è un grafo diretto i cui nodi rappresentano transazioni e i cui archi denotano il consumo di transazioni da parte di altre transazioni. Come illustrato nel Capitolo 2, questo grafo può essere costruito sfruttando i backlink contenuti all'interno delle transazioni di input.

Esempio 4.2.1. Consideriamo nuovamente l'Esempio 2.2.1, nel quale Alice vuole trasferire 0.00700767 bitcoin a Bob. Esaminando la nuova transazione prodotta da Alice e indirizzata a Bob con l'id “ddd587d54b693a9bc9bda2218c6f5e17979f6ac53755c5c1f668f3fa-728e472d”, possiamo osservare che all'interno della transazione di input era contenuto un id di transazione appartenente ad un precedente UTXO in possesso di Alice (“a57c2a427dfa-591b1243343c8413c249faac3e5df2fe4fa1fc93dca3d904f3c7”). Conseguentemente, il frammento di grafo corrispondente a questo scambio di bitcoin, mostrato in Figura 4.2, contiene due nodi, il primo dei quali corrisponde all'UTXO di Alice, mentre il secondo alla transazione di Alice verso Bob.



Figura 4.2: Frammento del grafo delle transazioni relativo alla transazione tra Alice e Bob descritta nell’Esempio 2.2.1.

4.3 Grafo degli address

Il grafo degli address è un grafo diretto i cui nodi sono address e i cui archi rappresentano flussi di bitcoin da un address ad un altro, indotti direttamente o indirettamente da transazioni Bitcoin.

La creazione di questa tipologia di grafi è molto meno intuitiva, poiché, come visto nel Capitolo 2, le transazioni non contengono alcun riferimento esplicito a wallet o persone, sfruttando, invece, chiavi private e corrispondenti chiavi pubbliche. Gli address generati attraverso la chiave pubblica sono contenuti all’interno dello script di blocco in modo da rendere la transazione sbloccabile solo dal proprietario dell’address.

Come descritto nella Sezione 2.3, gli address possono essere originati anche da uno script tipo P2SH, P2WPKH e P2WSH; ciò rende la creazione del grafo degli address molto insidiosa, perché gli address ricavati dagli script (se utilizzati correttamente) sono univoci e quindi rappresenterebbero un’identità diversa all’interno del grafo, anche se molti di questi potrebbero appartenere in realtà allo stesso wallet.

Esempio 4.3.1. Consideriamo nuovamente l’Esempio 2.2.1. In tale esempio Alice genera una nuova transazione consumando un suo UTXO con un valore maggiore o uguale alla quantità di bitcoin che desidera trasferire; nell’esempio viene utilizzato un UTXO con un quantitativo maggiore, il che comporta così la generazione di una exchange transaction per riaccreditare la somma eccedente attraverso un nuovo UTXO.

Modellare questo semplice trasferimento può portare a due problematiche.

- L’exchange transaction potrebbe contenere un address differente: in questo specifico caso la transazione contiene un address ricavato da uno script Witness (che potrebbe essere ricavato dalla medesima chiave pubblica o da chiavi pubbliche distinte contenute all’interno del wallet).
- L’exchange transaction potrebbe contenere uno script P2PKH e quindi un indirizzo primitivo, ma quest’ultimo potrebbe essere differente dall’address di origine. Questo lascerebbe pensare che i bitcoin siano diretti verso un nuovo proprietario, mentre nella maggior parte dei casi è solo una tecnica per aumentare la privacy utilizzata dagli wallet. Infatti, molti wallet comunemente utilizzati contengono un set di chiavi private che permette loro di generare un address diverso ad ogni nuova operazione.

La creazione del grafo degli address comporta quindi una serie di problematiche riguardanti gli address, che possono rivelarsi molto significative.

Indirizzi originati da script Il destinatario potrebbe usare indirizzi originati da script (come un address P2SH) rendendo così difficile associare l’appartenenza allo stesso wallet di più address; in alternativa, l’address potrebbe camuffare uno script P2MS N:M con

destinatari distinti.

In Figura 4.3 viene rappresentato il grafo di address coinvolti nell’Esempio 2.2.1, dove l’address “33mMAc6nGyENdKMQTr5SrKoEkwNTeZQUx9” rappresenta una particolare tipologia di indirizzo utilizzato da Bob, infatti l’address è ricavato da uno script P2WSH e come illustrato nella sezione 2.3.3 l’utilizzo di questo script viene introdotto per ridurre la complessità nella composizione di uno script P2MS; infatti Alice inserisce l’address di Bob all’interno lo script di blocco; Bob per sbloccare questa transizione oltre a fornire la sua firma deve inserire anche lo script di riscatto (illustrato nella sezione 2.3.3) all’interno lo script di sblocco.

Attraverso questo metodo offerto da Bitcoin Script, Bob può aumentare la sua privacy utilizzando un’indirizzo ricavato da uno script, il quale utilizza due address distinti appartenenti allo stesso wallet.

Gli address che iniziano per “bc1q” appartengono al medesimo wallet di Alice; i nodi corrispondenti sono collegati attraverso un arco corrispondente ad una exchange transaction.



Figura 4.3: Frammento del grafo di address corrispondente alla transazione illustrata nell’Esempio 2.2.1.

Address ricavati da chiavi pubbliche diverse La generazione delle exchange transaction contiene, nella maggior parte dei casi, un address ricavato da una chiave pubblica diversa, appartenente allo stesso wallet, come illustrato nel seguente esempio.

Esempio 4.3.2. Si consideri un nuovo scenario dove Alice spedisce dei bitcoin a Bob ed entrambi i partecipanti utilizzano address primitivi: Alice invia all’indirizzo di Bob 0.00336527 bitcoin, generando una nuova transazione con il seguente id “8b22a9d19ee58ee1-b5283632f70fbeceaf13948dbc3d48ea22c021a1d82e1f06” verso l’address di Bob “1CPD7DqgruujwvfaegKsV6ewiFvSZTiB3”. Il wallet di Alice utilizza un suo UTXO per effettuare la spedizione di bitcoin con un valore maggiore del necessario, quindi il wallet è costretto a generare una exchange transaction per dividere UTXO.

L’UTXO di Alice con un valore di bitcoin pari a 0.00682055 bitcoin si frammenta in due nuove transazioni:

- La transazione verso Bob del valore di 0.00336527 bitcoin.
- L’exchange transaction verso il wallet di Alice del valore di 0.00341851 bitcoin.

La Figura 4.4 rappresenta il grafo risultante degli address coinvolti.



Figura 4.4: Frammento del grafo degli address coinvolti nella transazione dell'esempio 4.3.2.

In Figura 4.4 si possono osservare tre address distinti originati da chiavi pubbliche distinte, dove però i proprietari sono solo Alice e Bob:

- Gli address di Alice sono:
 1. “1KEdN2XMuS2b8xVcHqjgJN7wwztTpPB1N3”, che rappresenta l’address di origine;
 2. “1JP5FmXVUCBCUhubMEoDmowpvQprchyP3q” che rappresenta l’address a cui viene indirizzata l’exchange transaction. Entrambi gli address appartengono al wallet di Alice, ma sono originati da chiavi pubbliche diverse.
- L’address di Bob inviato ad Alice per eseguire la spedizione di Bitcoin: “1CPD7DqguruujwvfaegKsV6ewiFvSZTiB3”.

Nelle prime versioni di Bitcoin, non tutti i wallet contenevano un set di chiavi private e questo comportava la creazione di exchange transaction verso il medesimo address, come mostrato nell’esempio 4.3.3.

Esempio 4.3.3. Prendiamo in esame la transazione con il seguente id “15bf8b35c9210efe7-e448c5fc6b69b47b3a8cac9c148c7cc57c65f266384d9b8” in cui possiamo osservare la presenza dell’address “1EinmJDn33yFPGafKuCw2guUqPSMaNKo5v” sia in output che in input. La Figura 4.5 rappresenta la transazione ricercata attraverso un esploratore blockchain.

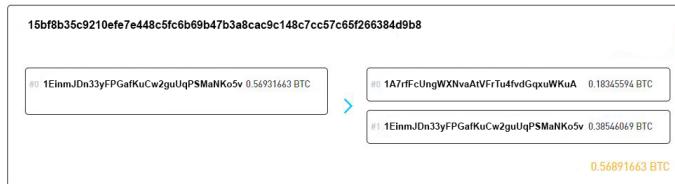


Figura 4.5: Una exchange transaction verso lo stesso indirizzo di origine[5].

In questo caso il flusso di bitcoin è chiaro e il corrispondente grafo degli address è mostrato nella Figura 4.6.

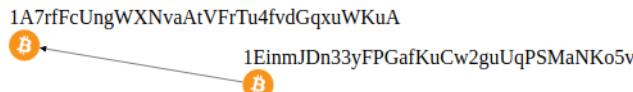


Figura 4.6: Il grafo degli address coinvolti nella transazione con id 15bf8b35c9210-efe7e448c5fc6b69b47b3a8cac9c148c7cc57c65f266384d9b8.

La costruzione del grafo degli address, oltre ad essere insidiosa per i problemi relativi alla frammentazione degli address, costringe ad accedere alle transazioni di output contenute all'interno delle transazioni, il cui riferimento risiede nelle transazioni di input della nuova transazione, per prelevare l'address di origine.

Capitolo 5

Stato dell'Arte

5.1 BlockSci

BlockSci [10] è un sistema open source, scritto in C++, per l’analisi di blockchain basate sull’organizzazione dati di Bitcoin. Sviluppato dall’Università di Princeton, BlockSci estrae i dati dalla blockchain di Bitcoin attraverso l’utilizzo del framework RPC di Bitcoin e di un parser per l’analisi delle informazioni serializzate dal nodo Bitcoin; i dati estratti vengono salvati in flat-file e interrogati attraverso SQLite. BlockSci, inoltre, offre un sofisticato sistema per l’analisi dei dati estratti con cui si ha la possibilità di implementare e applicare algoritmi di analisi sulla rete Bitcoin.

In questa sezione ci concentreremo solo sul modulo relativo al parser e sull’organizzazione delle informazioni estratte. Il parser di BlockSci legge le informazioni della blockchain sequenzialmente e, durante la lettura, applica una serie di ottimizzazioni per il corretto funzionamento del modulo di analisi. Il risultato della scansione sequenziale dei blocchi è un grafo delle transazioni minimale; il grafo è memorizzato in una singola tabella sequenziale, le cui entry hanno il formato mostrato in Tabella 5.1.

Transaction	
Size	Description
32 bit	Size
32 bit	LockTime
16 bit	Input count
16 bit	Output count
128 bits each	Outputs
128 bits each	Inputs

Tabella 5.1: Struttura delle transazioni in BlockSci per la costruzione del grafo delle transazioni [10].

In [10] Kalodner et al. descrivono i risultati di alcuni test relativi all’efficienza del parser; in tali test, il parser ha impiegato all’incirca 11 ore per effettuare l’estrazione dei

dati dalla blockchain di Bitcoin¹; aumentando la dimensione della memoria allocata a BlockSci si possono ottenere prestazioni migliori.

5.2 BiVA

BiVA (Bitcoin Network Visualization & Analysis) è un software sviluppato dall’Università di Singapore che utilizza Neo4J [21] per la costruzione di un grafo di address attraverso l’estrapolazione dei dati tramite il framework RPC di Bitcoin-core; esso, inoltre, implementa un algoritmo di analisi su grafo per la ricerca degli address più utilizzati all’interno della blockchain di Bitcoin.

In [22] Oggier et al. descrivono prevalentemente l’algoritmo di analisi utilizzato e non riportano dettagli sull’implementazione né risultati sperimentali relativi all’estrazione dei dati; tuttavia, poiché il framework RPC non è progettato per l’extrazione completa dei dati dalla blockchain, le tempistiche dovrebbero essere abbastanza elevate utilizzando un software *single thread*.

Una porzione di grafo prodotto attraverso BiVA è rappresentata in Figura 5.1.

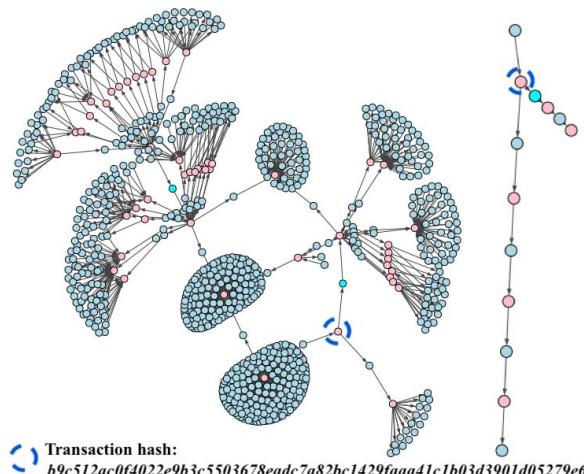


Figura 5.1: Frammento del grafo degli address creato attraverso BiVA [22].

5.3 Bitcoin Transaction Visualization

Uno studio condotto dall’Università di Saskatchewan e pubblicato in [26] descrive la creazione di un grafo di transazioni, localizzato attraverso l’utilizzo delle API di <https://blockchain.com>.

L’utilizzo di queste API ha permesso di localizzare le transazioni, attraverso la loro posizione geografica, grazie all’IP messo a disposizione per ogni transazione. Infatti <https://blockchain.com>:

¹Tali risultati fanno riferimento alla dimensione della blockchain assunta nell’agosto del 2017: 478,559 blocchi e 140 GB di spazio totale.

//blockchain.com arricchisce le informazioni estratte dalla blockchain di Bitcoin, aggiungendo informazioni circa il luogo di origine della transazione; quando questa informazione non è presente, verrà utilizzata la posizione dell'Università di Saskatchewan. La demo per la dimostrazione di questo studio viene sviluppata attraverso tecnologie web, con un uso estensivo di JQuery. La Figura 5.2 ne rappresenta un esempio.



Figura 5.2: Frammento del grafo di transazioni prodotto attraverso la demo pubblicata nell’articolo [26].

Capitolo 6

Tecnologie Utilizzate

6.1 Bitcoin-Cryptography-Library

La libreria Bitcoin-Cryptography-Library [20] ha permesso di importare la crittografia di base utilizzata in Bitcoin. Tale libreria è stata sviluppata in C++11 e in Java dallo sviluppatore Nayuki; la libreria risiede su Github sotto licenza MIT ed è progettata principalmente per microcontrollori ad 8 bit (e.g., Arduino, Atmel megaAVR/ATmega, ecc) con il supporto anche per dispositivi con architetture x86, x86-64 e 32-bit ARM.

6.2 RapidJSON

La libreria RapidJSON [29] ha permesso la decodifica in formato serializzato delle informazioni lette dal parser consentendo di creare una versione JSON della blockchain; tale libreria è scritta in C++ dal team di sviluppo Tencent ed è rilasciata su Github sotto licenza BSD. Con l'utilizzo di RapidJSON siamo riusciti ad introdurre all'interno del parser una deserializzazione quasi *real-time* di ogni blocco.

6.3 Bitcoin-api-cpp

La libreria bitcoin-api-cpp [19] implementa un wrapper per il framework RPC di Bitcoin-core; tale libreria risiede su Github sotto licenza MIT ed è stata implementata in C++ dallo sviluppatore Krzysztof Okupski. Con l'utilizzo di questa libreria siamo riusciti ad'interfacciarsi con il nodo Bitcoin per deserializzare le informazioni relative allo script di blocco.

6.4 JSON-RPC 1.0

Bitcoin-core offre un framework RPC (Remote Procedure Call) basato sul protocollo HTTP per consentire l'utilizzo dei servizi offerti dal nodo Bitcoin, quali, ad esempio, esaminare lo stato della blockchain oppure effettuare la crezione di transazioni in formato esadecimale.

Abbiamo utilizzato questo servizio offerto da Bitcoin-core per l'estrazione dell'address dallo script di blocco; esso, inoltre ci ha consentito di ricavare i dati della transazione precedente contenuta all'interno della transazione di input attualmente esaminata senza l'implementazione di una cache, a scapito, tuttavia, dell'efficienza della soluzione proposta.

6.5 Zlib

La libreria Zlib [17] è una famosa libreria di compressione open source, sviluppata da Jean-loup Gailly e Mark Adler in C; essa è divenuta uno standard RFC nel maggio 1996 ed è stata inserita all'interno del Java Development Kit a partire dalla versione 1.1. La libreria viene utilizzata per ridurre lo spazio richiesto dalle informazioni prodotte riguardanti i grafi.

6.6 Bitcoin core library

Sono state utilizzate alcune parti del codice sorgente di Bitcoin core per diminuire il quantitativo di test da produrre e soprattutto diminuire il numero di errori possibili durante il parsing dei dati.

Le librerie estratte sono:

- La libreria “serialize.h” con le relative dipendenze, con cui è stato possibile serializzare/deserializzare ogni singolo tipo di dato nel formato corretto, ad esempio: le informazioni vengono deserializzate in *little endian* e solo alcuni tipi di dato, come gli hash, vengono deserializzati in *big endian*.
- La libreria “endian.h” con le relative dipendenze, con cui è stato possibile gestire rappresentazione dei dati in formato little endian e big endian.
- La libreria “uint256.h” con le relative dipendenze, con cui è stato implementato il tipo di dato per rappresentare gli hash.

Tutte le librerie utilizzate ed estratte da Bitcoin core sono rilasciate sotto licenza MIT.

6.7 OpenMP

La libreria OpenMP (Open Multi-Processing) [6] è un API multipiattaforma per il supporto del multiprocessing della memoria condivisa per C/C++ e Fortran.

La version 5.0 della libreria rilasciata nel novembre 2018 supporta le versioni moderne del C++ come C++14 e C++17; essa viene sviluppata dal team di sviluppo OpenMP Architecture Review Board e la libreria viene inclusa nativamente all'interno delle principali versioni Unix-like.

Abbiamo utilizzato questa libreria per sviluppare una versione sperimentale del parser che utilizza più processori, perché fin dai primi utilizzi di esso, abbiamo notato il basso utilizzo di memoria RAM con un utilizzo massimo della potenza di un solo core di CPU.

Il modo con cui viene progettato il parser ci ha consentito di utilizzare un ciclo for che opera su multiprocessore per analizzare più file nello stesso momento (la motivazione per cui l'analisi parallela è possibile viene discussa nella Sezione 7.5).

6.8 Ngraph

Ngraph [11] rappresenta una famiglia di sottomoduli per la visualizzazione di grafi attraverso JavaScript; questi sottomoduli nascono dalla necessità di rendere versatile un noto progetto di visualizzazione open source per grafi denominato VivagraphJS [16].

La libreria VivagraphJS e i sottomoduli ngraph.* sono sviluppati interamente in JavaScript dallo sviluppatore Andrei Kashcha; risiedono su Github sotto diverse licenze open-source.

La libreria VivagraphJS dalla version 0.7 rappresenta una libreria per la visualizzazione

di grafi composta dai sottomoduli ngraph che consente il disegno del grafo attraverso la tecnologia SVG oppure la WebGL, quest’ultima ad oggi risulta essere difficile da personalizzare a livello di rendering, per questo motivo abbiamo deciso di comporre il nostro motore di rendering utilizzando diversi sottomoduli ngraph per consentire una maggiore personalizzazione di esso.

Il modulo principale utilizzato per creare l’applicativo di rendering è sicuramente ngraph.graph [12] che implementa la struttura dati per rappresentare il grafo.

Per implementare il motore di rendering vengono utilizzati due sottomoduli distinti che sono:

- ngraph.pixel [14], forse il modulo maggiormente ottimizzato per il rendering di grafi contenuto all’interno dei moduli ngraph, ngraph.pixel ci ha consentito di visualizzare il grafo di transazioni in 3D.
- ngraph.pixi [2], un modulo basato sul framework JavaScript PIXI.js [28], uno dei moduli di rendering 2D basato su WebGL più veloci della famiglia ngraph, ma anche il meno supportato, infatti abbiamo dovuto customizzare il progetto per aggiungere il supporto al rendering dei grafi diretti e il supporto per aggiunta delle label sui nodi. Al momento della stesura del documento il progetto è un fork separato del modulo ngraph.pixi ma è stato sottoposto a richiesta di unione con il progetto principale; il modulo viene utilizzato per visualizzare il grafo di address.

Vengono utilizzati inoltre i sottomoduli ngraph.native [1], ngraph.fromprecompute [24] e ngraph.tobinary [15] con cui è stato possibile effettuare il precalcolo del layout offline; abbiamo anche ottimizzato la velocità di esecuzione del modulo ngraph.native, ottimizzando la modalità di scrittura del C++ utilizzato.

Il modulo ngraph.fromprecompute viene utilizzato per il caricamento dei dati ottenuti dal modulo ngraph.native, questo modulo viene sviluppato da noi per la necessità di avere un modulo riutilizzabile.

Il modulo ngraph.tobinary converte il grafo ottenuto con il sottomodulo ngraph.graph in formato binario compatibile con il modulo ngraph.native.

Infine abbiamo utilizzato il modulo ngraph.louvain [15] che ci ha consentito di applicare facilmente l’algoritmo Louvain per calcolare le comunità all’interno del grafo visualizzato.

Capitolo 7

Soluzione Proposta

7.1 Introduzione

L'estrazione dei dati dalla blockchain di Bitcoin ha richiesto un analisi preliminare della tecnologia ed in particolare sul modo in cui vengono serializzati i dati dal client; questa prima analisi ha permesso di osservare la rapida crescita dello spazio richiesto della blockchain, rappresentata dalla Figura 7.1.

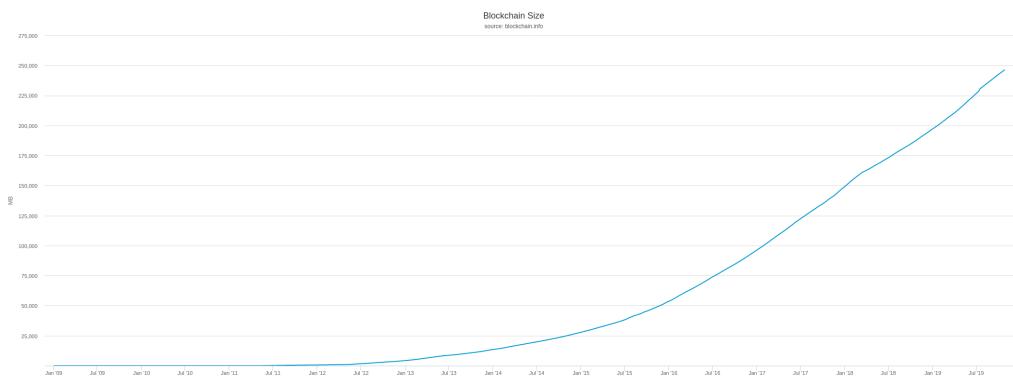


Figura 7.1: Crescita della dimensione della blockchain di Bitcoin nel tempo.

La rapida crescita della blockchain porta alle seguenti problematiche:

- L'estrazione dei dati dalla blockchain di Bitcoin richiede un software scalabile per permettere in qualsiasi condizione di utilizzo l'estrazione dei dati.
- L'organizzazione dei dati estratti della blockchain deve essere versatile per consentire analisi differenti, in modo tale da non richiedere una seconda scansione a seconda del tipo di analisi da effettuare.

7.2 SpyCBlock

SpyCBlock rappresenta la soluzione proposta per l'estrazione di dati dalla blockchain di Bitcoin in modo scalabile. Attraverso SpyCBlock viene effettuata la creazione dei due

principali grafi per l’analisi forense; SpyCBlock, inoltre, produce in formato JSON (JavaScript Object Notation) una deserializzazione completa della blockchain, arricchita con informazioni addizionali; infatti, come descritto nel Capitolo 2, i blocchi e le transazioni contengono solo gli identificativi dei loro predecessori, il che costringe il parser in fase di deserializzazione a ricostruire l’hash della transazione e del rispettivo blocco preso in analisi; questo è stato reso possibile dalla libreria “Bitcoin-Cryptography-Library” descritta nel Capitolo 6.1.

SpyCBlock rappresenta un prototipo di un software di analisi accademico, sviluppato interamente in C++14 e risiede su Github sotto licenza Apache License 2.0. Esso è l’implementazione di un parser delle informazioni serializzate dal nodo Bitcoin e utilizza le librerie di Bitcoin-core, descritte nella Sezione 6.6.

Il software è stato implementato utilizzando una metodologia agile; infatti lo sviluppo di SpyCBlock e lo studio della tecnologia Bitcoin sono avvenuti in parallelo. Questo ha permesso di convalidare tutte le nozioni studiate sulla tecnologia, basando l’intero ciclo di sviluppo sulla produzione di test di unità (9 batterie di test, per un totale di 46 test di unità).

I dati decodificati del singolo blocco vengono verificati all’interno dei test di unità utilizzando dati prototti attraverso l’utilizzo di altri parser, come Blocktools [30], e attraverso i dati esposti dagli explorer, come Explora [5] e Blockchain explorer[4].

Inoltre il parser è costituito da un sottomodulo per consentire la costruzione del grafo di address, utilizzando il nodo Bitcoin-core per effettuare la decodifica degli script.

7.3 SpyCBlockRPC

SpyCBlockRPC è un sottomodulo di SpyCBlock descritto nella Sezione 7.4; esso viene sviluppato in C++11 e risiede su Github sotto licenza Apache License 2.0.

SpyCBlockRPC rappresenta l’implementazione di un wrapper della libreria bitcoin-api-cpp descritta nella Sezione 6.3 che consente di effettuare interi casi d’uso costituiti da più comandi RPC di Bitcoin-core. L’utilizzo di questo sottomodulo permette al parser di rimanere disaccoppiato dalla libreria bitcoin-cpp-api e dal framework RPC; inoltre il sottomodulo fornisce un’interfaccia comune per la serializzazione di qualsiasi tipo di grafo che si voglia costruire usando le informazioni della blockchain.

SpyCBlock utilizza l’interfaccia comune offerta da SpyCBlockRPC per implementare la serializzazione del grafo di transazioni ed inoltre utilizza l’implementazione offerta dal sottomodulo SpyCBlockRPC per serializzare il grafo degli address.

7.4 Serializzazione della blockchain in formato JSON

Prima di effettuare la creazione dei grafi, descritti nelle sezioni successive, abbiamo dovuto affrontare varie problematiche che si sono presentate durante la fase di decodifica delle informazioni. Infatti, l’aggiornamento al Segregated Witness (descritto nella Sezione 3.3) ha introdotto alcune modifiche anche nel modo in cui viene serializzata una transazione; questo costringe il parser a utilizzare, a seconda del tipo di transazione, metodi diversi di deserializzazione. L’aggiornamento al Segregated Witness ha introdotto tre nuove informazioni all’interno delle transazioni, che sono:

- La proprietà flag.
- La proprietà Marker.
- Se le due proprietà precedenti sono rispettivamente 0 e 1, allora all'interno della transazione esisterà una lista di Script Witness, descritti nella Sezione 3.3, che vengono denominati “TransactionWitness”.

Il Frammento di codice descritto nell'Appendice A.1 rappresenta il codice con cui è possibile deserializzare una transazione dopo l'aggiornamento al Segregated Witness (viene usato Python per brevità).

I problemi relativi alla deserializzazione dei dati, combinati con la grande quantità di dati da deserializzare¹, hanno richiesto la progettazione di un metodo di testing automatico sull'intera blockchain oppure su un parte specifica. Questo ha richiesto di convertire le informazioni deserializzate dal parser in un formato universale per eseguire l'elaborazione di queste informazioni con qualsiasi linguaggio.

Il formato file utilizzato per la codifica delle informazioni è il formato JSON (JavaScript Object Notation) e la libreria RapidJSON descritta nella Sezione 6.2 ha reso possibile la serializzazione delle informazioni lette dal parser in maniera streaming; ogni blocco decodificato viene serializzato in JSON immediatamente con la conseguenza di un utilizzo di memoria RAM irrisorio.

L'adozione del formato JSON per l'organizzazione dei dati ha portato i seguenti benefici:

- I dati serializzati in JSON possono essere testati in maniera automatizzata attraverso l'uso di API esterne oppure attraverso il framework RPC di Bitcoin-core descritto nella Sezione 6.4.
- I dati serializzati in JSON possono essere testati a vari livelli di precisione. A titolo di esempio, un eventuale errore di deserializzazione in un determinato punto della blockchain può essere individuato facilmente e osservato attraverso il suo corrispettivo in formato JSON. Questo evita l'uso del debugger per errori introdotti in fase di sviluppo dal programmatore, perché essi sono facilmente individuabili se i dati sono convertiti in un formato comprensibile per l'uomo.
- La rappresentazione JSON dell'intera blockchain offre anche la possibilità di eseguire ulteriori tipi di analisi all'interno della blockchain di Bitcoin; ad esempio attraverso l'utilizzo di una semplice applicazione di analisi descritta nella Sezione 7.8, è possibile sfruttare i dati in formato JSON per un analisi sui tipi di script utilizzati all'interno della blockchain. In Figura 7.2 viene riportato un grafico che rappresenta l'utilizzo estensivo dell'operatore OP_RETURN (descritto nella Sezione 2.3.5) e quindi l'utilizzo della blockchain di Bitcoin per l'archiviazione di dati.

¹In data 14 novembre 2019 la dimensione della blockchain di Bitcoin è pari a 270 Gb.

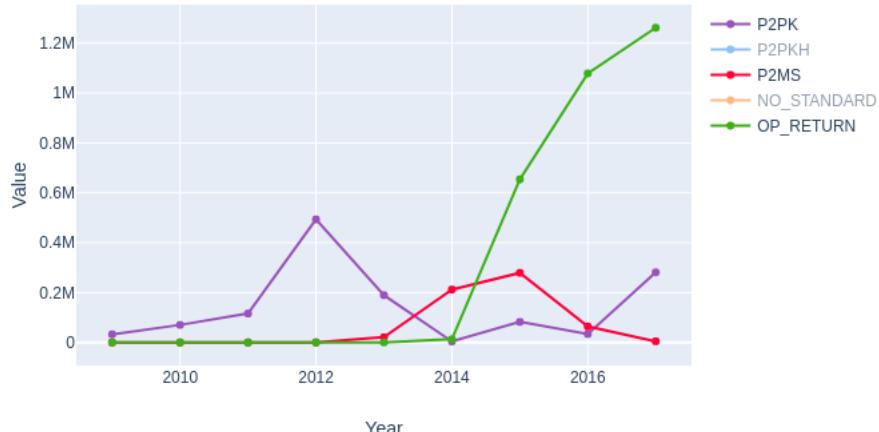


Figura 7.2: Frammento di grafico estrapolato dall'applicazione della Sezione 7.8.1 dove viene rappresentato l'utilizzo del operatore OP_RETURN all'interno degli script di blocco.

7.5 Grafo delle transazioni

Dopo un accurata convalida dei dati deserializzati dal parser, abbiamo eseguito una nuova iterazione all'interno del ciclo di sviluppo del software, aggiungendo all'interno del parser la possibilità di serializzare i dati letti come nodi e archi di un grafo orientato.

Ogni transazione, infatti, genera un arco (u, v) in cui il nodo di origine u rappresenta l'hash della transazione attualmente analizzata e il nodo di arrivo v rappresenta l'hash della transazione precedente, contenuta all'interno del tipo di dato “Outpoint” nella transazione di input (illustrato nella Sezione 3.3).

Come descritto nella Sezione 4.2, la creazione del grafo delle transazioni risulta abbastanza intuitiva, ma la struttura dati della blockchain di Bitcoin costringe il parser al calcolo dell'hash della transazione attualmente analizzata perché esso non viene incluso; per fare questo il parser riconverte in memoria i dati nel formato di serializzazione originario. In seguito alla riconversione con l'utilizzo della libreria Bitcoin-Cryptography-Library descritta nella Sezione 6.1, viene eseguito il doppio SHA256 dei bit riguardanti le informazioni in formato esadecimale, in cui ogni tipo della struttura viene prima convertito nel formato little-endian.

Esempio 7.5.1. Prendiamo in considerazione la transazione coinbase del blocco di genesi² rappresentata nella Figura 7.3.

²Il blocco di genesi è anche conosciuto come blocco 0 oppure blocco *never mined*, i cui dati sono stati codificati da Satoshi Nakamoto all'interno del codice sorgente di Bitcoin-core. Introdurre dei nuovi dati all'interno del blocco di genesi implica la creazione di una nuova blockchain.



Figura 7.3: Rappresentazione attraverso un'explorer [5] della transazione coinbase contenuta all'interno del blocco di genesi.

Analizzando i dati estratti dal parser possiamo notare che gli explorer non mostrano molte delle informazioni contenute all'interno delle transazioni; utilizzando invece il frammento di serializzazione JSON descritto nell'Appendice A.3 in cui viene raffigurata la prima transazione nella blockchain di Bitcoin, possiamo notare tutte le informazioni realmente contenute nella struttura di una transazione. Per effettuare il calcolo dell'hash della transazione il parser riconverte ogni tipo di dato nel formato little-endian e infine converte il valore nel corrispettivo esadecimale. La porzione di Codice 7.1 riporta il valore convertito con il processo descritto in precedenza.

Codice 7.1: Frammento di codice che rappresenta il valore esadecimale di ogni tipo di dato della transazione.

Infine il codice in Appendice A.4 riporta un test di unità del progetto SpyCBlock in cui si effettua il processo di creazione dell'hash completo.

Dopo aver ottenuto l'identificativo della transazione, il parser procede nella serializzazione della transazione come un arco (u, v) , aggiungendo delle informazioni addizionali all'arco, quali l'altezza del blocco calcolato durante il parsing; quest'ultima corrisponde alla sua posizione nella catena della blockchain. Inserendo il valore calcolato come informazione dell'arco si evita di inserire l'hash del blocco in cui risiede la transazione; questo implica un notevole risparmio di spazio delle informazioni serializzate. Inoltre, vengono

inseriti anche i valori corrispondenti al numero di bitcoin spediti e il valore nLockTime. La serializzazione delle informazioni avviene in streaming, cioè ogni blocco letto viene immediatamente convertito nel file contenente tutte le informazioni delle transazioni, ottenendo un formato come quello rappresentato dal Codice 7.2.

Codice 7.2: Informazioni della transazione in Figura 7.3 decodificata nell'arco corrispondente.

Attraverso la serializzazione delle transazioni con il formato appena descritto è stato possibile realizzare un applicativo web, chiamato SpyJSBlocks e descritto nella Sezione 7.8.2, per la visualizzazione del grafo utilizzando la libreria JavaScript descritta nella Sezione 6.8. Nella Figura 7.4 è rappresentata una porzione di grafo delle transazioni; da tale rappresentazione si può notare l'enorme quantità di transazioni contenute in una piccola parte di blockchain (50 Mb di 270 Gb) che rende difficile il rendering di esse. Attraverso alcuni zoom effettuati si può notare l'esistenza di piccole comunità di nodi che scambiano bitcoin; in particolare possiamo notare dallo zoom “B” alcuni nodi che fanno parte di transazioni M:1.

Esempio 7.5.2. Consideriamo lo zoom “B” della Figura 7.4. Possiamo notare 3 comunità di nodi, in cui solo uno di esso spedisce bitcoin ad un’altra comunità.

Prendendo in considerazione il nodo con identificativo di transazione “4e0b1f271c7add-530f8a4dd6ecdc3eb5ca417f0f587c628ecf4cb0b8b4bca40a”, si può notare attraverso l’uso di uno explorer che la transazione contiene 236 transazioni di input e 1 transazione di output, questo potrebbe indicare che il wallet abbia raggruppato tutte le transazioni su un unico address.

La blockchain di bitcoin serializza i blocchi all'interno di file chiamati "blk00000.dat" dove "00000" rappresenta l'indice dei file; Bitcoin-core stabilisce degli standard di dimensione per i file, pari a 100 Mb per ogni file.

SpyCBlock effettua la serializzazione 1 a 1, cioè per ogni file “blk0000.dat” produce un file con le informazioni in un specifico formato ad esempio “blk0000.json”; questo dà la possibilità di far evolvere facilmente il parser verso un’ implementazione parallela e/o distribuita.

7.6 Grafo degli address

Dopo avere implementato la costruzione del grafo delle transazioni, ci siamo concentrati sulla costruzione del grafo degli address. Come descritto nella Sezione 4.3, la costruzione di tale grafo introduce alcune problematiche che obbligano a rivedere il modo in cui si ricavano i dati durante il parsing delle informazioni.

Gli address sono infatti contenuti all'interno dello script di blocco di una transazione, come

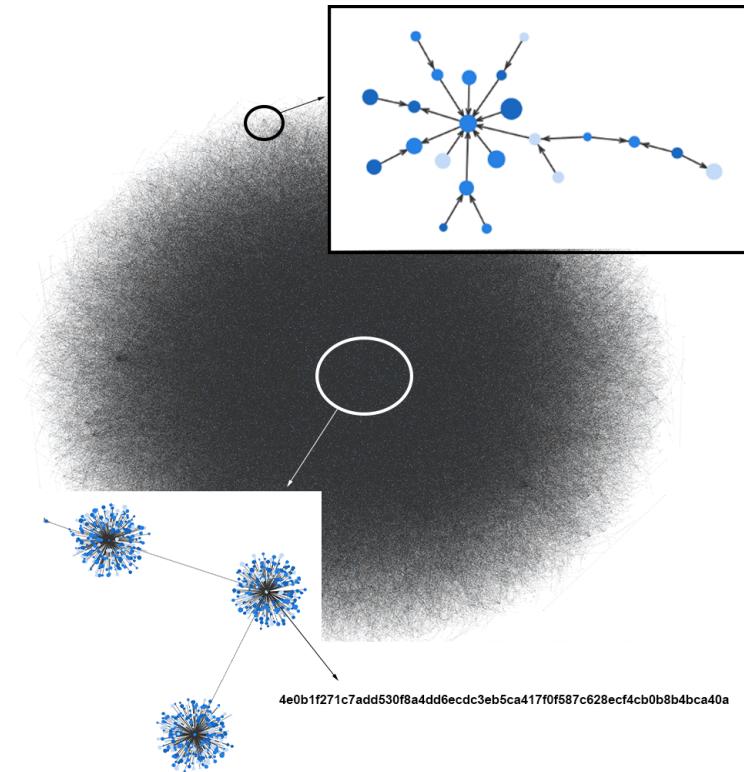


Figura 7.4: Porzione di grafo delle transazioni visualizzata attraverso SpyJSBlocks.

illustrato nelle Sezioni 2.3 e 3.4, mentre lo script di sblocco contiene solo le informazioni riguardanti le condizioni per cui un UTXO può essere sbloccato, non possedendo nessun riferimento all'indirizzo di origine. Per accedere allo script di blocco della precedente transazione, il cui identificativo (hash) risiede all'interno della transazione di input, appartenente alla transazione attualmente analizzata, bisogna memorizzare la cronologia delle transazioni decodificate e ricostruire un meccanismo per la gestione dell'UTXO, simile al meccanismo usato da Bitcoin-core: fornire un metodo di memorizzazione, che consente di prelevare una transazione precedentemente analizzata con costo $O(1)$ oppure con un costo logaritmico $O(N \log(N))$.

Al momento della stesura del documento, il parser non possiede nessun meccanismo di analisi degli script; essi con il passare del tempo sono diventati molto complessi e, come illustrato nella Sezione 3.4.3, sono stati introdotti metalinguaggi (come miniscript) per facilitare la scrittura di script “no standard”. Questo comporta la necessità di studiare in modo approfondito gli script per ottenere da essi maggiori informazioni.

Esempio 7.6.1. Consideriamo l'address “3CD1QW6fjgTwKq3Pj97nty28WZAVkziNom”, coinvolto in numerose truffe e illustrato all'interno dell'articolo [22]; tale address è originato da uno script (perché esso utilizza la convenzione del numero 3 come prima cifra) e potrebbe a sua volta contenere più address primitivi al suo interno e quindi rappresentare uno script P2MS oppure potrebbe rappresentare un address ricavato da uno script “no

standard”. Un esempio di “script no standard” è illustrato attraverso il Codice 3.6. Gli address originati da script, costringono ad un’analisi più approfondita per ottenere maggiori informazioni sulla loro vera identità; come illustrato nella Sezione 2.3.4, lo script P2SH è stato il primo esempio di script ad introdurre la possibilità di generare un address non primitivo, seguito dagli script P2WSH, P2WPKH e gli script “no standard”. Come possiamo osservare dalla Sezione 2.3.4, il processo di verifica di uno script P2SH costringe ad inserire all’interno dello script di sblocco anche lo script P2MS conosciuto come script di riscatto; questa informazione comporta l’analisi dello script di sblocco oltre che dallo script di blocco.

Considerando la transazione con identificativo

“82a69be69e0093791ab7d380369ecffba4f1fe0827a8625ede9d89e94776bc21”

bloccata attraverso lo script da cui è originato l’address “3CD1QW6fjgTwKq3Pj97nty28W-ZAVkziNom”, possiamo osservare che la transazione viene “consumata” da una nuova transazione con il seguente identificativo:

“457b5bde235dcf08de4159768c4f7abcb9697e1760a13e6caa15a7a2d8c90978”

Quest’ultima all’interno dello script di sblocco contiene le informazioni necessarie per lo sblocco della transazione bloccata dall’address “3CD1QW6fjgTwKq3Pj97nty28WZAVkziNom”.

Attraverso il Codice 7.3 possiamo osservare lo script di blocco da cui è originato l’address “3CD1QW6fjgTwKq3Pj97nty28WZAVkziNom”, dove possiamo osservare la condizione di sblocco di uno script P2SH in cui il valore “735d4de855597997b21588cc78ca2db696be1c5d” rappresenta l’HASH160 dello script P2MS.

Codice 7.3: Script da cui è originato l’address preso in esempio.

1 **OP_HASH160** 735d4de855597997b21588cc78ca2db696be1c5d **OP_EQUAL**

La condizione necessaria per sbloccare la transazione bloccata con lo script precedente è contenuta all’interno della transazione con identificativo

“457b5bde235dcf08de4159768c4f7abcb9697e1760a13e6caa15a7a2d8c90978”

Il Codice 7.4 riporta l’intero script di sblocco.

Codice 7.4: Script con cui è possibile eseguire con successo lo script illustrato attraverso il Codice 7.3.

1 **OP_0** 304402203638fc4c33f325d1c62c8feb4979d91300723f4766686e89de7380b269eec
2 c7602206d65886d69ccedc33de8f2840f27295b53dd9f9fd637b970664385b47c128da901
3 3045022100e8910b2162c08ec560d5737d09d361c07f90bfe58ee4e66a4fbebe246ea2b32c
4 02207fb1fc6296213b11ecb2eef3a375c8509776f75880ad5372417b55215fcdf74f01
5 **OP_PUSHDATA1**
6 522103385adff37fd3d0a620ebc4e9866e81dda8ba8616e5ebcae899c7f51899267
7 ae721034c08511718f947d1a3e152195c5e2756588e3e0c2c7730927eb6647af494210721033d
a9f8938a5b947a723df21b73fb3985b719249324d2c705acfb97d63a5df9e53ae

Dal precedente script possiamo estrarre lo script di riscatto contenuto dopo l'operatore OP_PUSHDATA1; il Codice 7.5 illustra lo script nel formato decodificato.

Codice 7.5: Readme Script contenuto all'interno dello script di sblocco 7.4.

1 **OP_2** 03385adff37fd3d0a620ebc4e9866e81dda8ba8616e5ebcae899c7f51899267ae7
 034c08511718f947d1a3e152195c5e2756588e3e0c2c7730927eb6647af4942107
 033da9f8938a5b947a723df21b73fdb3985b719249324d2c705acfb97d63a5df9e **OP_3**
 OP_CHECKMULTISIG

Dal Codice 7.5 si possono notare 3 chiavi pubbliche differenti:

- “03385adff37fd3d0a620ebc4e9866e81dda8ba8616e5ebcae899c7f51899267ae7”;
- “034c08511718f947d1a3e152195c5e2756588e3e0c2c7730927eb6647af4942107”;
- “033da9f8938a5b947a723df21b73fdb3985b719249324d2c705acfb97d63a5df9e”.

Le precedenti chiavi pubbliche rappresentano identità differenti appartenenti allo stesso wallet oppure a wallet differenti, utilizzando il codice descritto nell'Appendice A.4, possiamo ricavare un address primitivo dalle precedenti chiavi pubbliche, ottenendo i seguenti address:

- “14r7XjPtqVijLRhY9BkGAtDqVDp4txsK1X”;
- “1GryVw5pfa8a1Sc69PXywGLRDZJWc1C6wT”;
- “1G9VBMkqDzPmYxnFMkNxWGmfuQz1CHSYP6”.

Questo esempio mostra la necessità di analizzare attentamente gli address non primitivi, con i rispettivi script, perché potrebbero contenere informazioni rilevanti per la costruzione del grafo di address, in quanto quest'ultimo deve poter rappresentare il reale flusso di bitcoin tra gli address.

Riesaminando l'esempio appena descritto, dove l'address “3CD1QW6fjgTwKq3Pj97nty28W-ZAVkziNom” viene catalogato come address di *scam*, dopo aver condotto un'attenta analisi sullo script da cui è originato l'address si può concludere che lo script nasconde 3 chiavi pubbliche al suo interno.

Non effettuando quest'ultima analisi, ogni singolo address potrebbe operare singolarmente all'interno della blockchain di Bitcoin passando inosservato agli algoritmi di analisi.

Attraverso l'Esempio 7.6.1, possiamo notare come sia possibile costruire due tipi di grafi di address utilizzando le informazioni estratte dalla blockchain di Bitcoin:

- Il grafo naturale di address, che utilizza solo le informazioni contenute all'interno degli script di blocco. A titolo d'esempio, il frammento di grafo descritto dalla transazione illustrata nell'esempio 7.6.1 con identificativo “82a69be69e0093791ab7-d380369ecffba4f1fe0827a8625ede9d89e94776bc21” viene rappresentato dalla Figura 7.5.

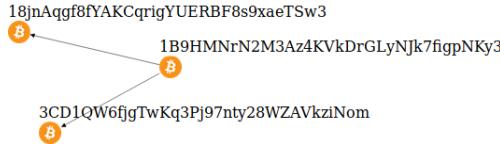


Figura 7.5: Frammento del grafo naturale degli address descritto dalla transazione con identificativo 82a69be69e0093791ab7d380369ecffba4f1fe0827a8625ede9d89e94776bc21.

Il grafo naturale di address risulta essere troppo superficiale, perché le chiavi pubbliche che fanno parte dello script da cui è ottenuto l'address “3CD1QW6fjgTwKq3Pj-97nty28WZAVkziNom” potrebbero essere riutilizzate singolarmente all'interno della blockchain di Bitcoin passando inosservate agli algoritmi di analisi;

- Il grafo degli address ottenuto con un'analisi degli indirizzi originati da script. A titolo d'esempio, il frammento di grafo illustrato nella Figura 7.5 in cui viene condotta un'analisi dello script da cui viene originato l'address “3CD1QW6fjgTwKq3Pj-97nty28WZAVkziNom” viene illustrato nella Figura 7.6



Figura 7.6: Frammento del grafo degli address ottenuto attraverso un analisi dell'address 3CD1QW6fjgTwKq3Pj97nty28WZAVkziNom.

Il grafo di address appena descritto risolve la problematica riguardante le chiavi pubbliche, contenute all'interno dello script, con la conseguenza della perdita del riferimento riguardante l'address originato dallo script, il che aumenta notevolmente il grado di complessità del grafo.

Una possibile soluzione, potrebbe essere quella di utilizzare una rappresentazione attraverso il grafo naturale rappresentato dalla Figura 7.5, con l'aggiunta delle chiavi pubbliche ricavate dallo script come proprietà del nodo, ottenendo la rappresentazione illustrata in Figura 7.7



Figura 7.7: Frammento del grafo naturale degli address arricchito delle informazioni delle chiavi pubbliche all'interno del nodo.

In seguito allo studio di fattibilità sulla creazione del grafo, ci siamo limitati alla costruzione di un grafo semplificato, anche perché oltre alle problematiche descritte abbiamo

dovuto affrontare anche le problematiche sulla serializzazione delle transazioni M:N (molti a molti): ogni transazione può contenere, infatti, M transazioni di input e N transazioni di output.

Questa caratteristica riguardante il numero di transazioni può essere risolta utilizzando una rappresentazione a multigrafo degli address. In questo documento, proponiamo una soluzione in cui si forza l'utilizzo di un comune grafo, in cui ogni transazione M:N viene serializzata in maniera iterativa: la prima transazione di input viene serializzata con tutte le transazioni di output e così a seguire, per le restanti transazioni di input, ignorando per il momento il problema delle chiavi pubbliche contenute all'interno degli script da cui si originano address. Un frammento del grafo rappresentante una transazione M:N è illustrato dalla Figura 7.8

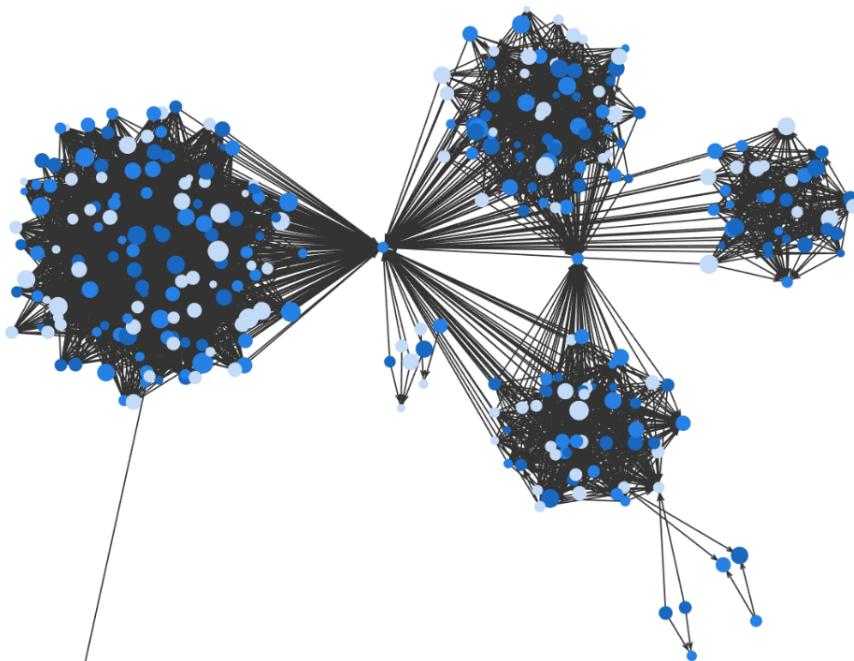


Figura 7.8: Frammento del grafo degli address, in cui viene rappresentata una serializzazione iterativa di una transazione M:N.

La costruzione del grafo attraverso SpyCBlock avviene grazie all'introduzione del sottomodulo SpyCBlockRPC descritto nella Sezione 7.3, il quale si occupa di accedere alle informazioni della transazione precedente tramite l'utilizzo del framework RPC di Bitcoin-core descritto nella Sezione 6.4. Il parser inoltre utilizza il framework RPC per decodificare lo script di blocco, da cui si ottiene l'address oppure gli address contenuti al suo interno. Il framework RPC di Bitcoin-core non essendo adatto all'estrazione completa delle informazioni dalla blockchain di Bitcoin limita le prestazioni del parser durante l'analisi delle informazioni.

Questo problema potrebbe essere risolto estendendo l'implementazione di SpyCBlockRPC ad un implementazione *multi thread* oppure implementando un decompilatore per gli

script di Bitcoin; quest’ultima soluzione implica anche l’implementazione di un metodo di memorizzazione per mantenere una cronologia delle transazioni decodificate dal parser.

Una porzione di grafo degli address viene rappresentata dalla Figura 7.9

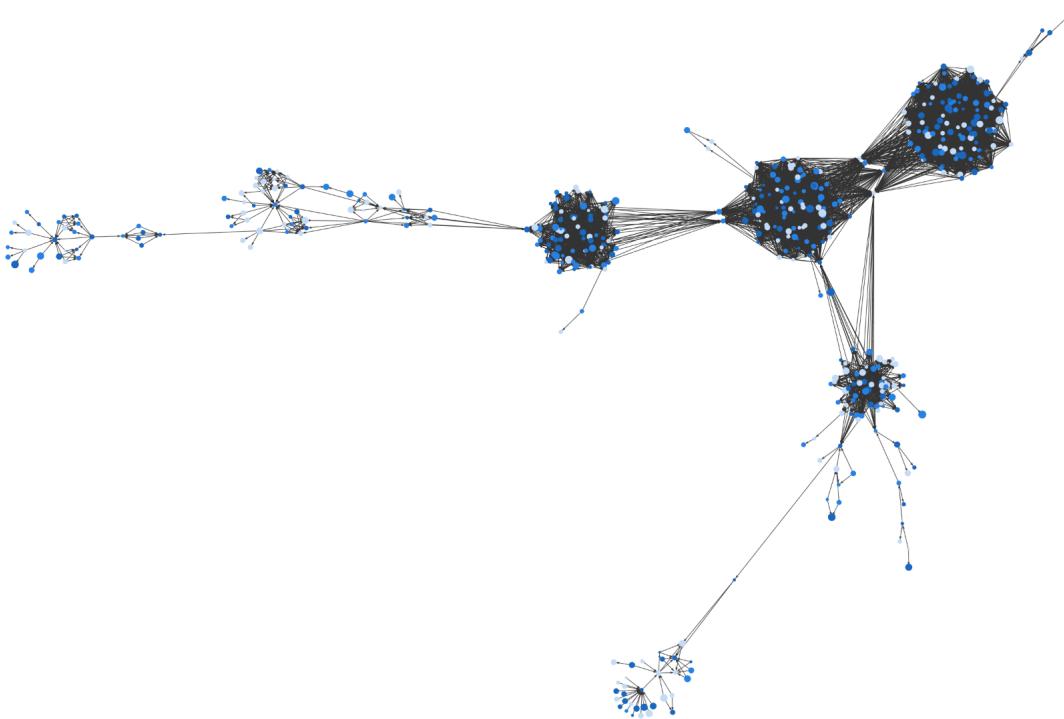


Figura 7.9: Frammento del grafo degli address estrapolato dalla demo descritta nella Sezione 7.8.2.

7.7 Risultati Sperimentali

La soluzione proposta si è limitata all’estrazione dei dati dalla blockchain di Bitcoin in maniera scalabile con una successiva riconversione dei dati tale da consentire la costruzione di un grafo di transazioni, oltre alla serializzazione completa in JSON della blockchain. Il parser proposto come soluzione oltre a serializzare il grafo di transazioni implementa anche una soluzione sperimentale per la creazione del grafo di address attraverso l’ausilio del framework RPC di Bitcoin descritto nella Sezione ??.

La valutazione sperimentale condotta si limita ad analizzare le tempistiche di analisi solo per la serializzazione in JSON della blockchain e per la serializzazione del grafo di transazioni, ignorando le tempistiche di serializzazione del grafo di address perché risultano essere molto elevate a causa della dipendenza diretta con il framework RPC.

L’analisi viene condotta eseguendo il parser su un laptop con una CPU DESCRIVI LA CPU,

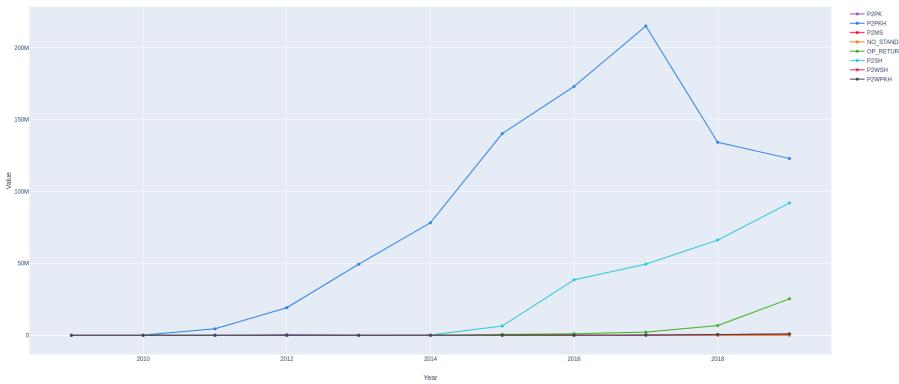
7.8 DEMO

7.8.1 AnalyticsPyBlock

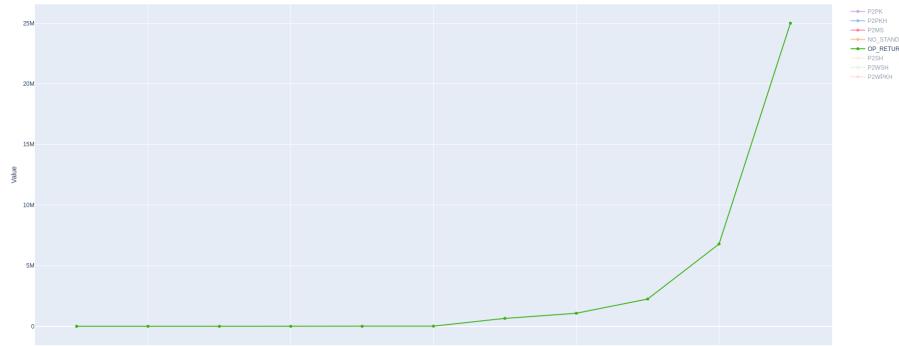
Per dimostrare un possibile utilizzo della blockchain di Bitcoin serializzata in JSON è stata sviluppato un semplice strumento di analisi denominato AnalyticsPyBlock, il quale permette anche di sviluppare analisi personalizzate.

AnalyticsPyBlock al momento della stesura del documento supporta solo un tipo di analisi, riguardante la tipologia di script utilizzati nel tempo nella rete Bitcoin; il software di analisi viene sviluppato in Python3 e risiede su Github sotto licenza Apache License 2.0.

La Figura 7.10 illustra il risultato dell’analisi sulle tipologie di script.



(a) Grafico che rappresenta gli script più utilizzati all’interno la blockchain di Bitcoin.



(b) Grafico che rappresenta l’utilizzo dello script OP_RETURN per archiviare dati.

Figura 7.10: Grafici estratti dalla demo descritta nel seguente capitolo.

La Figura 7.10(a) mostra gli script maggiormente utilizzati, che risultano essere P2PKH e P2SH; inoltre la Figura 7.10(b) mostra l’utilizzo della blockchain di Bitcoin per l’archiviazione di dati. Questa informazione è molto utile ai fini della costruzione del grafo perché attraverso una futura operazione di *data cleaning* si possono eliminare tutte le informazioni che riguardano l’archiviazione di dati attraverso l’operatore OP_RETURN.

La strumento impiega meno di 12 ore per analizzare i file JSON serializzati da SpyCBlock;

anche tale tool potrebbe essere migrato ad un’implementazione multiprocessore con cui si può migliorare la velocità di analisi.

7.8.2 SpyJSBlocks

Per visualizzare i grafi serializzati da SpyCBlock abbiamo sviluppato un applicazione web che fa uso delle librerie appartenenti alla famiglia ngraph descritte nella Sezione 6.8, con cui siamo riusciti ad ottenere una visualizzazione di una piccola parte dei grafi serializzati da SpyCBlock.

Attraverso la possibilità di visualizzare i grafi, siamo riusciti ad eseguire tutte le analisi discusse in questo documento; inoltre con l’utilizzo del modulo ngraph.louvian abbiamo condotto un’analisi minimale sul grafo degli address scomponendo quest’ultimo in comunità, anche se l’applicazione di tale algoritmo soffre delle problematiche descritte nella Sezione 7.6.

L’applicativo web viene sviluppata utilizzando JavaScript senza l’ausilio di framework; una possibile evoluzione di questo applicativo potrebbe essere la migrazione ad un framework come React [8].

Lo strumento software possiede due diverse implementazioni per via delle diverse problematiche riguardanti le librerie ngraph descritte nella Sezione 6.8; l’applicativo risiedono su Github sotto licenza Creative Commons 4.0 e vengono denominati SpyJSBlock [25] e SpyJSBlock-NGraph [9]. Le principali differenze tra gli strumenti software sono:

- SpyJSBlock è una prima implementazione dell’applicativo web che fa uso della libreria VivagraphJS descritta nella sezione 6.8, per via della difficoltà nel personalizzare il rendering tramite WebGL non viene eseguita nessuna personalizzazione eccetto l’aggiunta della possibilità di esplorare il nodo del grafo attraverso eventi di click che offrono la possibilità di ricercare l’id della transazione oppure l’address attraverso un’explorer come Explora [5].

Non utilizzando nessuna particolare personalizzazione il rendering risulta essere molto veloce, riuscendo a visualizzare una porzione di grafo molto più grande, ma risulta essere difficoltoso cambiare la modalità di rendering oppure applicare algoritmi di analisi sul grafo.

Un frammento di grafo visualizzato attraverso SpyJSBlock viene illustrato attraverso la Figura 7.12.

Per personalizzare il rendering ed avere più controllo sulla visualizzazione del grafo abbiamo sviluppato un’alternativa utilizzando le librerie della famiglia ngraph.

- SpyJSBlock-NGraph è un raffinamento dello strumento SpyJSBlock, dove vengono utilizzati i moduli della famiglia ngraph per il rendering dei grafi, con cui si riesce ad ottenere un miglioramento nella qualità del rendering a discapito però delle performance: avendo la possibilità di renderizzare una porzione più piccola del grafo. Viene utilizzato il modulo ngraph.pixel per il rendering in 3D del grafo delle transazioni, in cui viene implementata anche la possibilità di ricercare l’id della transazione attraverso l’explorer Explora, La Figura ?? illustra un frammento del grafo delle transazioni. Per il rendering del grafo degli address invece viene utilizzato il modulo ngraph.pixi con cui si riesce ad ottenere una visualizzazione 2D di un grafo diretto, La Figura 7.13(a) ne illustra un esempio. Inoltre sulla tipologia del grafo di

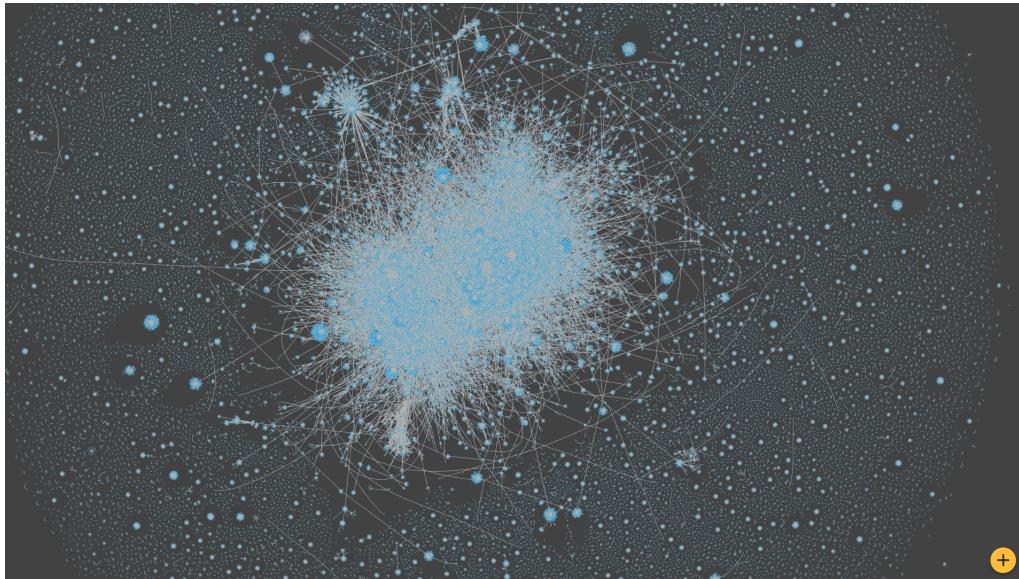


Figura 7.11: Frammento del grafo visualizzato attraverso l'applicativo SpyJSBlock.

address viene applicato un algoritmo di ricerca delle comunità attraverso il modulo ngraph.louvain descritto nella Sezione 6.8.

Dopo l'applicazione dell'algoritmo, le comunità di nodi vengono dipinte con colori diversi, semplificando l'individuazione di quest'ultime all'interno del grafo. La Figura 7.13(b) illustra una composizione in comunità del grafo di address.

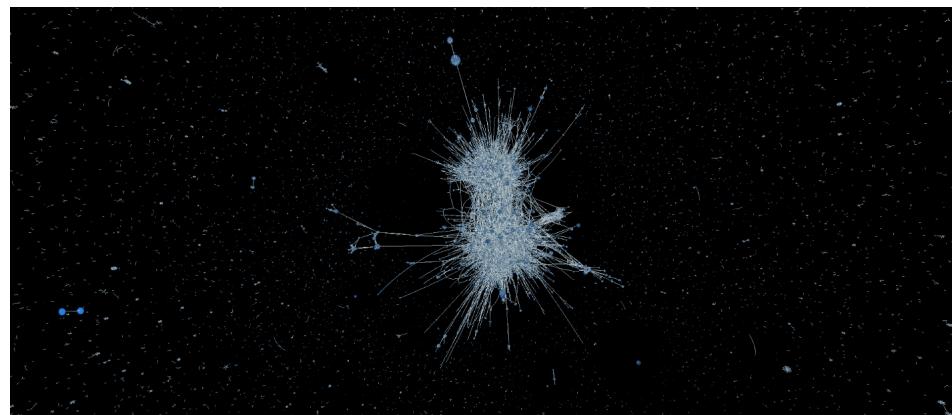
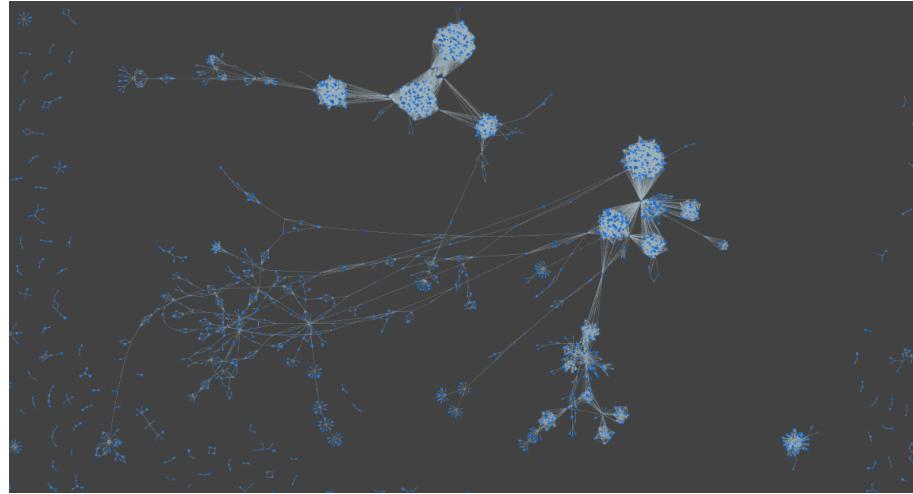
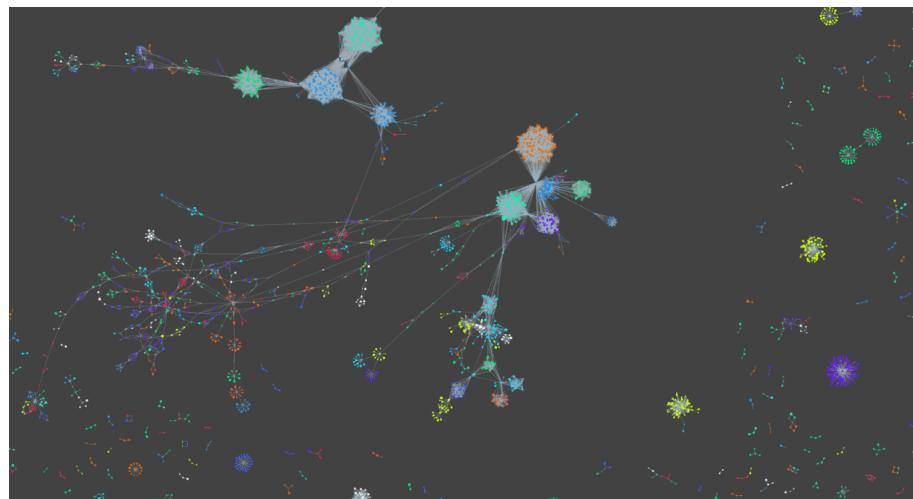


Figura 7.12: Frammento del grafo visualizzato attraverso l'applicativo SpyJSBlock.



(a) Grafo di address visualizzato attraverso il modulo ngraph.pixi[2].



(b) Grafo di address scomposto in comunità attraverso l'utilizzo del modulo ngraph.louvian[13]

Figura 7.13: Frammenti di visualizzazione dei grafi estrapolati dalla demo descritta nel seguente capitolo.

Capitolo 8

Conclusioni

Il lavoro condotto in questa tesi ha evidenziato in particolare la possibilità di analizzare la blockchain di Bitcoin in modo scalabile, inoltre questo studio ha approfondito lo l'analisis del linguaggio Bitcoin Script e sul modo in cui una transazione Bitcoin può essere spesa, facendo emergere la necessità di approfondire gli studi in questo campo.

I risultati ottenuti attraverso lo sviluppo del parser descritto in questo documento, sono andati ben oltre le nostre aspettative, perché dopo l'analisi dei numerosi esempi illustrati in questo documento, come BlockSci, non pensavamo di riuscire a ricavare un prototipo di un software scalabile e soprattutto che riuscisse ad essere eseguibile in macchine comuni eseguendo l'analisi con tempi ragionevoli.

Guardando alle possibili estensioni del lavoro svolto, si può pensare subito di evolvere il parser ad un'implementazione completa multiprocessore, ciò migliorerebbe notevolmente le prestazioni di deserializzazione.

Inoltre attraverso l'utilizzo del prototipo SpyCBlock sono emerse una serie di problematiche riguardante la costruzione del grafo degli address, questo lascia spazio ad un implementazione di un sottomodulo per la deserializzazione approfondita degli script utilizzati in Bitcoin. Questo implica anche lo sviluppo di un metodo per la memorizzazione delle transazione deserializzate dal parser; con l'apporto di queste due modifica si potrebbe lavorare solo con i file serializzati dal nodo Bitcoin-core senza l'utilizzo del framework RPC.

Ringraziamenti

Grazie a Grazia, Graziella e la sorella.

Appendice A

Appendice

A.1 Decodifica Transazione con Segregated Witness

Codice A.1: Porzione di codice estrapolato da [30] che riporta il metodo di deserializzazione della transazione dopo l'aggiornamento al Segregated Witness.

```
1 def __init__(self, blockchain):
2     self.version = uint4(blockchain)
3     self.inCount = varint(blockchain)
4     self.isWitness = False
5     if self.inCount == 0x00:
6         self.mark = 0
7         self.flag = varint(blockchain)
8         if self.flag == 0x01:
9             self.isWitness = True
10            self.inCount = varint(blockchain)
11        self.inputs = []
12        self.seq = 1
13        for i in range(0, self.inCount):
14            input = txInput(blockchain, self.isWitness)
15            self.inputs.append(input)
16        self.outCount = varint(blockchain)
17        self.outputs = []
18        if self.outCount > 0:
19            for i in range(0, self.outCount):
20                output = txOutput(blockchain)
21                self.outputs.append(output)
22        self.witness = []
23        if self.isWitness:
24            for i in range(0, self.inCount):
25                singleWitness = txWitness(blockchain)
26                self.witness.append(singleWitness)
27        self.lockTime = uint4(blockchain)
```

A.2 Frammento di codice estratto dalla serializzazione in JSON di SpyCBlock

Codice A.2: Frammento della serializzazione in JSON del blocco di genesi riguardante la prima transazione all'interno la blockchain di Bitcoin.

A.3 Processo per la creazione dell'hash rappresentato in un test di unità

Codice A.3: Processo di creazione dell'hash della transazione estratto dai test di unità di SpayCBlock.

```
1 TEST(hash_test, first_test_comparable_transaction_hash_value_readed)
2 {
3     string pathMockRoot =
4         ConfiguratorSingleton::getInstance().getPathFileMockTest() + "/";
5     Block block;
```

```
6     std::ifstream fileOut(pathMockRoot.append("bitcoin/block blk00000.dat"));
7
8     block.decode(fileOut);
9     fileOut.close();
10
11    string hexForm = SerializationUtil::toSerealizeForm(
12        block.getRawTransactions().at(0).getVersion());
13
14    hexForm += SerializationUtil::toSerealizeForm(
15        block.getRawTransactions().at(0).getNumberTxIn());
16
17    hexForm += block.getRawTransactions().at(0)
18        .getTxIn().at(0).getOutpoint().getHash().GetHex();
19
20    hexForm += SerializationUtil::toSerealizeForm(
21        block.getRawTransactions().at(0)
22            .getTxIn().at(0).getOutpoint().getN());
23
24    hexForm += SerializationUtil::toSerealizeForm(
25        block.getRawTransactions().at(0).getTxIn().at(0)
26            .getScript().getScriptLenght());
27
28    hexForm += block.getRawTransactions().at(0).getTxIn().at(0)
29        .getScript().getRawScriptString().substr(0,
30            block.getRawTransactions().at(0).getTxIn().at(0).getScript().getScriptLenght() *
31            2);
32
33    hexForm += SerializationUtil::toSerealizeForm(
34        block.getRawTransactions().at(0).getTxIn().at(0).getSequences());
35
36    hexForm += SerializationUtil::toSerealizeForm(
37        block.getRawTransactions().at(0).getNumberTxOut());
38
39    hexForm += SerializationUtil::toSerealizeForm(
40        block.getRawTransactions().at(0).getTxOut().at(0).getNValue());
41
42    hexForm += SerializationUtil::toSerealizeForm(
43        block.getRawTransactions().at(0).getTxOut().at(0).getScript().getScriptLenght());
44
45    hexForm += block.getRawTransactions().at(0).getTxOut().at(0)
46        .getScript().getRawScriptString().substr(0,
47            block.getRawTransactions().at(0).getTxOut().at(0)
48                .getScript().getScriptLenght().getValue() * 2);
49
50    hexForm +=
51        SerializationUtil::toSerealizeForm(block.getRawTransactions().at(0).getLockTime());
```

```
52     vector<unsigned char> vectorByte =
53         spyCBlock::UtilCrypto::ToHexIntoVectorByte(hexForm);
54
55     Sha256Hash shaHash = Sha256::getDoubleHash(vectorByte.data(),
56         vectorByte.size());
57
58     EXPECT_EQ(shaHash.ToStringForProtocol(),
59         "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b");
59 }
```

A.4 Processo per la creazione dell'address primitivo da una chiave pubblica

Codice A.4: Processo per la creazione dell'address primitivo da una chiave pubblica [23].

```
1 std::string buildAddressFromPubKey(std::string pubkey){
2     opcode = hex.substr(hex.length() - 2, 2);
3     optValue = std::stoul(opcode, nullptr, 16);
4     optMap = bitcoinOpCode.opCodeList.find(optValue);
5     opcode = optMap->second;
6     Bytes bytes = hexBytes(key.c_str());
7     //SHA256
8     Sha256Hash shaHash = Sha256::getHash(bytes.data(), bytes.size());
9
10    uint8_t result[Ripemd160::HASH_LEN];
11    Ripemd160::getHash(shaHash.value, sizeof(shaHash), result);
12
13    char address[36];
14    Base58Check::pubkeyHashToBase58Check(result, 0x00, address);
15    return std::string(address);
16 }
```

Elenco delle figure

1.1	Esempio di stato non valido per un cammino radice foglia. [27]	3
2.1	La nuova transazione creata da Alice ricercata attraverso un explorer [5].	5
2.2	Gestione degli UTXO nella blockchain di Bitcoin [5].	5
2.3	Transazione coinbase descritta [5].	6
2.4	La transazione creata da Bob verso Sara [5].	7
2.5	Rappresentazione del evento di blocco e sblocco di un UTXO.	7
2.6	Stato iniziale dello stack.	10
2.7	Stato dello stack dopo l'esecuzione dell'operatore OP_DUP.	11
2.8	Stato dello stack dopo l'esecuzione dell'operatore OP_HASH160.	11
2.9	Stato dello stack dopo l'inserimento della chiave pubblica attesa.	11
2.10	Stato dello stack dopo l'inserimento dello scriptSig.	12
2.11	Stato dello stack dopo l'inserimento del <public key>.	12
2.12	Rappresentazione del processo di creazione dello script P2PK e P2PKH.	13
2.13	Stato dello stack dopo l'inserimento dello scriptSig nello stack.	13
2.14	Stato dello stack dopo la verifica con OP_2 e l'inserimento nello stack delle chiavi pubbliche.	14
2.15	Esecuzione dell'operatore OP_CHECKMULTISIG per la verifica delle chiavi [32].	14
2.16	Stato dello stack dopo l'esecuzione dello ScriptSig come script P2MS.	15
2.17	Stato dello stack dopo l'esecuzione dell'operatore OP_HASH160.	16
2.18	Stato dello stack dopo l'inserimento dell'hash atteso.	16
2.19	Grafico della crescita della potenza di hashing in data 28/09/2019. Fonte: blockchain.com.	18
2.20	Grafico della crescita della difficoltà metrica in data 28/09/2019. Fonte: blockchain.com.	18
3.1	Confronto tra le tecnologie Lightning network e Bitcoin: transazioni per secondo (tps) [7].	25
4.1	Frammento del grafo delle transazioni relativo alla transazione tra Alice e Bob descritta nell'Esempio 2.2.1.	29
4.2	Frammento del grafo delle transazioni relativo alla transazione tra Alice e Bob descritta nell'Esempio 2.2.1.	31

4.3	Frammento del grafo di address corrispondente alla transazione illustrata nell'Esempio 2.2.1.	32
4.4	Frammento del grafo degli address coinvolti nella transazione dell'esempio 4.3.2.	33
4.5	Una exchange transaction verso lo stesso indirizzo di origine[5].	33
4.6	Il grafo degli address coinvolti nella transazione con id 15bf8b35c9210-efe7e448c5fc6b69b47b3a8cac9c148c7cc57c65f266384d9b8.	33
5.1	Frammento del grafo degli address creato attraverso BiVA [22].	36
5.2	Frammento del grafo di transazioni prodotto attraverso la demo pubblicata nell'articolo [26].	37
7.1	Crescita della dimensione della blockchain di Bitcoin nel tempo.	41
7.2	Frammento di grafico estrapolato dall'applicazione della Sezione 7.8.1 dove viene rappresentato l'utilizzo del operatore OP_RETURN all'interno degli script di blocco.	44
7.3	Rappresentazione attraverso un'explorer [5] della transazione coinbase contenuta all'interno del blocco di genesi.	45
7.4	Porzione di grafo delle transazioni visualizzata attraverso SpyJSBlocks.	47
7.5	Frammento del grafo naturale degli address descritto dalla transazione con identificativo 82a69be69e0093791ab7d380369ecffba4f1fe0827a8625ede9d89e9-4776bc21.	50
7.6	Frammento del grafo degli address ottenuto attraverso un analisi dell'address 3CD1QW6fjgTwKq3Pj97nty28WZAVkziNom.	50
7.7	Frammento del grafo naturale degli address arricchito delle informazioni delle chiavi pubbliche all'interno del nodo.	50
7.8	Frammento del grafo degli address, in cui viene rappresentata una serializzazione iterativa di una transazione M:N.	51
7.9	Frammento del grafo degli address estrapolato dalla demo descritta nella Sezione 7.8.2.	52
7.10	Grafici estrapolati dalla demo descritta nel segunte capitolo.	53
7.11	Frammento del grafo visualizzato attraverso l'applicativo SpyJSBlock.	55
7.12	Frammento del grafo visualizzato attraverso l'applicativo SpyJSBlock.	55
7.13	Frammenti di visualizzazione dei grafi estrapolati dalla demo descritta nel segunte capitolo.	56

Elenco delle tabelle

1.1	Rappresentazione generale del concetto di blocco in una blockchain.	2
2.1	Struttura delle transazioni in Bitcoin.	8
2.2	Rappresentazione struttura delle transazioni di output.	8
2.3	Rappresentazione struttura delle transazione di input in Bitcoin.	9
3.1	Struttura di un block header in Bitcoin core.	19
3.2	Struttura di un blocco in Bitcoin core.	20
3.3	Struttura della transazione dopo l'aggiornamento al Segregated witness. . .	21
3.4	La struttura della transazione input all'interno di Bitcoin core.	22
3.5	Struttura del tipo di dato Outpoint di Bitcoin core.	22
3.6	La struttura della transazione di output di Bitcoin core.	22
3.7	La struttura della serializzazione dei dati del testimone; in Bitcoin core questa struttura è chiamata CScriptWitness.	23
3.8	Struttura del tipi di dato CScript di Bitcoin core.	23
5.1	Struttura delle transazioni in BlockSci per la costruzione del grafo delle transazioni [10].	35

Lista dei codici

2.1	Porzione di codice che riporta il tipo enumerazione nel file standard.h di bitcoin core.	9
2.2	P2PKH Script di blocco.	10
2.3	P2PKH Script di sblocco.	10
2.4	Script completo.	10
2.5	P2PK Script completo.	12
2.6	Script P2MS completo.	13
2.7	Script P2SH completo.	15
2.8	Script P2SH di sblocco.	15
2.9	Script P2SH di blocco.	15
2.10	Indirizzo Bitcoin P2SH.	16
2.11	Indirizzo Bitcoin primitivo.	16
2.12	Script P2SH di blocco con Indirizzo Bitcoin P2SH.	16
2.13	Script P2SH completo con indirizzo Bitcoin P2SH.	16
2.14	Uso dell'operatore OP_RETURN.	17
3.1	Esempio generale della struttura di uno script P2WPKH.	24
3.2	Esempio reale di uno script P2WPKH.	24
3.3	Esempio di uno script P2WSH.	24
3.4	Address base32 della rete mainet.	24
3.5	Address base32 della rete testnet.	24
3.6	Script di Blocco che esprime una multipla condizione di spesa utilizzando OP_CHECKLOCKTIMEVERIFY.	26
3.7	Clausola if-then-else in Bitcoin script.	26
3.8	Uno script complesso che utilizza l'operatore OP_CHECKSEQUENCEVERIFY.	27
3.9	Un esempio di utilizzo di miniscript.	28
7.1	Frammento di codice che rappresenta il valore esadecimale di ogni dipo di dato della transazione.	45
7.2	Informazioni della transazione in Figura 7.3 decodificata nell'arco corrispondente.	46
7.3	Script da cui è originato l'address preso in esempio.	48
7.4	Script con cui è possibile eseguire con successo lo script illustrato attraverso il Codice 7.3.	48
7.5	Readme Script contenuto all'interno dello script di sblocco 7.4.	49

Bibliografia

- [1] Vincenzo Palazzo Andrei Kashcha. *ngraph.native*. URL: <https://github.com/anvaka/ngraph.native>.
- [2] Vincenzo Palazzo Andrei Kashcha. *ngraph.pixi*. URL: <https://github.com/vincenzopalazzo/ngraph.pixi>.
- [3] Andreas M. Antonopoulos. *Mastering Bitcoin 2nd Edition - Programming the Open Blockchain*. O'Reilly, 2017.
- [4] Blockchain. *Explorer*. URL: <https://www.blockchain.com/explorer>.
- [5] Blockstream. *Explora*. URL: <https://github.com/Blockstream/esplora>.
- [6] Leonardo Dagum e Ramesh Menon. «OpenMP: an industry standard API for shared-memory programming». In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.
- [7] DataLight. *Lightning Network Study. Will this technology become the new standard for Bitcoin transactions?* URL: <https://datalight.me/blog/researches/lightning-network-study-will-this-technology-become-the-new-standart-for-bitcoin-transactions/>.
- [8] Facebook. *React*. URL: <https://github.com/facebook/react>.
- [9] Facebook. *SpyJSBlock-NGraph*. URL: <https://github.com/vincenzopalazzo/SpyJSBlock-NGraph>.
- [10] Harry A. Kalodner et al. «BlockSci: Design and applications of a blockchain analysis platform». In: *CoRR* abs/1709.02489 (2017). arXiv: [1709.02489](https://arxiv.org/abs/1709.02489). URL: <http://arxiv.org/abs/1709.02489>.
- [11] Andrei Kashcha. *ngraph*. URL: <https://github.com/anvaka/ngraph>.
- [12] Andrei Kashcha. *ngraph.graph*. URL: <https://github.com/anvaka/ngraph.graph>.
- [13] Andrei Kashcha. *ngraph.louvain*. URL: <https://github.com/anvaka/ngraph.louvain>.
- [14] Andrei Kashcha. *ngraph.pixel*. URL: <https://github.com/anvaka/ngraph.pixel>.
- [15] Andrei Kashcha. *ngraph.tobinary*. URL: <https://github.com/anvaka/ngraph.tobinary>.
- [16] Andrei Kashcha. *VivaGraphJS*. URL: <https://github.com/anvaka/VivaGraphJS>.

BIBLIOGRAFIA

- [17] Madler. *zlib*. URL: <https://github.com/madler/zlib>.
- [18] Nicolas Dorier kinoshitajona Mark Friedenbach BtcDrak. *Relative lock-time using consensus-enforced sequence numbers*. URL: <https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki>.
- [19] Minium. *bitcoin-api-cpp*. URL: <https://github.com/minium/bitcoin-api-cpp>.
- [20] nayuki. *Bitcoin-Cryptography-Library*. URL: <https://github.com/nayuki/Bitcoin-Cryptography-Library>.
- [21] Team NEO4J. *NEO4J*. URL: <https://neo4j.com>.
- [22] Frédérique E. Oggier, Silivanxay Phetsouvanh e Anwitaman Datta. «BiVA: Bitcoin Network Visualization & Analysis». In: *2018 IEEE International Conference on Data Mining Workshops, ICDM Workshops, Singapore, Singapore, November 17-20, 2018*. A cura di Hanghang Tong et al. IEEE, 2018, pp. 1469–1474. ISBN: 978-1-5386-9288-2. DOI: [10.1109/ICDMW.2018.00210](https://doi.org/10.1109/ICDMW.2018.00210). URL: <https://doi.org/10.1109/ICDMW.2018.00210>.
- [23] Vincenzo Palazzo. *decompiler-bitcoin-script*. URL: <https://github.com/vincenzopalazzo/decompiler-bitcoin-script>.
- [24] Vincenzo Palazzo. *ngraph.fromprecompute*. URL: <https://github.com/vincenzopalazzo/ngraph.fromprecompute>.
- [25] Vinenzo Palazzo. *SpyJSBlock*. URL: <https://github.com/vincenzopalazzo/SpyJSBlocks>.
- [26] Ajay Shrestha e Julita Vassileva. «Bitcoin Blockchain Transactions Visualization». In: nov. 2018, pp. 1–6. DOI: [10.1109/ICCBB.2018.8756455](https://doi.org/10.1109/ICCBB.2018.8756455).
- [27] Ethereum Team. *A Next-Generation Smart Contract and Decentralized Application Platform*. URL: <https://github.com/ethereum/wiki/wiki/White-Paper>.
- [28] Pixi Developer Team. *pixi.js*. URL: <https://github.com/pixijs/pixi.js>.
- [29] Tencent. *rapidjson*. URL: <https://github.com/Tencent/rapidjson>.
- [30] Tenthirtyone. *Blocktools*. URL: <https://github.com/tenthirtyone/blocktools>.
- [31] Hanghang Tong et al., cur. *2018 IEEE International Conference on Data Mining Workshops, ICDM Workshops, Singapore, Singapore, November 17-20, 2018*. IEEE, 2018. ISBN: 978-1-5386-9288-2. URL: <https://ieeexplore.ieee.org/xpl/conhome/8626049/proceeding>.
- [32] Greg Walker. *Pay To Multisig*. URL: <https://learnmeabitcoin.com/glossary/p2ms>.