Una Breve, Dettagliata e Incompleta Introduzione a Bitcoin

Giancarlo Giuffra Moncayo

13 febbraio 2018

Indice

1 Il Protocollo Bitcoin

Vorrei inizialmente fare una distinzione importante. Bitcoin[1] è un protocollo, quindi delle regole per costruire, interpretare ed scambiarsi dei messaggi, che nel caso di Bitcoin rappresentano delle transazioni. Il protocollo[2] ha anche la caratteristica di essere peer-to-peer, cioè tutti i nodi che decidono di partecipare al protocollo hanno gli stessi privilegi, tutti i nodi sono

[1] Satoshi Nakamoto, il creatore di Bitcoin, pubblicò la prima specifica del protocollo in questo paper [1] del 2008.

[2] Per entrare nei dettagli dell'attuale protocollo si può partire da questa pagina [2] della bitcoin wiki.

uguali. D'altra parte, Bitcoin è anche l'oggetto di questi messaggi, cioè si chiama Bitcoin alla crittovaluta o asset digitale che viene scambiato nelle transazioni. Anche se risulta facile capire dal contesto a quale Bitcoin uno si riferisce penso che sia utile fare attenzione alla differenza.

Come abbiamo detto Bitcoin è un protocollo, una specifica, e di conseguenza ha diverse implementazioni. Queste implementazioni si chiamano bitcoin client[3], ed è il software che i nodi che formano parte della rete di Bitcoin hanno in esecuzione nei loro dispositivi, e.g. PC, GPU, ASIC. Il primo bitcoin client fu pubblicato nel 2009 da Satoshi Nakamoto, questo client open source si è evoluto nel tempo e attualmente viene identificato con il nome di Bitcoin Core[4]. Il client viene mantenuto da una communità di svilupattori che viene denominata con lo stesso nome del client.

1.1 Panoramica del funzionamento

Prima di entrare nei dettagli del protocollo vorrei dare una panoramica del funzionamento ed approfittare per introdurre dei concetti di base. Partiamo dal fatto che Bitcoin nasce come protocollo per dare una soluzione peer-to-peer al problema del double spending, dove l'enfasi è in peer-to-peer. Questo problema consiste nell'impedire che un nodo del sistema possa spendere più di una volta la stessa moneta elettronica. Attualmente, quando effettuiamo dei pagamenti elettronici, il compito di controllare che non succedano casi di double spending viene delegato al circuito di pagamento utilizzato, che in modo centralizzato svolge tale compito.Tentativi di sistemi di pagamento elettronico[5] precedenti a Bitcoin risolvevano sempre in modo centralizzato questo problema. In effetti, la decentralizzazione di Bitcoin è una delle sue caratteristiche più innovative e importanti.

Bitcoin risolve il problema del double spending utilizzando l'algoritmo di Proof of Work. L'obbiettivo di questo algoritmo è quello di ordinare temporalmente le transazioni. In questo modo se ci sono due transazioni che cercano di spendere gli stessi Bitcoin la rete considererà valida solo la transazione che in accordo con l'algoritmo di Proof of Work è avvenuta per prima. L'innovazione in Bitcoin è che il consenso per quanto riguarda l'ordine temporale delle transazioni viene raggiunto in modo decentralizzato, ciascun nodo arriva alle stesse conclusioni in modo autonomo. Di fatto ciascun nodo ricostruisce l'intera catena di blocchi nel proprio dispositivo.

[3]Si può trovare una lista dei maggiori bitcoin client qui [3].

[4]La pagina github di Bitcoin Core si può trovare qui [4].

[5]Trovate un resoconto dei predecessori di Bitcoin nella prefazione del libro Bitcoin and Cryptocurrency Technologies [5].

Per entrare nei particolari di questo processo penso che siano necessarie due cose: introdurre il ruolo della figura del Miner e definire più nel dettaglio l'implementazione dell'algoritmo di Proof of Work. Iniziamo con la figura del Miner, e partiamo dal concetto di validazione delle transazioni. Come abbiamo detto prima Bitcoin è un protocollo per costruire, interpretare ed scambiarsi transazioni. Ad ogni istante quindi ci sono nodi che inviano ai suoi vicini le transazioni che hanno costruito secondo il formato del protocollo. Quando una di queste transazioni arriva a un nodo, il bitcoin client processa la transazione e determina se questa è valida o meno, e.g. controlla se i Bitcoin di questa transazione sono stati già spesi e che chi sta spendendo tali Bitcoin abbia effettivamente il diritto di farlo[6]. Se la transazione è valida il nodo propaga la stessa ai suoi vicini, altrimenti la transazione viene rifiutata e semplicemente non viene propagata alla rete. In questo modo tutti i nodi vengono a conoscenza delle transazioni valide che gli altri nodi creano.

A questo punto entra in scena il ruolo del Miner, che raggruppa delle transazioni valide in una struttura dati particolare[7] e aggiunge alcuni altri dati. L'insieme di questi dati più la struttura che contiene le transazioni formano quello che viene chiamato un blocco, uno dei tanti che fanno parte della blockchain. Un dettaglio importante è che tra i dati che il Miner aggiunge al blocco c'è l'hash del blocco precedente. Questo è il meccanismo che permette di legare temporalmente i blocchi formando così una catena che rappresenta il consenso della rete riguardo l'ordine delle transazioni. La creazione di questi blocchi però è soggetta a certe regole. Abbiamo in effetti tralasciato un dettaglio importante, la validazione del blocco.

Man mano che i blocchi vengono creati questi vengono propagati alla rete e come avete intuito vengono sottoposti ad una validazione da parte da ciascun nodo che li riceve, propagando solo quelli validi. In effetti una parte di tale validazione[8] è legata all'algoritmo di Proof of Work e riguarda l'hash del blocco. Un blocco è considerato valido solo se il suo hash è minore[9] di un certo valore chiamato *target*. Per soddisfare questa condizione il Miner ha a disposizione, tra i dati che aggiunge al blocco, una componente che

[6]Entreremo più avanti nel dettaglio di questa validazione dopo aver studiato la struttura delle transazioni.

[7]La struttura per raggruppare le transazioni si chiama Merkle Tree [6], con un singolo hash può identificare un vasto gruppo di transazioni.

[8]Per avere una descrizione completa della validazione di un blocco si può vedere la sezione corrispondente nella seguente pagina [7].

[9]Siamo abituati a rappresentare un hash in base esadecimale, ma essendo un array di byte può facilmente rappresentarsi in base decimale ed ereditare così le operazioni di confronto che conosciamo.

può scegliere a suo piacimento. Questa componente viene chiamata *nonce*. La proprietà di non invertibilità della funzione di hash SHA-256 costringe al Miner a semplicemente provare diversi *nonce* per trovare un hash che soddisfi la condizione di validità. Non esiste effettivamente una strategia che garantisca una maggiore probabilità di trovare un hash valido[10]. L'unica altra variabile che il Miner può controllare oltre al *nonce* è la potenza di calcolo che decide di dedicare e che aumenta in modo proporzionale la sua probabilità di trovare un blocco valido. In questo modo il Miner inviando un blocco valido alla rete sta effettivamente dando prova di aver fatto del lavoro per mantenere sicura la rete, cioè è a tutti gli effetti una Proof of Work.

Rimane in sospeso il dettaglio di come viene determinato il valore di *target* che definisce la condizione di validità dell'hash di un blocco. Il protocollo Bitcoin utilizza questo *target* per mantenere attorno ai 10 minuti la frequenza di creazione dei blocchi. In effetti all'aumentare della capacità computazionale che i Miner dedicano alla rete, aumenta la probabilità che qualcuno trovi un hash valido e di conseguenza anche la frequenza di creazione dei blocchi. Vale ovviamente anche il viceversa. La scelta dei 10 minuti è stata fatta per motivi di stabilità e latenza[11]. Per adeguarsi quindi alla capacità computazionale della rete, il *target* viene aggiornato ogni 2016 blocchi. Ogni nodo della rete calcola l'intervallo temporale che è stato necessario per la creazione degli ultimi 2016 blocchi e lo compara con un intervallo di due settimane, cioè il tempo che sarebbe trascorso se tutti i blocchi fossero stati creati esattamente ogni 10 minuti. Dopodiché calcola la differenza percentuale tra queste due quantità e aggiorna l'attuale valore di *target* in base a tale percentuale[12].

Abbiamo visto finora come i nodi Miner dedicano capacità computazionale alla rete per mantenere ordinate temporalmente le transazioni in una catena di blocchi e come tutti gli altri nodi, inclusi quindi anche eventuali nodi Miner, validano questo lavoro computazionale espresso in ciascun blocco. Abbiamo anche visto come la rete Bitcoin si autoregola per mantenere la capacità di processare transazioni stabile attorno ai 10 minuti per blocco. Vorrei adesso soffermarmi su due aspetti: chi sono effettivamente questi nodi Miner e come si risolve l'eventualità di ricevere due blocchi diversi ma validi che puntano allo stesso blocco precedente.

[10] Si veda la sezione 3 del paper di hashcash [8].

[11] Si veda la relativa FAQ [9] della bitcoin wiki per approfondire il tema della scelta dei 10 minuti.

[12] Si può trovare uno storico della difficoltà(inversamente proporzionale al *target*) qui [10].

Occupiamoci quindi della prima questione. La risposta si trova nella natura peer-to-peer di Bitcoin. Qualunque nodo può fare il Miner se ha delle risorse computazionali da dedicare alla rete. Ovviamente le risorse computazionali hanno un costo e questo è stato considerato nel disegno del protocollo. In effetti, il Miner ha diritto ad includere nel blocco una transazione addizionale[13]. Il beneficiario di questa transazione è scelto dal Miner e molto probabilmente sarà lui stesso. Invece l'ammontare della transazione è definito dal protocollo. Quindi il ruolo del Miner non è solo quello di mantenere sicura la rete ma anche quello di generare o minare Bitcoin. In questo modo il Miner ha un incentivo per dedicare risorse alla sicurezza della rete. Oltre a questa transazione, il Miner ha anche diritto a prendersi le commissioni[14] espresse in ciascuna transazione inclusa nel blocco che ha generato. Come abbiamo detto la quantità di nuovi Bitcoin che il Miner può mettere in circolazione dipende dal protocollo. Inizialmente questa quantità era 50 Bitcoin ma viene dimezzata ogni 210 000 blocchi, che corrispondono circa a 4 anni. Questo fa di Bitcoin una moneta non inflazionaria, in effetti il meccanismo di dimezzamento garantisce che non potranno mai esistere più di 21 millioni di Bitcoin.

La seconda questione riguarda il problema di definire la catena di blocchi attiva. Risulta possibile che due Miner in contemporanea trovino blocchi validi. Questi due blocchi saranno propagati nella rete e di conseguenza i nodi avranno due catene valide che differiscono unicamente nell'ultimo blocco. Questa ambiguità temporanea viene risolta quando il blocco successivo viene trovato. Questo nuovo blocco sarà in effetti collegato solo a uno dei due blocchi e il bitcoin client sceglierà, come da protocollo, la catena che abbia la difficoltà complessiva più alta consentendo alla rete di raggiungere così di nuovo il consenso. Il valore di difficoltà viene calcolato in base al *target*. Quindi normalmente la catena con più difficoltà complessiva sarà quella più lunga. Solo in concomitanza con l'aggiornamento del valore di *target* è probabile che questo non sia il caso.

Questa panoramica del funzionamento di Bitcoin ci è servita per capire come si incastrano i diversi pezzi del protocollo. Abbiamo però parlato in modo generico del concetto di transazione e non abbiamo descritto come vengono effettivamente trasferiti i Bitcoin. Il passo successivo è capire meglio la struttura di una transazione.

[13] Questa transazione viene chiamata coinbase [11].

[14] Le commissioni sono definite autonomamente dal nodo che ha generato la transazione. Vedremo come vengono specificate queste commissioni dopo aver studiato la struttura di una transazione.

1.2 Struttura di una transazione

Una transazione contiene le informazioni riguardo chi invia e chi riceve i Bitcoin. La particolarità di una transazione nella rete di Bitcoin è che in una di esse ci possono essere più di un mittente e più di un destinatario. Le informazioni che riguardano i mittenti sono specificate nel campo `vin`, che è effettivamente un array di oggetti json[15], mentre i destinatari vengono specificati nel campo `vout`, anch'esso un array di oggetti. I componenti di questi array vengono chiamati input e output[16] rispettivamente. Prima di spiegare in dettaglio come vengono codificate le informazioni riguardo mittenti e destinatari penso che sia necessario ricordare come viene utilizzata la crittografia asimmetrica in Bitcoin.

Ricordiamo che la crittografia asimmetrica, chiamata anche crittografia a chiave pubblica, si basa sul concetto di coppia di chiavi. Quindi ad ogni attore, nel nostro caso un possessore di Bitcoin, viene associata una coppia di chiavi. Una di queste si chiama chiave privata mentre l'altra, chiave pubblica. Come dal nome si può evincere, quella privata deve essere mantenuta al sicuro dall'attore e quella pubblica viene comunicata agli altri attori. Vorrei approfittare per fare alcune precisazioni. La generazione della coppia di chiavi deve essere fatta in modo autonomo da chi vuole ricevere dei Bitcoin[17]. Inoltre l'algoritmo che viene utilizzato non è particolare di Bitcoin ma bensì uno standard della crittografia ellittica[18]. Infine vorrei dare enfasi al fatto che la conoscenza della chiave privata implica la possibilità di spendere i Bitcoin associati alla corrispondente chiave pubblica. Quindi bisogna stare molto attenti nella gestione delle chiavi.

Proseguiamo quindi con l'analisi della struttura di una transazione. Occupiamoci per prima del campo `vout`. Come abbiamo detto, in ciascun oggetto di questo array viene codificato un destinatario. Questo oggetto contiene due informazioni importanti: la quantità di Bitcoin che il destinatario riceve e le caratteristiche della prova crittografica che dovrà provvedere chi vorrà in futuro spendere questi Bitcoin. La quantità corrisponde semplicemente al campo value dell'oggetto json. Invece le informazioni riguardanti la prova sono più complicate e vengono codificate nel campo scriptPubKey utilizzando il linguaggio di scripting di Bitcoin. Approfondiremo più avanti

[15] Un formato molto comune per rappresentare le transazioni Bitcoin è il formato json [12].

[16] Gli output che non sono stati ancora spesi vengono chiamati UTXO, cioè Unspent Transaction Output.

[17] Ci sono dei software che aiutano nella creazione e gestione delle chiavi tra altre cose. Questi software vengono chiamati wallet, una lista si trova qui [13]. Bitcoin Core, ad esempio, offre anche le funzionalità di un wallet.

[18] La curva utilizzata in Bitcoin si chiama secp256k1 [14].

questo linguaggio. Per il momento ci basta sapere che la chiave pubblica del destinatario viene inclusa in questo campo. Occupiamoci adesso dei mittenti. Come abbiamo detto, nel campo `vin` vengono codificati i mittenti, cioè chi sta spendendo i loro Bitcoin. Ciascun oggetto di questo array contiene due importanti informazioni: quali Bitcoin si stanno spendendo e la prova crittografica[19] che il possessore di questi Bitcoin autorizza la spesa. Anche questa prova viene codificata nel linguaggio di scripting di Bitcoin. L'informazione riguardo quali Bitcoin si stanno spendendo viene codificata in due campi, `txid` e `vout`[20]. Per capire la natura di questi due campi risulta utile notare che un mittente è a sua volta il destinatario dell'output di una precedente transazione. Il campo txid si riferisce quindi all'hash di una tale precedente transazione mentre il campo `vout` si riferisce all'indice dell'array `vout` che corrisponde all'output che si sta spendendo. L'altra informazione importante, cioè la prova che il mittente è autorizzato a spendere i Bitcoin viene codificata nel campo scriptSig[21]. Si osserva come un input spende per intero un output di una transazione precedente, quindi normalmente quando si vuole spendere un output parzialmente si includono nella transazione due output: uno che corrisponde alla spesa che si sta effettuando e l'altro che corrisponde al resto. Ovviamente quest'ultimo deve contenere una chiave pubblica controllata da chi sta spendendo l'input, altrimenti non potrebbe essere considerato un resto.

Avendo parlato degli input e degli output possiamo introdurre il concetto di commissioni di una transazione. Gli input che appartengono a una transazione definiscono il totale di Bitcoin che può essere speso. Questo è semplicemente la somma dei campi value degli output che ciascun input spende. Gli output della transazione invece definiscono il totale di Bitcoin spesi, anche in questo caso la somma dei loro campi value. Una delle condizioni necessarie affinchè la transazione sia considerata valida è che il totale speso sia minore o uguale al totale che si può spendere. Nel caso questo sia minore, la differenza costituisce le commissioni della transazione. Questo è un meccanismo di incentivazione per i Miner, più alte le commissioni più è probabile che la transazione venga inclusa in un blocco. Inoltre si osserva che questo diventerà l'unico meccanismo di incentivazione una volta che il value dell'output della transazione coinbase arrivi a zero.

Adesso che abbiamo introdotto i campi più importanti di una transazione occupiamoci più in dettaglio della relazione tra i campi scriptPubKey

[19] La validità di tutte queste prove, una per ciascun input, è una delle condizioni necessarie affinchè la transazione corrispondente sia valida.

[20] Sì, anche questo campo si chiama vout. Ma osservate che questo è un campo degli input che formano l'array vin, mentre l'altro vout è un campo dell'oggetto transazione.

[21] Si veda il Listing 1.1 per un esempio di transazione.

e scriptSig. Come abbiamo visto, il campo scriptSig contiene la prova crittografica che permette spendere l'output di una transazione precedente. E se vi ricordate, ogni output contiene nel campo scriptPubKey le caratteristiche di una prova crittografica che deve essere presentata da chi poi vorrà spendere l'output. Quindi ogni input di una transazione contiene nel campo scriptSig la prova crittografica le cui caratteristiche vengono specificate nel campo scriptPubKey dell'output che si sta spendendo. Questo meccanismo collega l'input di una transazione all'output di una precedente. Notate come in questo modo, partendo da un output, si può ricostruire una specie di albero genealogico di tale output andando a vedere gli output associati agli input che lo hanno generato e ripetendo ricorsivamente questo processo per ciascuno di questi[22].

Penso che sia opportuno spiegare più in dettaglio come funzionano sia la prova crittografica contenuta nel campo scriptSig sia le caratteristiche specificate nel campo scriptPubKey. Come abbiamo detto, in questo ultimo campo viene inclusa la chiave pubblica[23] del destinatario dell'output. Lo script viene costruito in modo che la prova crittografica richiesta sia una firma digitale associata alla chiave pubblica specificata. Nel campo scriptSig viene quindi inclusa una tale firma digitale. Si ricorda che una firma digitale valida viene generata solo se si conosce la chiave privata associata alla chiave pubblica e dopo aver scelto il messaggio da firmare. Nel caso dell'input di una transazione Bitcoin il messaggio da firmare corrisponde all'hash di una versione semplificata[24] della stessa transazione.

Adesso che sappiamo che il linguaggio di scripting di Bitcoin viene utilizzato nei campi scriptSig e scriptPubKey delle transazioni possiamo dedicarci a capire meglio la sua sintassi, il suo utilizzo e le sue capacità.

1.3 Programmabilità delle transazioni

Siamo abituati a pensare a Bitcoin come un semplice sistema di pagamenti, ma ci sfugge il fatto che dovrebbe essere più visto come una piattaforma. In effetti, specificare il destinatario di un pagamento è solo una delle possibilità che ci offre il linguaggio di scripting di Bitcoin.

[22] E' di questa relazione tra transazioni della quale si parla quando ci si riferisce alla storia di un Bitcoin.

[23] Non è del tutto preciso. Vedremo i dettagli nella sezione riguardanti il linguaggio di scripting di Bitcoin.

[24] Le possibili semplificazioni verrano trattate nella sezione riguardante il linguaggio di scripting di Bitcoin.

```
{
  "txid": "acf4038165ba8481be4fd729d10730339152dcd8c11386a7c7f477657ba
      c50a3",
  "hash": "acf4038165ba8481be4fd729d10730339152dcd8c11386a7c7f477657ba
      c50a3",
  "version": 2,
  "size": 191,
  "vsize": 191,
  "locktime": 0,
  "vin": [
    {
      "txid": "c717a095c5f330027ac69a9ebff3f2263eef3a1782df5c37c0b3341
          b646bb480",
      "vout": 0,
      "scriptSig": {
        "asm": "304402205ec8e82a688e61c668eb27473df402c0d91d35bf4ccd26
            12a56cfbc8d92c7fec0220157f4b864936bfd01fc9fcdc886ab262575
            684139ce473f790bf6afc96719b93[ALL]
            03a54c780ec0b3ac4535c074e630ca635152cd790738bb47c68e82a70
            9ee357032",
        "hex": "47304402205ec8e82a688e61c668eb27473df402c0d91d35bf4ccd
            2612a56cfbc8d92c7fec0220157f4b864936bfd01fc9fcdc886ab2625
            75684139ce473f790bf6afc96719b93012103a54c780ec0b3ac4535c0
            74e630ca635152cd790738bb47c68e82a709ee357032"
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.19900000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160
            38fb769fb56737de1f9b86e66100105d15e788ff OP_EQUALVERIFY
            OP_CHECKSIG",
        "hex": "76a91438fb769fb56737de1f9b86e66100105d15e788ff88ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "mkiFNn5jf6UwGBX6nJDHFEvgTheB4aFiaT"
        ]
      }
    }
  ]
```


Listing 1.1: Esempio di transazione P2PKH con un singolo input e un singolo output. Si specifica che è una transazione in testnet. Potete vedere la transazione live nella blockchain qui.

Cominciamo elencando le caratteristiche del linguaggio che, senza molta fantasia, si chiama Script[25]:

1. E' un linguaggio di programmazione basato su stack, come il più conosciuto Forth[26],

2. Le istruzioni vengono processate da sinistra verso destra,

3. E' volutamente non Turing completo, in particolare non possiede comandi per specificare dei cicli.

La forma più semplice per capire come funziona il linguaggio è attraverso un esempio. Prima vediamo come si scrive una semplice equazione in Script e poi procediamo ad analizzare come un bitcoin client interpreterebbe i comandi. L'equazione che abbiamo scelto è la seguente:

$$17 + 25 = 42.$$

In Script tale equazione viene scritta come segue:

```
17 25 OP_ADD 42 OP_EQUAL
```

In Figura 1 vediamo come un bitcoin client interpreterebbe questo script. Il client inizia con uno stack vuoto e comincia a leggere lo script da sinistra verso destra. Quando trova delle costanti le inserisce nello stack, come nel caso dei due numeri 17 e 25. Poi incontra il comando `OP_ADD`. Questo comando richiede di fare due estrazioni, sommare i due valori estratti e inserire la somma nello stack. Quindi viene estratto prima 25, successivamente 17, e la somma 42 viene inserita. Il client poi incontra la costante 42 e la inserisce nello stack. Infine arriva al comando `OP_EQUAL`. Questo comando richiede di fare due estrazioni, verificare se i due valori estratti sono uguali e inserire il risultato nello stack. Quindi vengono estratti i due numeri 42 e alla fine viene inserito il numero 1 nello stack per indicare che l'uguaglianza è vera.

Adesso che capiamo meglio come funziona Script possiamo parlare più in dettaglio della relazione tra i campi scriptSig e scriptPubKey delle transazioni Bitcoin. Ricordiamo che scriptSig contiene la prova crittografica che soddisfa le caratteristiche specificate nel campo scriptPubKey dell'output che si sta spendendo. In effetti, quando viene validato l'input di una

[25]Si veda la seguente pagina [15] della bitcoin wiki per una descrizione più approfondita e per una lista di tutti i comandi del linguaggio.

[26]Si veda la seguente pagina [16] wiki.

```
17 25 OP_ADD 42 OP_EQUAL     [
25 OP_ADD 42 OP_EQUAL        [ 17
OP_ADD 42 OP_EQUAL           [ 17 25
42 OP_EQUAL                  [ 42
OP_EQUAL                     [ 42 42
.                            [ 1
```

Figura 1: Esempio di processamento di uno script. A destra si trova rappresentato lo stato dello stack prima di processare il relativo pezzo di script.

transazione, il bitcoin client costruisce uno script prendendo il contenuto del campo scriptSig e appendendo il contenuto del corrispondente campo scriptPubKey. Lo script così ottenuto viene poi processato e l'input è considerato valido se si verificano i seguenti due criteri:

Regole di Validazione 1.

1. Durante l'esecuzione la transazione non è stata marcata come invalida[27] ,

2. Al termine dell'esecuzione l'elemento in cima allo stack non è zero, cioè contiene un valore che rappresenta il booleano true.[28]

Vediamo un esempio concreto della costruzione di un tale script e la relativa validazione dal client. Prendiamo l'unico output della transazione nel Listing 1.1 e spendiamolo. L'input che lo spende viene mostrato nel Listing 1.2. Riportiamo di seguito il suo scriptSig:

```
3045...8d85[ALL] 0370...a689
```

Abbiamo omesso per semplicità alcune cifre, comunque la struttura si può ben apprezzare. Lo scriptSig è diviso in due parti separate da uno spazio. La prima parte contiene due informazioni: la firma digitale che spende l'output corrispondente e, tra parentesi quadre, un codice che indica la

[27] In Script ci sono dei comandi che possono marcare una transazione come invalida, e.g. `OP_VERIFY`. Si veda la relativa descrizione nella seguente pagina [15] della bitcoin wiki.

[28] Si osserva che per soddisfare questa condizione non è necessario che lo stack contenga un unico valore, e.g. la condizione sarebbe soddisfatta con uno stack pieno che in cima contiene il numero 42.

semplificazione applicata alla transazione prima di firmarla. Questo codice viene chiamato Hashtype Value e ne parleremo più avanti in questa sezione. La seconda parte contiene la chiave pubblica alla quale corrisponde la firma digitale. Vi starete domandando perchè è necessario includere la chiave pubblica nello scriptSig se abbiamo precedentemente detto che questa chiave veniva inclusa nello scriptPubKey per specificare il destinatario dell'output. Quest'ultima informazione non è esatta e adesso vediamo il perchè analizzando lo scriptPubKey:

```
OP_DUP OP_HASH160 38fb...88ff OP_EQUALVERIFY OP_CHECKSIG
```

```
{
      "txid": "acf4038165ba8481be4fd729d10730339152dcd8c11386a7c7f4776
          57bac50a3",
      "vout": 0,
      "scriptSig": {
        "asm": "3045022100e3cb87c2e824da45e16ae2f5476e94c8680e8a869d3a
            08550c97830ba64b36b502202bd0595d991e843c8cbcb1772eaba8ebf
            4d592c931fe9b1baf786e3cd9848d85[ALL]
            03709b41f316b5561c577a401ac2b66394a494c3b609826d46ee8d2c0
            01999a689",
        "hex": "483045022100e3cb87c2e824da45e16ae2f5476e94c8680e8a869d
            3a08550c97830ba64b36b502202bd0595d991e843c8cbcb1772eaba8e
            bf4d592c931fe9b1baf786e3cd9848d85012103709b41f316b5561c57
            7a401ac2b66394a494c3b609826d46ee8d2c001999a689"
      },
      "sequence": 4294967295
}
```

Listing 1.2: Input che spende l'unico output della transazione nel Listing 1.1. Potete vedere la transazione che lo contiene qui.

Abbiamo sempre omesso alcune cifre per facilitare la visualizzazione. Come si può vedere, sono tutti comandi ad eccezione della terza componente. Questa è in effetti un hash, e come avete intuito è l'hash della chiave pubblica che corrisponde al destinatario dell'output. Per questo motivo è necessario includere nello scriptSig la chiave pubblica. Questo tipo di output viene chiamato Pay-To-Public-Key-Hash(P2PKH) e riprenderemo questo e altri casi d'uso nella sezione successiva. Adesso mettiamo insieme lo scriptSig e lo scriptPubKey e seguiamo il processamento dello script da parte di un bitcoin client. In Figura 2 vediamo rappresentata questa elaborazione.

Inizialmente abbiamo lo script completo, cioè scriptSig seguito da scriptPubKey, e lo stack vuoto. Poi sia la firma digitale sia la chiave pubblica vengono inserite nello stack. Il comando `OP_DUP` duplica l'elemento in cima allo stack, in questo caso la chiave pubblica. Di seguito il comando `OP_HASH160`[29] estrae l'elemento in cima, calcola il suo hash e lo inserisce nello stack. L'algoritmo di hash usato è la composizione di due funzioni di hash, prima viene applicato l'algoritmo SHA-256 e successivamente l'algoritmo RIPEMD-160. L'elemento successivo è l'hash incluso nello scriptPubKey e viene semplicemente inserito nello stack. Di seguito arriva il processamento del comando `OP_EQUALVERIFY`. Essendo la composizione di due comandi, per chiarezza, abbiamo deciso di dividerlo nei due comandi che lo compongono. Quindi prima si processa il comando `OP_EQUAL` che estrae i due elementi in cima, verifica l'uguaglianza e inserisce il risultato nello stack. In questo caso i due hash sono uguali e quindi inserisce 1 nello stack. Poi segue il comando `OP_VERIFY` che estrae l'elemento in cima. Se questo è zero marca la transazione come invalida, altrimenti lascia che il processamento continui. Funziona effettivamente come un controllo delle asserzioni. Infine arriva il comando `OP_CHECKSIG` che estrae la chiave pubblica e la firma digitale e verifica che la firma sia valida inserendo poi il risultato nello stack. Nel nostro caso è valida e viene inserita la costante 1. Avendo finito l'esecuzione dello script e avendo in cima allo stack il valore 1, l'input viene considerato valido. Il comando `OP_CHECKSIG` è particolarmente complesso[30] perchè oltre ai valori che ha estratto dallo stack utilizza due altre informazioni: tutta la transazione alla quale l'input che si sta validando appartiene e l'indice di questo input. In effetti, il Hashtype Value, in questo caso ALL, dà delle indicazioni al comando per quanto riguarda la semplificazione applicata alla transazione prima di generare la firma. Vedremo tra poco i diversi tipi di Hashtype Value che offre il protocollo.

Abbiamo visto come con Script si possono specificare nello scriptPubKey di un output le condizioni da soddisfare per spenderlo. Nel caso analizzato l'output è stato speso presentando la chiave pubblica corrispondente all'hash specificato e una firma digitale valida. In sostanza con questo tipo di script si sono inviati dei Bitcoin a un destinatario scelto senza compromettere la sua identità. In effetti la chiave pubblica è stata rivelata solo quando i Bitcoin sono stati spesi e non prima. Ci sono molti altri comandi in Script che permettono di specificare diverse condizioni da soddisfare. Vedremo più avanti alcuni di questi casi d'uso.

[29] Si veda la sezione Crypto della seguente pagina [15] della bitcoin wiki per la definizione del comando.

[30] Si veda la relativa pagina [17] della bitcoin wiki per i dettagli. In particolare la rappresentazione grafica dei passaggi.

```
30..85[ALL] 03..89 OP_DUP OP_HASH160 38..ff OP_EQUALVERIFY OP_CHECKSIG
03..89 OP_DUP OP_HASH160 38..ff OP_EQUALVERIFY OP_CHECKSIG
OP_DUP OP_HASH160 38..ff OP_EQUALVERIFY OP_CHECKSIG
OP_HASH160 38..ff OP_EQUALVERIFY OP_CHECKSIG
38..ff OP_EQUALVERIFY OP_CHECKSIG
OP_EQUALVERIFY OP_CHECKSIG
OP_EQUAL OP_VERIFY OP_CHECKSIG
OP_VERIFY OP_CHECKSIG
OP_CHECKSIG
.
```

```
[
[ 30..85[ALL]
[ 30..85[ALL] 03..89
[ 30..85[ALL] 03..89 03..89
[ 30..85[ALL] 03..89 38..ff
[ 30..85[ALL] 03..89 38..ff 38..ff
[ 30..85[ALL] 03..89 38..ff 38..ff
[ 30..85[ALL] 03..89 1
[ 30..85[ALL] 03..89
[ 1
```

Figura 2: Esempio di processamento di uno script P2PKH. In basso si trova rappresentato lo stato dello stack prima di processare il relativo pezzo di script.

Riprendiamo il concetto di Hashtype Value che accompagna una firma digitale. Come abbiamo detto, questa componente serve a specificare la semplificazione applicata alla transazione prima di generare la firma. Si ricorda anche che quello che viene firmato è l'hash della transazione semplificata. In particolare si usa l'algoritmo SHA-256 due volte per calcolare questo hash. Prima di introdurre i diversi Hashtype Values bisogna spiegare che c'è un primo passaggio di semplificazione effettuato in ogni caso. Questo consiste principalmente[31] in svuotare tutti gli scriptSig e metterli uguali al byte 0x00[32]. Quindi rimangono comunque i riferimenti a quali output ciascun input della transazione sta spendendo. Di seguito presentiamo i diversi tipi di Hashtype Value offerti dal protocollo con la sua relativa spiegazione:

1. ALL: Non viene effettuata nessun'altra semplificazione. Quindi corrisponde a firmare tutti gli input e tutti gli output.

[31] Alcuni dettagli non vengono trattati per favorire la comprensione. Si rimanda alla relativa pagina [17] della bitcoin wiki.

[32] 0x00 è la rappresentazione esadecimale del singolo byte zero. Si ricorda che in esadecimale due cifre corrispondono a un byte.

2. NONE: L'array `vout` della transazione viene svuotato. Quindi corrisponde a firmare tutti gli input ma nessun output.

3. SINGLE: L'array `vout` viene ridimensionato in modo che l'ultimo output abbia lo stesso indice dell'input che si sta validando. A tutti gli altri output gli si assegna uno scriptPubKey vuoto e un value pari a -1. Quindi corrisponde a firmare tutti gli input e un solo output.

4. ANYONECANPAY: L'array `vin` viene lasciato con un unico elemento, quello corrispondente all'input che si sta validando. Quindi corrisponde a firmare un solo input e tutti gli output.

Occupiamoci infine di un ultimo dettaglio che riguarda i campi scriptSig e scriptPubKey. Se avete notato nell'esempio di output P2PKH che abbiamo analizzato abbiamo solo preso in considerazione il contenuto del campo asm di scriptSig e scriptPubKey. Questo campo contiene la rappresentazione simbolica di Script, cioè quella che utilizza i comandi, anche detti opcodes, come **`OP_CHECKSIG`**. Un'altra rappresentazione di Script è quella esadecimale che troviamo nei corrispondenti campi hex di scriptSig e scriptPubKey. Per passare dalla rappresentazione simbolica a quella esadecimale si seguono basicamente due regole:

1. ogni comando ha un codice esadecimale associato,[33]

2. le costanti vengono precedute da zero, uno, due, tre o cinque byte che indicano la quantità di bytes che sono utilizzati per rappresentare la costante.[34]

Analizziamo i campi hex dell'esempio di output P2PKH visto in precedenza. Iniziamo con il campo scriptSig dell'input nel Listing 1.2. Riportiamo di seguito il relativo campo hex:

```
48 304..d85 01 21 037..689
```

Abbiamo introdotto degli spazi e omesso alcune cifre per facilitare la comprensione. Il primo byte, 0x48, indica che segue una costante da 72[35] byte. In questo caso sono 71 byte che corrispondono alla firma digitale più

[33] Si vedano le diverse tabelle nella sezione Opcodes della seguente pagina [15] della bitcoin wiki.

[34] Si veda la tabella Constants della seguente pagina [15] della bitcoin wiki per i dettagli.

[35] Si ricorda che 0x48 è espresso in base 16.

un byte che corrisponde all'Hashtype Value ALL indicato col byte 0x01[36]. Poi vediamo il byte 0x21 che indica che seguono 33 byte, nel nostro caso corrispondono alla chiave pubblica. Per quanto riguarda lo scriptPubKey della transazione nel Listing 1.1 abbiamo il seguente campo hex:

```
76 a9 14 38fb769fb56737de1f9b86e66100105d15e788ff 88 ac
```

Il byte 0x76 corrsiponde al comando `OP_DUP` mentre 0xa9 corrisponde al comando `OP_HASH160`. Invece il byte 0x14 indica che seguono 20 byte che, nel nostro caso, corrispondono all'hash della chiave pubblica. Gli ultimi due byte, 0x88 e 0xac, corrispondono ai comandi `OP_EQUALVERIFY` e `OP_CHECKSIG` rispettivamente.

In questa sezione abbiamo introdotto il linguaggio di scripting di Bitcoin e visto come viene utilizzato per specificare delle condizioni da soddisfare per poter spendere l'output di una transazione. Nella sezione successiva vedremo altri casi d'uso che Script può supportare introducendo dei nuovi comandi.

1.4 Use cases: P2PKH, `OP_RETURN`, P2SH, Multisignatures

P2PKH

Abbiamo già visto un esempio di questo caso d'uso, lo scripPubKey che invia i Bitcoin nel Listing 1.1 e lo scriptSig che li spende nel Listing 1.2. Questo è il caso più comune[37] nella rete di Bitcoin, corrisponde in effetti a semplicemente inviare i Bitcoin a un'altra chiave pubblica. Questo tipo di output è il successore di Pay-To-Public-Key, identificato con P2PK e che attualmente viene considerato deprecato. Si ricorda che nel caso di P2PKH veniva incluso l'hash della chiave pubblica nello scriptPubKey, invece nel caso di P2PK viene inclusa direttamente la chiave pubblica. A titolo di esempio vediamo quali sarebbero stati gli script utilizzati in una transazione P2PK analoga alla transazione P2PKH dei Listing 1.1 e 1.2. Riportiamo di seguito lo scriptPubKey:

```
037...689 OP_CHECKSIG
```

[36]Si veda la tabella Hashtype Values della seguente pagina [17] per la rappresentazione esadecimale degli altri valori.

[37]Una distribuzione per tipologia degli UTXO si può trovare qui [18] in Tabella 2.

Semplicemente include la chiave pubblica e il comando per verificare la firma digitale. Quindi lo scriptSig che spende tale output include unicamente una firma valida:

```
304..d85
```

Notate che una transazione P2PK è molto più semplice ma espone la chiave pubblica. E' preferibile utilizzare una P2PKH per motivi di sicurezza. Soprattutto perché, nell'eventualità di computer quantistici facilmente disponibili, questi potrebbero essere usati per risalire, a partire dalle chiavi pubbliche esposte, alle chiavi segrete. Potendo così spendere i rispettivi Bitcoin senza problemi. Esponendo solo l'hash, un computer quantistico non rappresenta una così grave minaccia[38].

`OP_RETURN`

Il comando `OP_RETURN` marca l'output di una transazione come invalido. La presenza di questo comando nello scriptPubKey di un output rende automaticamente non spendibili i Bitcoin indicati nel campo value. Di conseguenza questo meccanismo può essere utilizzato per dimostrare di aver bruciato i Bitcoin indicati nell'output. La capacità di poter dimostrare di aver bruciato dei Bitcoin si può usare in due modi principalmente:

1. Servire come base per implementare l'algoritmo di consenso Proof of Burn[39],
2. Definire la distribuzione iniziale di un'altra crittovaluta consegnando delle unità di questa nuova crittovaluta in base ai Bitcoin che sono stati bruciati[40].

Il comando `OP_RETURN` può essere anche utilizzato per inserire dei dati nella blockchain di Bitcoin. Il meccanismo per farlo è semplice, bisogna includere nella transazione un output che abbia un value pari a zero e uno scriptPubKey che contenga il comando `OP_RETURN` seguito dalla rappresentazione esadecimale dei dati che si vogliono includere. Il Listing 1.3 mostra una tale transazione. La transazione ha un input e due output. Un output

[38] Si veda questa discussione [19] riguardo la sicurezza delle funzioni di hash in relazione ai computer quantistici.

[39] Si veda la seguente pagina [20] della bitcoin wiki riguardo la Proof of Burn. Ad esempio, il progetto Slimcoin [21] implementa questo algoritmo di consenso.

[40] Così è stata implementata la distribuzione iniziale di XCP, la crittovaluta del progetto Counterparty [22].

è utilizzato per includere dei dati nella blockchain mentre il secondo è un output P2PKH che restituisce i Bitcoin dell'input, ad eccezione delle commissioni, a una chiave pubblica controllata dallo stesso utente. Riportiamo lo scriptPubKey dell'output che ci interessa:

```
OP_RETURN 737...e2e
```

come si può osservare include semplicemente il comando seguito dalla rappresentazione esadecimale dei dati inclusi.

La possibilità di includere dei dati arbitrari nella blockchain ha due casi d'uso importanti:

1. Servire come base per servizi di timestamping, cioè per provare che dei dati esistevano prima di un certo istante temporale[41],

2. Servire come base per includere dei messaggi che appartengono ad altri protocolli[42] che sfruttano così l'ordinamento che offre la blockchain di Bitcoin.

P2SH

Un'altra tipologia di output è quella indicata con l'acronimo P2SH, Pay-To-Script-Hash. Questo tipo di output offre molta flessibilità perchè permette di spendere un output secondo le regole codificate in uno script arbitrario. Come si può intuire dal nome, e analogamente all'output di tipo P2PKH, nel campo scriptPubKey di questo tipo di output viene incluso l'hash di uno script. Riportiamo di seguito lo schema generale di un tale scriptPubKey:

```
OP_HASH160 hashOfScript OP_EQUAL
```

Ricordando le regole per la validazione di un input (si vedano quelle in 1) potremmo essere tentati a pensare che per poter spendere l'output basterebbe includere nello scriptSig soltanto la rappresentazione esadecimale di uno script il cui hash corrisponda al valore `hashOfScript`. Invece, per

[41] Nel caso del timestamping si include l'hash del dato dopo il comando `OP_RETURN`. Il progetto OpenTimestamps [23] usa questo meccanismo ottimizzando il numero di timestamps per transazione.

[42] Blockstack [24] usa `OP_RETURN` in questo modo per creare un sistema DNS decentralizzato.

```
{
  "txid": "90779f381724efafef9c69f88248527b591d211741d8be8ae2ffa4de7beb989e",
  "hash": "90779f381724efafef9c69f88248527b591d211741d8be8ae2ffa4de7beb989e",
  "version": 2,
  "size": 370,
  "vsize": 370,
  "locktime": 0,
  "vin": [
    {
      "txid": "0bb736419fbf85b3225449831f697210ff498a0c605032c4f914db2d789f2829",
      "vout": 0,
      "scriptSig": {
        "asm": "3045022100bcc3bb805fe4dc664ce979d7eeffc3ad84ee132de420d62afb3a33d4c5f9e
              e500220612d7dc8a4a5a60125888e25a92beb531cf0046cfc441c09420b29648360eb17[A
              LL]
              03c2b4a6821b29b9e0a90915f3d2041fe6abefdb2810b74c99196171c3f72c19e6",
        "hex": "483045022100bcc3bb805fe4dc664ce979d7eeffc3ad84ee132de420d62afb3a33d4c5f
              9ee500220612d7dc8a4a5a60125888e25a92beb531cf0046cfc441c09420b29648360eb17
              012103c2b4a6821b29b9e0a90915f3d2041fe6abefdb2810b74c99196171c3f72c19e6"
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.00000000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_RETURN 737065726f20636865207175657374 6f206c6962726f20766920736961207
              574696c652c20686f207363726974746f206c61206d6961207061727465207065 6e73616e
              646f20616c206c6962726f20636865206176726569207 66f6c75746f2074726f766172652
              07175616e646f20686f20696e697a6961746f20616420696e7465726573 7361726d692061
              6c6c65207465636e6f6c6f67696520626c6f636b636861696e2e",
        "hex": "6a4ca6737065726f20636865207175657374 6f206c6962726f207669207369612075746
              96c652c20686f207363726974746f206c61206d6961207061727465207065 6e73616e646f
              20616c206c6962726f20636865206176726569207 66f6c75746f2074726f766172652071 7
              5616e646f20686f20696e697a6961746f20616420696e7465726573 7361726d6920616c6c
              65207465636e6f6c6f67696520626c6f636b636861696e2e",
        "type": "nulldata"
      }
    },
    {
      "value": 0.19700000,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 bdef79534666c895ca92e80f03df621958ba89c2
              OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a914bdef79534666c895ca92e80f03df621958ba89c288ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "mxqEtFTdn4jd6EsSTfJQ1Fpw5UqTAX7Vix"
        ]
      }
    }
  ]
}
```


Listing 1.3: Esempio di transazione con un output utilizzato per includere dei dati arbitrari. Potete vedere la transazione live in testnet qui.

questo tipo di output, le regole[43] che il bitcoin client segue per la validazione dell'input sono leggermente diverse a quelle usate per un output P2PKH. Per poter spendere un output P2SH bisogna includere nello scriptSig due informazioni:

A.1 La rappresentazione esadecimale dello script il cui hash corrisponde all'hash incluso nello scriptPubKey, come si poteva intuire, e

A.2 Uno script che concatenato alla rappresentazione simbolica dello script al punto precedente risulti in una validazione riuscita secondo le regole descritte in 1

Lo scriptSig è quindi composto dallo script del punto A.2 seguito dalla rappresentazione esadecimale del punto A.1. Come nel caso della validazione di un output P2PKH, a questo scriptSig viene concatenato lo scriptPubKey e lo script così ottenuto è processato dal bitcoin client. Una condizione necessaria per una validazione riuscita è che lo script del punto A.2 sia composto di sole operazioni di inserimento. Se non è così la validazione fallisce. Questa condizione implica che inizialmente il client eseguirà tutte le operazioni di inserimento incluse in questo script e poi inserirà la rappresentazione esadecimale del punto A.1 nello stack prima di arrivare al comando `OP_HASH160` dello scriptPubKey. Da questo punto si distinguono due fasi in questa validazione:

1. La prima fase consiste nel processamento dei 3 elementi dello scriptPubKey, e corrisponde a verificare che l'hash della rappresentazione esadecimale corrisponda all'hash incluso nello scriptPubKey. La validazione fallisce immediatamente se i due hash non coincidono, cioè se l'elemento in cima allo stack, il risultato del comando `OP_EQUAL`, è zero. Il risultato del comando viene estratto dallo stack.

2. Nella seconda fase rimangano nello stack solo i dati inseriti precedentemente dal processamento dello script al punto A.2. Si procede poi a trasformare la rappresentazione esadecimale nella corrispondente rappresentazione simbolica e si processano i comandi lì contenuti. La validazione segue poi le regole descritte in 1.

Vediamo un esempio concreto di validazione di un output P2SH. Prendiamo spunto dal semplice script visto in precedenza all'inizio della sottosezione 1.3:

[43] Si veda la BIP 16(Bitcoin Improvement Proposal) [25] che dettaglia l'output P2SH per approfondimenti.

```
17 25 OP_ADD 42 OP_EQUAL
```

Il pezzo di questo script che dovremo trasformare in esadecimale, per poi calcolare l'hash, è:

```
OP_ADD 42 OP_EQUAL
```

La sua rappresentazione esadecimale, con qualche spazio inserito per facilitare la lettura, è:

```
93 01 2a 87
```

Il byte 0x93 corrisponde al comando **OP_ADD**. Il byte 0x01 indica che il prossimo byte deve essere incluso nello stack. 0x2a è la rappresentazione esadecimale di 42 e infine 0x87 corrisponde al comando **OP_EQUAL**. L'hash di questa rappresentazione esadecimale è:

```
e46bc9d4dc5d12c1be1cf6c3c01af14d5285d924
```

A questo punto abbiamo tutto per definire sia lo scriptPubKey dell'output P2SH sia lo scriptSig dell'input che lo spende. Iniziamo con lo scriptPubKey:

```
OP_HASH160 e46...924 OP_EQUAL
```

Invece lo scriptSig è:

```
17 25 93012a87
```

In Figura 3 vediamo rappresentato il processamento fino al termine della prima fase della validazione. Si inizia con la concatenazione dello scriptSig seguito dallo scriptPubKey e con lo stack vuoto. Si verifica che lo scriptSig contenga solo operazioni di inserimento, essendo questo il nostro caso si procede con la validazione. Vengono inseriti nello stack il valore 17 seguito dal valore 25 e infine la rappresentazione esadecimale. Si arriva al comando **OP_HASH160** che estrae l'elemento in cima, calcola il suo hash e inserisce poi il risultato nello stack. Poi si incontra l'hash incluso nello scriptPubKey che viene inserito anch'esso nello stack. Segue il comando **OP_EQUAL** che estrae i due elementi in cima, li confronta e inserisce il risultato nello stack. Nel nostro caso i due hash coincidono e viene inserito nello stack il valore 1. Infine si estrae il valore in cima e si controlla che non sia zero, essendo

```
17 25 93012a87 OP_HASH160 e46...924 OP_EQUAL
25 93012a87 OP_HASH160 e46...924 OP_EQUAL
93012a87 OP_HASH160 e46...924 OP_EQUAL
OP_HASH160 e46...924 OP_EQUAL
e46...924 OP_EQUAL
OP_EQUAL
.
.
```

```
[
[ 17
[ 17 25
[ 17 25 93012a87
[ 17 25 e46...924
[ 17 25 e46...924 e46...924
[ 17 25 1
[ 17 25
```

Figura 3: Esempio della prima fase di validazione di uno script P2SH. In basso si trova rappresentato lo stato dello stack prima di processare il relativo pezzo di script.

uno si procede con la seconda fase della validazione. Si osserva come siano rimasti nello stack i valori 17 e 25, cioè quelli inseriti per via dell'esecuzione di quello che nel nostro caso è lo script del punto A.2.

In Figura 4 vediamo rappresentata la seconda fase della validazione. Si inizia con lo stack lasciato alla fine della prima fase. Si trasforma la rappresentazione esadecimale in rappresentazione simbolica e si inizia a processarla. Il primo elemento è il comando **OP_ADD** che estrae i due elementi in cima, li somma e inserisce il risultato 42 nello stack. Il secondo elemento è la costante 42 che viene inserita nello stack. In seguito arriva il comando **OP_EQUAL** che estrae i due elementi in cima, li confronta ed essendo questi uguali inserisce il valore 1 nello stack. Avendo finito le istruzioni e rimanendo il valore 1 in cima la validazione si conclude con esito positivo.

Adesso che abbiamo visto come si effettua la validazione dell'input che spende un output P2SH vorrei soffermarmi su due aspetti. Il primo è il fatto che lo script che viene validato nella seconda fase del processo può contenere qualsiasi logica che possa essere codificata in Script. L'unico limite imposto allo script è che la sua rappresentazione esadecimale, venendo inserita nello stack, non superi i 520 bytes definiti come contromisura per gli attacchi DoS[44]. Il secondo aspetto riguarda la differenza tra la sempli-

[44]Si veda la sezione *Denial of Service(DoS) attacks* della seguente pagina [26] della

```
.
OP_ADD 42 OP_EQUAL
42 OP_EQUAL
OP_EQUAL
.
```

```
[ 17 25
[ 17 25
[ 42
[ 42 42
[ 1
```

Figura 4: Esempio della seconda fase di validazione di uno script P2SH. In basso si trova rappresentato lo stato dello stack prima di processare il relativo pezzo di script.

cità dello scriptPubKey e la complessità dello scriptSig. Le transazioni che inviano dei Bitcoin a output di questo tipo sono grosse, in termini di byte, circa quanto quelle che usano output P2PKH. Invece quelle che spendono gli output sono considerevolmente più grosse rispetto alle transazioni che spendono output P2PKH. Questo implica che il destinatario di un output P2SH, per spenderlo, deva sostenere dei costi più alti, in termini di commissioni, rispetto a un destinatario di output P2PKH. Si può dire che il costo più elevato è attribuibile alla flessibilità che offre l'output P2SH e viene giustamente sostenuto da chi effettivamente ne beneficia, cioè il destinatario e non il mittente.

Multisignatures

Come visto precedentemente, il tipo di output P2SH offre molte possibilità nel definire le condizioni che devono essere soddisfate per spenderlo. Una, molto importante, è il caso delle Multisignatures. La condizione per spendere un output Multisignature m-of-n è che si includano nell'input le firme digitali di m utenti presi da un insieme di n chiavi pubbliche, dove ovviamente $m \leq n$. Il comando che permette la costruzione di un output Multisignature è il comando **OP_CHECKMULTISIG**. Vediamo come funziona questo comando con un esempio di Multisignature 2-of-3:

```
x sign1 sign3 2 pubKey1 pubKey2 pubKey3 3 OP_CHECKMULTISIG
```

bitcoin wiki.

Abbiamo utilizzato dei placeholder per facilitare la comprensione. Prima descriviamo i diversi elementi dello script. Iniziamo con le firme digitali che sono state indicate con `sign1` e `sign3`. Queste due firme corrispondo alle chiavi pubbliche `pubKey1` e `pubKey3`, rispettivamente. Per la chiave `pubKey2` non è stata inclusa una firma, adesso vedremo il perchè. Rimangono, oltre al comando, i valori `x`, `2` e `3`. Il valore `3`, che separa il comando dalle 3 chiavi pubbliche, indica, appunto, il numero di chiavi pubbliche dell'insieme. Il valore `2`, che separa le chiavi dalle firme, indica il numero di firme richieste per avere un risultato positivo da parte del comando **`OP_CHECKMULTISIG`**. Infine il valore `x` è un valore arbitrario che deve essere incluso perchè, per motivi di un bug[45], il comando estrae un valore extra dallo stack. Se notate tutti i valori che precedono il comando sono dati e quindi vengono semplicemente inseriti nello stack. Poi il comando **`OP_CHECKMULTISIG`** li estrae, li processa e inserisce il risultato nello stack. Vedremo un esempio concreto tra poco. Adesso vorrei spiegare come il comando opera su questi dati perchè questo implica un dettaglio importante: tutte le firme devono corrispondere a una chiave pubblica e inoltre l'ordine nel quale vengono inserite deve rispecchiare l'ordine delle chiavi. Nel nostro caso, per esempio, se le firme fossero invertite il comando non darebbe un risultato positivo. Riportiamo di seguito l'algoritmo che esegue il comando su questi dati:

1. Si prende la prima firma e la si confronta con ciascuna chiave pubblica fino a trovare una corrispondenza.

2. Si prende la seconda firma e la si confronta con ciascuna chiave del sottoinsieme che si trova a destra della chiave che corrispondeva alla firma precedente.

3. Il processo si ripete fino a che tutte le firme abbiano trovato una corrispondenza oppure fino a che non ci siano sufficienti chiavi rimanenti per produrre un risultato positivo.

Questo modo di processare i dati richiede, oltre a includere il numero necessario di firme valide, che tutte le firme abbiano una corrispondenza e che siano nell'ordine dettato dalle chiavi pubbliche. Altrimenti, ad esempio, se si includesse una firma non valida, al punto 2 dell'algoritmo, l'insieme rimanente di chiavi con le quali confrontare le firme sarebbe vuoto. Lo stesso succederebbe se si includesse per prima una firma corrispondente all'ultima chiave pubblica.

[45] Si veda la tabella Crypto della seguente pagina [15] della bitcoin wiki.

Adesso vediamo un esempio concreto, nel Listing 1.4 vediamo un output P2SH che contiene l'hash di uno script Multisignature 2-of-3. Lo script Multisignature è il seguente:

```
2
02120cba3feed9d13de38525fccda0616e48490f977ed918fe929b72070959dbe6
03f45c2c89173154db67e6e99188e0a14fcdeb54a8d741c6bdb50e413cf1fc5568
03e4ea596fe3d4b93f9e271b4fc7a922d4515dad398c110b267ace054913523d72
3
OP_CHECKMULTISIG
```


```
{
      "value": 0.19600000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_HASH160 9e7de39f1f7b8c63c7311d60586a6dd3096cbbfe
            OP_EQUAL",
        "hex": "a9149e7de39f1f7b8c63c7311d60586a6dd3096cbbfe87",
        "reqSigs": 1,
        "type": "scripthash",
        "addresses": [
          "2N7hFZXNQPNXjwFztFoTJs5j6GpgWN9pLy7"
        ]
      }
}
```


Listing 1.4: Esempio di output P2SH che contiene l'hash di uno script Multisignature 2-of-3. Potete vedere la transazione che lo contiene live in testnet qui.

Il lettore può verificare che l'hash nello scriptPubKey dell'output nel Listing 1.4 corrisponde a questo script. Si ricorda solo che prima si deve passare alla rappresentazione esadecimale dello script. Nel Listing 1.5 vediamo invece l'input che spende il nostro output Multisignature. Riportiamo di seguito lo scriptSig:

```
0 304..38d[ALL] 304..e92[ALL] 522..3ae
```

Si osserva come i primi 3 elementi corrispondono al valore arbitrario, in questo caso zero, e alle due firme digitali, prima e seconda chiave pubblica

```
{
      "txid": "de467a2f57ad8bb4e60bc72241aace4734df3cdbcb11a39a2c7e4551140220af",
      "vout": 0,
      "scriptSig": {
        "asm": "0 30440220548f0fd9d1998b1d1fdfa507b4097fde5eaaab04391061f1955613133b7f0bd40220316765e81ed077d12ecb22b9ff9684cd1e3252319ccf2310cb706d89fdcdb38d[ALL] 30440220375061ef536bef3c36e7f78c24e93ca410c96703dc6d19d22dd2739db85cd827022007b0640451df0100776e238ab398ac37705211a99900f76a3596a3c4ff9dbe92[ALL] 522102120cba3feed9d13de38525fccda0616e48490f977ed918fe929b72070959dbe62103f45c2c89173154db67e6e99188e0a14fcdeb54a8d741c6bdb50e413cf1fc55682103e4ea596fe3d4b93f9e271b4fc7a922d4515dad398c110b267ace054913523d7253ae",
        "hex": "00473044022054 8f0fd9d1998b1d1fdfa507b4097fde5eaaab04391061f1955613133b7f0bd40220316765e81ed077d12ecb22b9ff9684cd1e3252319ccf2310cb706d89fdcdb38d01473044022037 5061ef536bef3c36e7f78c24e93ca410c96703dc6d19d22dd2739db85cd8270220007b0640451df0100776e238ab398ac37705211a99900f76a3596a3c4ff9dbe92014c69522102120cba3feed9d13de38525fccda0616e48490f977ed918fe929b72070959dbe62103f45c2c89173154db67e6e99188e0a14fcdeb54a8d741c6bdb50e413cf1fc55682103e4ea596fe3d4b93f9e271b4fc7a922d4515dad398c110b267ace054913523d7253ae"
      },
      "sequence": 4294967295
}
```


Listing 1.5: Input che spende l'output Multisignature 2-of-3 nel Listing 1.4. Le firme corrispondono alla prima e seconda chiave pubblica, rispettivamente. Potete vedere la transazione che lo contiene live in testnet qui.

```
.
2 021..be6 03f..568 03e..d72 3 OP_CHECKMULTISIG
021..be6 03f..568 03e..d72 3 OP_CHECKMULTISIG
03f..568 03e..d72 3 OP_CHECKMULTISIG
03e..d72 3 OP_CHECKMULTISIG
3 OP_CHECKMULTISIG
OP_CHECKMULTISIG
.
```

```
[ 0 304..38d[ALL] 304..e92[ALL]
[ 0 304..38d[ALL] 304..e92[ALL]
[ 0 304..38d[ALL] 304..e92[ALL] 2
[ 0 304..38d[ALL] 304..e92[ALL] 2 021..be6
[ 0 304..38d[ALL] 304..e92[ALL] 2 021..be6 03f..568
[ 0 304..38d[ALL] 304..e92[ALL] 2 021..be6 03f..568 03e..d72
[ 0 304..38d[ALL] 304..e92[ALL] 2 021..be6 03f..568 03e..d72 3
[ 1
```

Figura 5: Esempio della seconda fase di validazione di uno script P2SH Multisignature 2-of-3. In basso si trova rappresentato lo stato dello stack prima di processare il relativo pezzo di script.

rispettivamente, che verificano le condizioni espresse nel nostro script Multisignature. L'ultimo elemento è la rappresentazione esadecimale di questo script. Infine riportiamo in Figura 5 la seconda fase di validazione dell'input.

Bibliografia

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[2] Bitcoin Wiki. Protocol documentation. https://en.bitcoin.it/wiki/Protocol_documentation.

[3] Bitcoin Wiki. Clients. https://en.bitcoin.it/wiki/Clients.

[4] Bitcoin Core. Github. https://github.com/bitcoin/bitcoin.

[5] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.

[6] Wikipedia. Merkle tree. https://en.wikipedia.org/wiki/Merkle_tree.

[7] Bitcoin Wiki. Protocol rules. https://en.bitcoin.it/wiki/Protocol_rules.

[8] Adam Back. Hashcash-a denial of service counter-measure. 2002.

[9] Bitcoin Wiki. Faq. https://en.bitcoin.it/wiki/Help:FAQ.

[10] BitcoinWisdom. Difficulty. https://bitcoinwisdom.com/bitcoin/difficulty.

[11] Bitcoin Wiki. Coinbase. https://en.bitcoin.it/wiki/Coinbase.

[12] Wikipedia. Json. https://en.wikipedia.org/wiki/JSON.

[13] Bitcoin.org. Bitcoin wallets. https://bitcoin.org/en/choose-your-wallet.

[14] Bitcoin Wiki. Secp256k1. https://en.bitcoin.it/wiki/Secp256k1.

[15] Bitcoin Wiki. Script. https://en.bitcoin.it/wiki/Script.

[16] Wikipedia. Forth. https://en.wikipedia.org/wiki/Forth_(programming_language).

[17] Bitcoin Wiki. Opchecksig. https://en.bitcoin.it/wiki/OP_CHECKSIG.

[18] Sergi Delgado-Segura, Cristina Pérez-Sola, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomartı. Analysis of the bitcoin utxo set.

[19] Cryptography Stack Exchange. Are cryptographic hash functions quantum secure? https://crypto.stackexchange.com/questions/44386/are-cryptographic-hash-functions-quantum-secure.

[20] Bitcoin Wiki. Proof of burn. https://en.bitcoin.it/wiki/Proof_of_burn.

[21] SlimCoin. Github. https://github.com/slimcoin-project.

[22] Counterparty. https://counterparty.io/.

[23] OpenTimestamps. https://opentimestamps.org/.

[24] Blockstack. https://blockstack.org/.

[25] Bitcoin Core. Bip 16. https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki.

[26] Bitcoin Wiki. Weaknesses. https://en.bitcoin.it/wiki/Weaknesses.