Da Zero a Monero: Seconda Edizione

una guida tecnica alla valuta digitale privata per eccellenza; per utenti principianti ed esperti

Data di pubblicazione 24 luglio 2025 (v2.0.0)

koe[1], Kurt M. Alonso[2], Sarang Noether[3]

Traduzione a cura di

everoddandeven[4]

Licenza: Da Zero a Monero: Seconda Edizione è rilasciato al pubblico dominio.

[1] ukoe@protonmail.com
[2] kurt@oktav.se
[3] sarang.noether@protonmail.com
[4] everoddandeven@protonmail.com

Abstract

Crittografia: potrebbe sembrare che solo matematici e informatici abbiano accesso a questo argomento oscuro, esoterico, potente ed elegante. In realtà, diverse tipologie di crittografia sono abbastanza semplici da poter essere comprese da chiunque ne apprenda i concetti fondamentali.

È risaputo che la crittografia viene utilizzata per proteggere le comunicazioni, che si tratti di messaggi cifrati o interazioni digitali private. Un'altra applicazione si trova nelle cosiddette criptovalute. Queste valute digitali utilizzano la crittografia per assegnare e trasferire la proprietà dei fondi. Per garantire che nessuna unità di valuta possa essere duplicata o creata arbitrariamente, le criptovalute si basano generalmente su blockchain, ovvero registri pubblici e distribuiti contenenti tutte le transazioni di valuta, verificabili da terze parti [95].

A prima vista potrebbe sembrare che le transazioni debbano essere inviate e archiviate in modo trasparente per poter essere pubblicamente verificabili. In realtà, è possibile nascondere sia i partecipanti di una transazione sia gli importi coinvolti, utilizzando strumenti crittografici che permettono comunque la verifica e il raggiungimento del consenso da parte di osservatori esterni [134]. Questo approccio è esemplificato nella criptovaluta Monero.

L'obiettivo di questo documento è spiegare a chiunque conosca l'algebra di base e alcuni semplici concetti informatici, come la 'rappresentazione binaria di un numero', non solo come funziona Monero in modo approfondito e completo, ma anche quanto possa essere utile e affascinante la crittografia.

Per i lettori più esperti: Monero è una criptovaluta basata su una blockchain standard, a grafo aciclico diretto (DAG) unidimensionale [95] in cui le transazioni utilizzano la crittografia a curve ellittiche (in particolare la curva Ed25519 [35]). Gli input delle transazioni sono firmati con firme di gruppo anonime multilivello, spontanee, collegabili, in stile Schnorr (MLSAG) [107], mentre gli importi in uscita (comunicati ai destinatari tramite ECDH [49]) sono nascosti mediante impegni di Pedersen [85] e dimostrati appartenere ad un intervallo valido grazie ai Bulletproof [40]. Gran parte della prima sezione di questo documento è dedicata a spiegare questi concetti.

Indice

1 Introduzione **1**

1.1 Obiettivi . . . 2

1.2 Lettori . . . 3

1.3 Le Origini di Monero . . . 4

1.4 Struttura . . . 4

1.4.1 Parte 1: Elementi Essenziali . . . 4

1.4.2 Parte 2: Estensioni . . . 5

1.4.3 Contenuti Aggiuntivi . . . 5

1.5 Avviso . . . 5

1.6 Storia di Da Zero a Monero . . . 6

1.7 Ringraziamenti . . . 6

I Elementi Essenziali **8**

2 Concetti Basilari **9**

2.1 Notazione . . . 9

2.2 Aritmetica Modulare . . . 10

2.2.1 Somma e Moltiplicazione Modulari . . . 11
2.2.2 Potenza Modulo . . . 12
2.2.3 Inverso Moltiplicativo Modulo . . . 13
2.2.4 Equazioni Modulari . . . 13
2.3 Crittografia a Curva Ellittica . . . 14
2.3.1 Le Curve Ellittiche . . . 14
2.3.2 Crittografia a Chiave Pubblica con Curve Ellittiche . . . 17
2.3.3 Scambio Chiavi Diffie-Hellman con Curve Ellittiche . . . 17
2.3.4 Firme di Schnorr e la Trasformazione di Fiat-Shamir . . . 18
2.3.5 Firma dei Messaggi . . . 20
2.4 Curva Ed25519 . . . 21
2.4.1 Rappresentazione Binaria . . . 22
2.4.2 Compressione dei Punti . . . 22
2.4.3 Algoritmo di Firma EdDSA . . . 23
2.5 Operatore Binario XOR . . . 25

3 Firme Avanzate in Stile Schnorr **26**
3.1 Dimostrazione della Conoscenza di un Logaritmo Discreto su Basi Multiple . . . 26
3.2 Più Chiavi Private in una Dimostrazione . . . 27
3.3 Firme di Gruppo Anonime Spontanee (Spontaneous Anonymous Group, SAG) . . . 28
3.4 Le Firme bLSAG . . . 31
3.5 Firme Multilivello MLSAG . . . 34
3.6 Firme Concise CLSAG . . . 35

4 Indirizzi Monero **38**
4.1 Chiavi Utente . . . 39
4.2 Indirizzi Monouso (One-time) . . . 39
4.2.1 Transazioni Multi-Output . . . 41
4.3 Sottoindirizzi . . . 42
4.3.1 Trasferimento Fondi ad un Sottoindirizzo . . . 42
4.4 Indirizzi Integrati . . . 44
4.5 Indirizzi Multifirma . . . 45

5 **Offuscamento degli Importi** 46
5.1 Commitment . . . 46
5.2 Commitment di Pedersen . . . 47
5.3 Commitment degli Importi . . . 48
5.4 Introduzione a RingCT . . . 49
5.5 Prove di Intervallo . . . 50
6 **Transazioni Confidenziali ad Anello (RingCT)** 52
6.1 Tipologie di Transazione . . . 53
6.2 Transazioni Confidenziali ad Anello `RCTTypeBulletproof2` . . . 53
6.2.1 Commitment sugli Importi e Commissioni di Transazione . . . 53
6.2.2 Firma . . . 54
6.2.3 Il Problema della Doppia Spesa . . . 56
6.2.4 Requisiti di Memoria . . . 56
6.3 Riepilogo Concettuale: Transazioni Monero . . . 58
6.3.1 Requisiti di Memoria . . . 60
7 **La Blockchain di Monero** 61
7.1 Valuta Digitale . . . 62
7.1.1 Versione Distribuita degli Eventi . . . 62
7.1.2 Blockchain Semplice . . . 63
7.2 Difficoltà . . . 64
7.2.1 Estrazione di un Blocco . . . 64
7.2.2 Velocità di Estrazione . . . 65
7.2.3 Consenso: Difficoltà Cumulativa Maggiore . . . 65
7.2.4 Mining in Monero . . . 66
7.3 Offerta di Moneta . . . 67
7.3.1 Ricompensa per Blocco . . . 68
7.3.2 Peso Dinamico dei Blocchi . . . 69

7.3.3 Penalità su Ricompense di Blocco . . . 71
7.3.4 Commissione Minima Dinamica . . . 72
7.3.5 Emissione Residua . . . 75
7.3.6 Transazione Miner: `RCTTypeNull` . . . 75
7.4 Struttura della Blockchain . . . 76
7.4.1 ID della Transazione . . . 77
7.4.2 Albero di Merkle . . . 77
7.4.3 Blocchi . . . 79

II Estensioni **80**

8 Prove di Conoscenza Relative alle Transazioni Monero **81**
8.1 Prove di Transazione Monero . . . 81
8.1.1 Prove di Transazione Monero Multi-Base . . . 82
8.1.2 Prova di Creazione di Input di Transazione (`SpendProofV1`) . . . 82
8.1.3 Prova di Creazione di Output di Transazione (`OutProofV2`) . . . 84
8.1.4 Dimostrare la Proprietà di un Output (`InProofV2`) . . . 86
8.1.5 Dimostrare che un Output Posseduto Non è Stato Speso in una Transazione (UnspentProof) . . . 87
8.1.6 Provare che un Indirizzo ha un Saldo Non Speso Minimo (`ReserveProofV2`) 89
8.2 Framework di Audit Monero . . . 90
8.2.1 Dimostrare la Corrispondenza tra Indirizzo e Sottoindirizzo (SubaddressProof) 91
8.2.2 Il Framework di Audit . . . 91

9 Multifirme in Monero **93**
9.1 Comunicazione tra Co-Firmatari . . . 94
9.2 Aggregazione delle Chiavi per gli Indirizzi . . . 95
9.2.1 Approccio Ingenuo . . . 95
9.2.2 Svantaggi dell'Approccio Ingenuo . . . 95
9.2.3 Aggregazione Robusta delle Chiavi . . . 97

9.3 Firme in Stile Schnorr con Soglia 98
9.4 Firme Confidenziali ad Anello MLSTAG per Monero 100
9.4.1 `RCTTypeBulletproof2` con Multisig N-su-N 101
9.4.2 Comunicazione Semplificata 103
9.5 Calcolo delle Immagini Chiave 105
9.6 Soglie più Piccole 106
9.6.1 Aggregazione delle Chiavi 1-su-N 107
9.6.2 Aggregazione delle Chiavi (N-1)-su-N 107
9.6.3 Aggregazione delle Chiavi M-su-N 110
9.7 Famiglie di Chiavi 112
9.7.1 Alberi Genealogici 112
9.7.2 Annidamento delle Chiavi Multisig 113
9.7.3 Implicazioni per Monero 115

10 Marketplace Monero con Deposito a Garanzia (Escrow) 116
10.1 Caratteristiche Essenziali 117
10.1.1 Flusso di Lavoro per l'Acquisto 118
10.2 Multisig Monero Trasparente 119
10.2.1 Basi dell'Interazione Multisig 119
10.2.2 Esperienza Utente con Escrow 121

11 Transazioni Monero Congiunte (TxTangle) 127
11.1 Costruzione di Transazioni Congiunte 128
11.1.1 Canale di Comunicazione a n Vie 128
11.1.2 Turni di Messaggi per Costruire una Transazione Congiunta 129
11.1.3 Debolezze 131
11.2 TxTangle Ospitato 132
11.2.1 Comunicazione di Base con un Host su I2P e Altre Funzionalità 132
11.2.2 Host come Servizio 134
11.3 Dealer Fidato 135
11.3.1 Procedura Basata su Dealer 135

Bibliography **137**

Appendices **144**

A Struttura delle Transazioni `RCTTypeBulletproof2` **146**

B Contenuto dei Blocchi **153**

C Blocco Genesi **157**

CAPITOLO 1

Introduzione

Nel mondo digitale è spesso banale creare infinite copie di un'informazione, e apportare altrettante modifiche. Perché una valuta possa esistere in forma digitale ed essere ampiamente adottata, i suoi utilizzatori devono assicurarsi che la sua disponibilità sia rigidamente limitata. Chi riceve del quantitativo di denaro deve poter avere fiducia che non si tratti di monete false, né di monete clonate già inviate a qualcun altro. Per ottenere tutto ciò senza richiedere l'intervento di un'entità esterna, come un'autorità centrale, è necessario che sia la disponibilità della valuta sia l'intera cronologia delle transazioni siano pubblicamente verificabili.

Possiamo utilizzare strumenti crittografici per fare in modo che i dati registrati in un database facilmente accessibile — la blockchain — siano virtualmente immutabili e non falsificabili, con una legittimità che non possa essere messa in discussione da nessuno.

Le criptovalute consentono la memorizzazione delle transazioni all'interno della blockchain, che funge da registro pubblico[1] di tutte le operazioni effettuate con quella moneta. La maggior parte delle criptovalute registra le transazioni in chiaro, per facilitarne la verifica da parte della comunità di utenti.

È evidente, però, che una blockchain trasparente contraddice ogni concetto elementare di privacy e fungibilità[2], poiché rende letteralmente pubblica la cronologia completa delle transazioni degli utenti.

[1] In questo contesto, "registro" consiste in un semplice un archivio di tutti gli eventi di generazione e scambio della moneta. In particolare, registra quanto denaro è stato trasferito in ciascun evento e a chi è stato destinato.

[2] "**Fungibile** significa capace di essere scambiato o sostituito con un altro bene equivalente nell'uso o nell'adempimento di un contratto. ... Esempi: ... denaro, ecc."[63] In una blockchain trasparente come quella di Bitcoin, le

Per affrontare questa mancanza di privacy, gli utenti di criptovalute come Bitcoin possono offuscare le transazioni utilizzando indirizzi intermedi temporanei [97]. Tuttavia, con gli strumenti adeguati, è possibile analizzare i flussi e, in larga misura, collegare i veri mittenti ai destinatari [124, 38, 111, 45].

Al contrario, la criptovaluta Monero (si pronuncia Mo-ne-ro) cerca di risolvere il problema della privacy memorizzando sulla blockchain solo indirizzi monouso per la ricezione dei fondi, ed autorizzando l'invio dei fondi in ogni transazione tramite firme ad anello. Grazie a queste funzionalità, non esistono ad oggi tecniche generalmente efficaci per collegare i destinatari o rintracciare l'origine dei fondi.[3]

Inoltre, gli importi delle transazioni sulla blockchain di Monero sono nascosti grazie a costrutti crittografici, rendendo i flussi di valuta opachi e difficili da rilevare.

Il risultato è una criptovaluta con un elevato livello di privacy e fungibilità.

1.1 Obiettivi

Monero è una criptovaluta consolidata con oltre cinque anni di sviluppo [125, 94], e mantiene un livello di adozione in costante crescita [51].[4] Purtroppo, esiste poca documentazione che descriva i meccanismi su cui si basa.[5,6] Ancora peggio, parti essenziali del suo impianto teorico sono state pubblicate in articoli non-peer-reviewed, spesso incompleti o contenenti errori. Per molte componenti fondamentali della teoria dietro Monero, l'unica fonte affidabile è il codice sorgente stesso.[7]

Inoltre, per chi non ha una formazione in matematica, apprendere le basi della crittografia a curve ellittiche, ampiamente utilizzata in Monero, può risultare un percorso confuso e frustrante.[8]

---

monete possedute da Alice possono essere distinte facilmente da quelle possedute da Bob in base alla cronologia delle transazioni associate a ciascuna moneta. Se la cronologia di Alice include transazioni legate ad attori considerati discutibili o addirittura criminali, le sue monete potrebbero essere considerate "macchiate" (tainted) [92], e quindi avere un valore inferiore rispetto a quelle di Bob, anche se l'importo posseduto è lo stesso. Alcune fonti autorevoli affermano che i Bitcoin appena coniati vengono scambiati con un sovrapprezzo rispetto a quelli usati, proprio perché privi di una cronologia [117].

[3] A seconda del comportamento degli utenti, possono esserci casi in cui le transazioni risultano parzialmente analizzabili. Per un esempio concreto, si veda questo articolo: [58].

[4] In termini di capitalizzazione di mercato, Monero è rimasto stabile rispetto ad altre criptovalute. Era 14-esima nel Giugno 2018, e 12-esima il 5 Gennaio 2020; vedi `https://coinmarketcap.com/`.

[5] Uno sforzo di documentazione disponibile su `https://monerodocs.org/`, ha alcunve voci utili, in particolare quelle relative alla Command Line Interface. La CLI è un portafoglio Monero accessibile da console/terminale. Ha più funzionalità rispetto ad altri portafogli Monero, al costo del sacrificio di un interfaccia user-friendly.

[6] Un altro, più generale, impegno nella documentazione è Mastering Monero che può essere trovato qui: [54].

[7] Mr. Seguias ha creato un'ottima serie chiamata Monero Building Blocks [122], che contiene un trattemento approfondito sulle prove di sicurezza crittografica utilizzate per giustificare lo schema di firme di Monero. Come con Da Zero a Monero: Prima Edizione [30], le serie di Seguias sono incentrate sulla versione v7 del protocollo.

[8] Un tentativo iniziale di spiegare i meccanismi di Monero [109] che non ha però spiegato la crittografia a curve ellittiche, ed oltre ad essere incompleto, sono passati circa cinque anni dalla sua pubblicazione.

L'intenzione degli autori del presente documento è di migliorare questa situazione introducendo i concetti fondamentali necessari per comprendere la crittografia a curve ellittiche, passando in rassegna algoritmi e schemi crittografici, e raccogliendo informazioni approfondite sul funzionamento interno di Monero.

Per offrire ai lettori la migliore esperienza possibile, gli autori si sono occupati di fornire una descrizione costruttiva e passo dopo passo della criptovaluta Monero.

In questa seconda edizione del documento, è stata concentrata l'attenzione sulla versione 12 del protocollo Monero[9], corrispondente alla versione 0.15.x.x della suite software Monero. Tutti i meccanismi relativi alle transazioni e alla blockchain descritti in questo documento fanno riferimento a tali versioni.[10,11] Gli schemi di transazione obsoleti non sono stati trattati, nemmeno parzialmente, anche se potrebbero ancora essere supportati per motivi di retrocompatibilità. Lo stesso vale per le funzionalità della blockchain ormai superate. La prima edizione [30] era invece riferita alla versione 7 del protocollo e alla versione 0.12.x.x della suite software.

1.2 Lettori

Abbiamo previsto che molti lettori si avvicineranno a questo manuale senza alcuna o con pochissima conoscenza di matematica discreta, strutture algebriche, crittografia[12], e blockchain. Gli autori hanno cercato di essere sufficientemente completi nella trattazione dei vari argomenti affinché persone comuni, provenienti da qualsiasi ambito, possano apprendere il funzionamento di Monero senza dover ricorrere a ricerche esterne.

La scelta di omettere — o relegare come note a piè di pagina — alcuni tecnicismi matematici è voluta, in quanto questi avrebbero compromesso la chiarezza dell'esposizione. Sono stati evitati anche alcuni dettagli concreti di implementazione non ritenuti essenziali. L'obiettivo è stato quello di presentare l'argomento a metà strada tra la crittografia matematica e la programmazione informatica, puntando alla completezza e alla chiarezza concettuale.[13]

[9] Il 'protocollo' è l'insieme di regole contro cui ogni nuovo blocco viene verificato prima di poter essere aggiunto alla blockchain. Questo insieme di regole include il 'protocollo delle transazioni' (attualmente versione 2, RingCT), che sono regole generali riguardanti come una transazione deve essere costruita. Regole specifiche sulle transazioni possono, e spesso lo fanno, cambiare senza che cambi la versione del protocollo delle transazioni. Solo modifiche su larga scala alla struttura della transazione giustificano un aggiornamento del numero di versione.

[10] L'integrità e l'affidabilità del codice di Monero si basano sull'assunzione che un numero sufficiente di persone lo abbia revisionato, individuando la maggior parte o tutti gli errori significativi. I lettori non dovrebbero dare per scontate le nostre spiegazioni, ma sono invitati a verificare autonomamente che il codice faccia davvero ciò che dovrebbe. Se così non fosse, auspichiamo una divulgazione responsabile (`https://hackerone.com/monero`) per i problemi gravi, oppure una pull request su Github (`https://github.com/monero-project/monero`) per le questioni minori.

[11] Sono in corso ricerche e studi su diversi protocolli potenzialmente adatti alla prossima generazione di transazioni Monero, tra cui Triptych [105], RingCT3.0 [141], Omniring [82] e Lelantus [69].

[12] An extensive textbook on applied cryptography can be found here: [39].

[13] Alcune note a piè di pagina, soprattutto nei capitoli relativi al protocollo, introducono capitoli o sezioni successivi. Ciò è voluto per rendere il concetto più chiaro a una seconda lettura, poiché di solito includono dettagli implementativi specifici utili solo a chi ha una conoscenza approfondita del funzionamento di Monero.

1.3 Le Origini di Monero

La criptovaluta Monero, inizialmente nota come BitMonero, è stata creata nell'aprile del 2014 come implementazione della *Proof of Concept* di criptovaluta CryptoNote [125]. Monero significa denaro in Esperanto, e la sua forma plurale è Moneroj (mo-ne-roi, simile a Moneros ma con la "j" pronunciata come in orange).

CryptoNote è un protocollo ideato da più individui. Un Whitepaper fondamentale che lo descrive è stato pubblicato nell'ottobre 2013 da un utente sotto lo pseudonimo di Nicolas van Saberhagen [134]. Questo sistema offre anonimato per il destinatario attraverso l'uso di indirizzi monouso, e offuscamento del mittente mediante le firme ad anello.

Fin dalla sua nascita ad oggi, Monero ha ulteriormente potenziato i propri aspetti legati alla privacy, implementando anche il mascheramento degli importi, come descritto da Greg Maxwell (e altri) in [86], integrato nelle firme ad anello seguendo le raccomandazioni di Shen Noether [107], e successivamente reso più efficiente con l'adozione dei Bulletproof [40].

1.4 Struttura

Come accennato precedentement, l'obiettivo è offrire una descrizione per un lettura autonoma, spiegando passo dopo passo, la criptovaluta Monero. Il presente documento, Da Zero a Monero, è stato strutturato per raggiungere questo scopo, guidando il lettore attraverso tutte le componenti del funzionamento interno della criptovaluta.

1.4.1 Parte 1: Elementi Essenziali

Nel percorso verso la completezza, è stato deciso di presentare tutti gli elementi base della crittografia necessari per comprendere la complessità di Monero, insieme ai loro fondamenti matematici. Nel Capitolo 2 vengono sviluppati gli aspetti fondamentali della crittografia a curve ellittiche.

Il Capitolo 3 approfondisce lo schema di firma di Schnorr introdotto nel capitolo precedente, e presenta gli algoritmi di firme ad anello utilizzati per garantire la riservatezza delle transazioni.

Il Capitolo 4 spiega come Monero usa gli indirizzi per conferire la proprietà dei fondi, e i diversi tipi di indirizzi esistenti.

Nel Capitolo 5 vengono introdotti i meccanismi crittografici utilizzati per nascondere gli importi delle transazioni.

Una volta illustrate tutte le componenti, nel Capitolo 6, è descritto nel dettaglio lo schema di transazione adottato da Monero.

Infine, nel Capitolo 7, è analizzato il funzionamento della blockchain di Monero. .

1.4.2 Parte 2: Estensioni

Una criptovaluta è più della semplice implementazione del protocollo su cui si basa, e nella sezione Estensioni sono tratti una serie di concetti diversi, molti dei quali non sono ancora stati implementati.[14]

È possibile dimostrare varie informazioni su una transazione agli osservatori esterni, e questi metodi costituiscono il contenuto del Capitolo 8.

Pur non essendo essenziali per il funzionamento di Monero, le multifirme (multisignature) sono molto utili, poiché permettono a più persone di inviare e ricevere denaro in modo collaborativo. Il Capitolo 9 descrive l'attuale approccio di Monero alle multifirme e delinea possibili sviluppi futuri in quest'area.

Di fondamentale importanza è l'applicazione delle multifirme alle interazioni tra venditori e acquirenti nei marketplace online. Il Capitolo 10 presenta un progetto originale di marketplace con escrow basato sulle multisignature di Monero ideato dall'autore di questo documento.

Presentato per la prima volta in questo documento, TxTangle, descritto nel Capitolo 11, è un protocollo decentralizzato per unire le transazioni di più individui in una singola operazione.

1.4.3 Contenuti Aggiuntivi

L'appendice A spiega la struttura di una transazione di esempio dalla blockchain. L'appendice B descrive la struttura dei blocchi, comprese le intestazioni (header) e transazioni di mining, all'interno della blockchain di Monero. Infine, l'appendice C chiude il documento spiegando la struttura del blocco di genesi di Monero. Questi appendici forniscono un collegamento tra gli elementi teorici trattati nelle sezioni precedenti e la loro implementazione concreta.

Usiamo delle note a margine per indicare dove è possibile trovare dettagli sull'implementazione di Monero all'interno del codice sorgente.[15] Di solito viene indicato un percorso di file, ad esempio src/ringct/rctOps.cpp, e una funzione, come `ecdhEncode()`. Nota: il simbolo '-' indica un testo suddiviso, come crypto- note → cryptonote, e omettiamo quasi sempre i qualificatori di namespace (es. `Blockchain::`).

Non pensi che sia utile?

1.5 Avviso

Tutti gli schemi di firma, le applicazioni delle curve ellittiche e i dettagli dell'implementazione di Monero devono essere considerati solo a scopo descrittivo. I lettori che intendono utilizzare

[14] Si noti che le future versioni del protocollo di Monero, in particolare quelle che implementano nuovi protocolli di transazione, potrebbero rendere una o tutte queste idee impraticabili o impossibili da implementare.

[15] Le nostre note a margine sono accurate per la versione 0.15.x.x della suite software Monero, ma potrebbero diventare gradualmente inaccurate poiché il codice sorgente è in continuo cambiamento. Tuttavia, il codice è conservato in un repository git (https://github.com/monero-project/monero), quindi è disponibile una cronologia completa delle modifiche.

queste conoscenze per applicazioni pratiche serie (e non solo per esplorazioni amatoriali) dovrebbero consultare le fonti primarie e le specifiche tecniche (che abbiamo citato ove possibile). Gli schemi di firma richiedono prove di sicurezza ben validate, mentre i dettagli di implementazione di Monero sono disponibili nel codice sorgente di Monero. In particolare, come si suol dire in ambiti di sicurezza, "non reinventare la crittografia". Il codice che implementa primitive crittografiche dovrebbe essere accuratamente revisionato da esperti ed avere una lunga storia di affidabilità. Inoltre, i contributi originali presenti in questo documento potrebbero non essere stati adeguatamente revisionati e potrebbero non essere stati testati, quindi si invita il lettore a utilizzare il proprio giudizio nel leggerli.

1.6 Storia di Da Zero a Monero

Da Zero a Monero è un'espansione della tesi di laurea magistrale di Kurt Alonso, 'Monero - Privacy in the Blockchain' [29], pubblicata nel maggio del 2018. La prima edizione di questo documento è stata pubblicata circa un mese dopo, nel giugno 2018 [30].

Nella seconda edizione sono state migliorate le modalità in cui vengono introdotte le firme ad anello (Capitolo 3), riorganizzato la descrizione delle transazioni (aggiungendo il Capitolo 4 sugli indirizzi Monero), modernizzato il metodo usato per comunicare gli importi delle uscite (Sezione 5.3), sostituito le firme ad anello Borromean con Bulletproof (Sezione 5.5), deprecato `RCTTypeFull` (Capitolo 6), aggiornato e approfondito il sistema dinamico del peso dei blocchi e delle commissioni di rete (Capitolo 7), analizzato le prove correlate alle transazioni (Capitolo 8), descritto le multifirme di Monero (Capitolo 9), progettato soluzioni per i marketplace con deposito a garanzia (escrow) (Capitolo 10), proposto un nuovo protocollo decentralizzato per l'unione di transazioni chiamato TxTangle (Capitolo 11), aggiornato e aggiunto vari dettagli per allinearsi alla versione del protocollo più recente (v12) e alla suite software di Monero (v0.15.x.x), ed infine è stato revisionato il documento per migliorarne la leggibilità.[16]

1.7 Ringraziamenti

Scritto dall'autore 'koe'.

Questo report non esisterebbe senza la tesi originale di Kurt [29], a cui devo grande riconoscenza. I ricercatori del Monero Research Lab (MRL) Brandon "Surae Noether" Goodell e il pseudonimo 'Sarang Noether' (che ha collaborato con me alla Sezione 5.5 e nel Capitolo 8) sono stati una risorsa affidabile e competente durante lo sviluppo di entrambe le edizioni di Da Zero a Monero. Lo pseudonimo 'moneromooo', il più prolifico sviluppatore core del Monero Project, probabilmente ha la più vasta conoscenza del codice sorgente al mondo, e mi ha indicato la strada giusta innumerevoli volte. Naturalmente, molti altri meravigliosi contributori di Monero hanno dedicato tempo a

[16] Il codice sorgente LaTeX di Da Zero a Monero può essere trovato qui (prima edizione nel branch 'ztm1'): https://github.com/UkoeHB/Monero-RCT-report.

rispondere alle mie infinite domande. Infine, grazie a tutte le persone che ci hanno contattato con suggerimenti di correzione e commenti incoraggianti!

Parte I

Elementi Essenziali

CAPITOLO 2

Concetti Basilari

2.1 Notazione

L'obiettivo principale di questo documento è di raccogliere, revisionare, correggere e omologare tutte le informazioni esistenti riguardanti i meccanismi interni della criptovaluta Monero. Allo stesso tempo, gli autori hanno voluto fornire tutti i dettagli necessari per presentare il materiale in modo costruttivo e lineare. Un mezzo importante per raggiungere questo scopo è stato stabilire alcune convenzioni di notazione. Tra queste, abbiamo usato:

- lettere minuscole per indicare valori semplici, interi, stringhe, rappresentazioni binarie, ecc.
- lettere maiuscole per indicare punti su curve e costrutti più complessi.

Per gli elementi con un significato speciale, gli autori hanno cercato di usare, il più possibile, gli stessi simboli in tutto il documento. Per esempio, un generatore di curva è sempre indicato con G, il suo ordine con l, e chiavi private e pubbliche sono denotate, quando possibile, rispettivamente con K e k, ecc.

Oltre a questo, gli autori si sono impegnati nel fornire una presentazione *concettuale* chiara degli algoritmi e degli schemi. Il lettore con un background in informatica potrebbe trovare che siano state un po' trascurate questioni come la rappresentazione binaria degli elementi o, in alcuni casi, come eseguire operazioni concrete. Inoltre, anche gli studenti di matematica potrebbero notare che abbiamo evitato spiegazioni di algebra astratta.

Tuttavia, queste semplificazioni non sono da considerarsi un male. Un oggetto semplice come un intero o una stringa può sempre essere rappresentato come una sequenza di bit. Il cosiddetto "endianness" è raramente rilevante ed è per lo più una questione di convenzione.[1]

I punti su curve ellittiche sono normalmente indicati come coppie (x, y), possono quindi essere rappresentati con due numeri interi. Tuttavia, nel mondo della crittografia è comune applicare tecniche di *compressione dei punti*, che permettono di rappresentare un punto utilizzando solo lo spazio di una coordinata. Per l'approccio concettuale adottato da questo documento, l'uso della compressione dei punti è spesso superfluo, ma nella maggior parte dei casi si assume implicitamente che venga utilizzata.

È stato fatto un uso libero di funzioni hash crittografiche senza specificare in concreto gli algoritmi alla base. Nel caso di Monero, si tratta tipicamente di una variante di *Keccak*[2], ma a meno che non venga menzionato esplicitamente, ciò non è rilevante ai fini della teoria.

src/crypto/ `keccak.c`

Una funzione hash crittografica (d'ora in poi semplicemente "funzione hash" o "hash") prende in input un messaggio $\mathfrak{m}$ di lunghezza arbitraria e restituisce un hash h (o 'digest del messaggio') di lunghezza fissa, con ogni possibile output equiprobabile per un dato input. Le funzioni hash crittografiche sono difficili da invertire, possiedono una caratteristica molto interessante nota come *effetto valanga* che può far sì che messaggi molto simili generino hash molto diversi, ed è difficile trovare due messaggi con lo stesso digest.

Le funzioni hash saranno applicate a interi, stringhe, punti su curve o combinazioni di questi oggetti. Tali applicazioni devono essere interpretate come hash delle rappresentazioni in bit, o della concatenazione di tali rappresentazioni dei vari elementi. A seconda del contesto, il risultato di un hash potrà essere numerico, una stringa di bit o persino un punto su curva. Ulteriori dettagli a riguardo saranno forniti nel momento opportuno.

2.2 Aritmetica Modulare

La maggior parte della crittografia moderna parte dall'aritmetica modulare, che a sua volta inizia con l'operatore *modulo* (indicato con mod). In questo documento verrà preso in considerazione soltanto il modulo positivo, che restituisce sempre un numero intero positivo.

Il modulo positivo è simile al *resto* dopo la divisione tra due numeri, ad esempio c è il 'resto' di a/b. Si immagini una retta numerica: per calcolare $c = a \pmod{b}$ ci posizioniamo al punto a e

[1] Nella memoria del computer, ogni byte è memorizzato in un proprio indirizzo (un indirizzo è simile a una casella numerata in cui può essere memorizzato un byte). Una data *parola* o variabile è referenziata dall'indirizzo più basso dei suoi byte. Se la variabile x ha 4 byte, memorizzati negli indirizzi 10-13, l'indirizzo 10 viene usato per trovare x. Il modo in cui i byte di x sono organizzati nel suo insieme di indirizzi dipende dall'*endianness*, anche se ogni singolo byte è sempre memorizzato nello stesso modo all'interno del suo indirizzo. In pratica, quale estremo di x è memorizzato nell'indirizzo di riferimento? Può essere il *big end* oppure il *little end*. Dato $x = 0x12345678$ (esadecimale, 2 cifre esadecimali occupano 1 byte, ad esempio 8 cifre binarie, dette anche bit), e un array di indirizzi $\{10, 11, 12, 13\}$, la codifica big endian di x è $\{12, 34, 56, 78\}$ mentre la codifica little endian è $\{78, 56, 34, 12\}$. [76]

[2] The Keccak hashing algorithm forms the basis for the NIST standard *SHA-3* [99].

camminiamo verso lo zero con passi di ampiezza step $= b$ fino a raggiungere un numero intero ≥ 0 e $< b$. Quel numero è c. Ad esempio, 4 (modulo 3) = 1, $-5 \pmod 4 = 3$, e così via.

Formalmente, il modulo positivo per $c = a \pmod b$ è definito come $a = bx + c$, dove $0 \leq c < b$ e x è un intero con segno (mentre b è un intero maggiore di zero).

Nota che, se $a \leq n$ allora $-a \pmod n$ è uguale a $n - a$.

2.2.1 Somma e Moltiplicazione Modulari

In informatica è importante evitare l'uso di grandi numeri quando si compiono operazioni di aritmetica modulare. Ad esempio, se abbiamo $29 + 87 \pmod{99}$ e non abbiamo la possibilità di utilizzare tre o più cifre (come $116 = 29 + 87$), allora non è possibile calcolare $116 \pmod{99} = 17$ direttamente.

Per calcolare $c = a + b \pmod n$, dove a e b sono minori del modulo n, basta tener conto del seguente:

- Calcola $x = n - a$. Se $x > b$ allora $c = a + b$, altrimenti $c = b - x$.

Nell'aritmetica modulare l'operazione di moltiplicazione ($a * b \pmod n = c$) può essere eseguita attraverso l'algoritmo *double-and-add*. Introduciamolo attraverso un esempio: si supponga di voler calcolare $7 * 8 \pmod 9 = 2$. Che equivale a

$$7 * 8 = 8 + 8 + 8 + 8 + 8 + 8 + 8 \pmod 9$$

Raggruppando in gruppi di due, otteniamo:

$$(8 + 8) + (8 + 8) + (8 + 8) + 8$$

E ancora, per gruppi di due:

$$[(8 + 8) + (8 + 8)] + (8 + 8) + 8$$

Il numero totale di operazioni di somma + si riduce da 6 a 4 perché è necessario calcolare $(8 + 8)$ solo una volta.[3]

L'implementazione dell'algoritmo double-and-add comporta la conversione del primo numero (il 'moltiplicando' a) in binario (nel nostro esempio, $7 \rightarrow [0111]$). In seguito la rappresentazione binaria prodotta sarà utilizzata come un array di bit su cui si baseranno le operazioni previste dell'algoritmo.

Supponiamo di avere un array $A = [0111]$ di indice i che può assumere valori 3,2,1,0.[4] A[0] = 1 è il primo elemento di A ed è anche il bit meno significativo. Dichiariamo una variabile risultato con valore iniziale $r = 0$, ed una variabile somma $s = 8$ (in generale $s = b$). Seguiamo il seguente algoritmo:

[3] L'effetto di double-and-add diventa più evidente con grandi numeri. Ad esempio, con $2^{15} * 2^{30}$ somme dirette richiederebbe circa 2^{15} + operazioni, mentre double-and-add solo 15!

[4] Conosciuta anche come numerazione 'LSB 0', dato che il bit meno significativo ha indice 0. Al fine di facilitare l'approccio, la numerazione 'LSB 0' sarà utilizzata nel resto del capitolo, traslaciando eventuale convenzioni più accurate.

1. Per ogni: $i = (0, ..., A_{size} - 1)$
   (a) Se A[i] == 1, allora $r = r + s \pmod{n}$.
   (b) Calcola $s = s + s \pmod{n}$.
2. Usa l'ultimo risultato r: $c = r$.

Nell'esempio di prima $7 * 8 \pmod{9}$, l'esecuzione di questo algoritmo apparirà come segue:

1. $i = 0$
   (a) A[0] = 1, dunque $r = 0 + 8 \pmod{9} = 8$
   (b) $s = 8 + 8 \pmod{9} = 7$
2. $i = 1$
   (a) A[1] = 1, dunque $r = 8 + 7 \pmod{9} = 6$
   (b) $s = 7 + 7 \pmod{9} = 5$
3. $i = 2$
   (a) A[2] = 1, dunque $r = 6 + 5 \pmod{9} = 2$
   (b) $s = 5 + 5 \pmod{9} = 1$
4. $i = 3$
   (a) A[3] = 0, dunque r rimane uguale
   (b) $s = 1 + 1 \pmod{9} = 2$
5. $r = 2$ è il risultato

2.2.2 Potenza Modulo

Definita l'operazione di elevamento alla potenza $8^7 \pmod{9} = 8*8*8*8*8*8*8 \pmod{9}$. Come *double-and-add*, esiste anche l'algoritmo *square-and-multiply*. In generale $a^e \pmod{n}$ l'algoritmo è implementato come segue:

1. Definita $e_{scalar} \to e_{binary}$; $A = [e_{binary}]$; $r = 1$; $m = a$
2. Per ogni: $i = (0, ..., A_{size} - 1)$
   (a) Se A[i] == 1, allora $r = r * m \pmod{n}$.
   (b) Calcola $m = m * m \pmod{n}$.
3. Usa l'ultimo r come risultato.

2.2.3 Inverso Moltiplicativo Modulo

A volte abbiamo bisogno di $1/a \pmod n$, o detto diversamente $a^{-1} \pmod n$. L'inverso di qualcosa moltiplicato per se stesso è per definizione uno (identità). Immagina $0.25 = 1/4$, e quindi $0.25 \times 4 = 1$.

Nell'aritmetica modulare, per $c = a^{-1} \pmod n$, vale $ac \equiv 1 \pmod n$ con $0 \leq c < n$ e con a e n *coprimi*.[5] Due numeri *coprimi* non condividono alcun divisore tranne 1 (la frazione a/n non può essere ridotta/semplificata).

Possiamo usare il metodo 'square-and-multiply' per calcolare l'inverso moltiplicativo modulo quando n è un numero primo, grazie al *piccolo teorema di Fermat*:[6]

$$\begin{aligned} a^{n-1} &\equiv 1 \pmod n \\ a \cdot a^{n-2} &\equiv 1 \pmod n \\ c \equiv a^{n-2} &\equiv a^{-1} \pmod n \end{aligned}$$

In generale (e più rapidamente), il cosiddetto 'algoritmo esteso di Euclide' [7] può trovare anche gli inversi modulari.

2.2.4 Equazioni Modulari

Supponiamo di avere un'equazione $c = 3\times4\times5 \pmod 9$. Calcolarla è semplice. Dato un operatore $\circ$ (per esempio, $\circ = *$) tra due espressioni A e B:

$$(A \circ B) \pmod n = [A \pmod n] \circ [B \pmod n] \pmod n$$

Nel nostro esempio poniamo $A = 3 \times 4$, $B = 5$ e $n = 9$:

$$\begin{aligned} (3 \times 4 \times 5) \pmod 9 &= [3 \times 4 \pmod 9] \times [5 \pmod 9] \pmod 9 \\ &= [3] \times [5] \pmod 9 \\ c &= 6 \end{aligned}$$

Ora abbiamo un modo per fare la sottrazione in modulo.

$$\begin{aligned} A - B \pmod n &\rightarrow A + (-B) \pmod n \\ &\rightarrow [A \pmod n] + [-B \pmod n] \pmod n \end{aligned}$$

[5] Nell'equazione $a \equiv b \pmod n$, a è *congruente* a $b \pmod n$, il che significa semplicemente che $a \pmod n = b \pmod n$.

[6] L'inverso moltiplicativo modulo soddisfa la seguente regola:
Se $ac \equiv b \pmod n$ con a e n coprimi, la soluzione a questa congruenza lineare è data da $c = a^{-1}b \pmod n$.[10]
Ciò significa che possiamo fare $c = a^{-1}b \pmod n \rightarrow ca \equiv b \pmod n \rightarrow a \equiv c^{-1}b \pmod n$.

Lo stesso principio si applica a qualcosa come $x = (a - b \times c \times d)^{-1}(e \times f + g^h) \pmod n$.[7]

2.3 Crittografia a Curva Ellittica

2.3.1 Le Curve Ellittiche

Un campo primo $\mathbb{F}_q$, dove q è un numero primo maggiore di 3, è un insieme formato dagli elementi $\{0, 1, 2, ..., q-1\}$. Somma e moltiplicazione $(+, \cdot)$ e cambio di segno $(-)$ sono calcolati $\pmod q$.

`fe`: elemento del campo

"Calcolato $\pmod q$" significa che $\pmod q$ è eseguito in qualche istanza di un'operazione aritmetica tra due elementi del campo (o cambio di segno di un singolo elemento del campo). Ad esempio, dato un campo primo $\mathbb{F}_p$ con $p = 29$, $17 + 20 = 8$ perché $37 \pmod{29} = 8$. Anche $-13 = -13 \pmod{29} = 16$.

Tipicamente, data una coppia (a, b) è possibile definire una curva ellittica come l'insieme dei punti (x, y) che soddisfano l'equazione di *Weierstraß* [67]:[8]

$$y^2 = x^3 + ax + b \quad \text{dove} \quad a, b, x, y \in \mathbb{F}_q$$

La criptovaluta Monero utilizza una particolare curva che appartiene alla categoria di curve ellittiche *Twisted Edwards* [35], le quali sono comunemente espresse nella forma (data una coppia (a, d)):

$$ax^2 + y^2 = 1 + dx^2y^2 \quad \text{dove} \quad a, d, x, y \in \mathbb{F}_q$$

Il vantaggio che offre la seconda rispetto alla prima forma di Weierstraß è quello di ridurre le operazioni di aritmetica modulare necessarie alle primitive crittografiche, consentendo di costruire algoritmi crittografici più veloci (vedi Bernstein *et al.* in [37] per dettagli).

Siano $P_1 = (x_1, y_1)$ e $P_2 = (x_2, y_2)$ due punti appartenenti ad una curva ellittica Twisted Edwards (d'ora in poi EC per semplicità). Definiamo la somma tra i punti $P_1 + P_2 = (x_1, y_1) + (x_2, y_2)$ come punto $P_3 = (x_3, y_3)$ dove[9]

$$x_3 = \frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2} \pmod q$$
$$y_3 = \frac{y_1y_2 - ax_1x_2}{1 - dx_1x_2y_1y_2} \pmod q$$

[7] Il calcolo del modulo di numeri grandi può sfruttare le equazioni modulari. Per esempio, $254 \pmod{13} \equiv 2 \times 10 \times 10 + 5 \times 10 + 4 \equiv (((2) \times 10 + 5) \times 10 + 4) \pmod{13}$. Un algoritmo per calcolare $a \pmod n$ quando $a > n$ è:

1. Definire $A \to [a_{decimal}]$; $r = 0$
2. Per $i = A_{size} - 1, ..., 0$
   (a) $r = (r \times 10 + A[i]) \pmod n$
3. Usare il valore finale di r come risultato.

[8] Nota: La dicitura $a \in \mathbb{F}$ indica che a è un qualche elemento del campo $\mathbb{F}$.

[9] Tipicamente i punti di una curva ellittica sono convertiti in coordinate proiettive prima di operazioni come somma tra punti, in modo tale da evitare di eseguire inversioni di campo migliorando così l'efficienza. [140]

src/crypto/crypto_ops_builder/ref10CommentedCombined/ge.h

Queste formule per l'addizione valgono anche per il raddoppio di un punto; cioè, quando $P_1 = P_2$. Per sottrarre un punto, si invertono le sue coordinate rispetto all'asse y, $(x, y) \rightarrow (-x, y)$ [35], e si usa l'addizione di punti. Ricordiamo che gli elementi 'negativi' $-x$ di $\mathbb{F}_q$ sono in realtà $-x \pmod q$.

Ogni volta che si sommano due punti di una curva, il risultato P_3 è anch'esso un punto appartenente alla curva ellittica, ovvero tutti $x_3, y_3 \in \mathbb{F}_q$ soddisfano l'equazione della EC.

Ogni punto P appartenente ad una curva ellittica può generare un sottogruppo di ordine (dimensione) u composto da alcuni altri punti della curva multipli del punto P. Ad esempio, il sottogruppo di un punto P può avere ordine 5 e contenere i punti $(0, P, 2P, 3P, 4P)$, ciascuno appartenente alla curva. Al punto $5P$ compare il cosiddetto *punto di divergenza*, che è come la posizione 'zero' su una curva ellittica, e ha coordinate $(0, 1)$.[10]

Per convenzione, $5P + P = P$. Questo significa che il sottogruppo è *ciclico*.[11] Tutti i punti P nella curva generano un sottogruppo ciclico. Se P genera un sottogruppo di ordine primo, allora tutti i punti inclusi (eccetto il punto di divergenza) generano lo stesso sottogruppo. Nel nostro esempio, consideriamo i multipli del punto $2P$:

$$2P, 4P, 6P, 8P, 10P \rightarrow 2P, 4P, 1P, 3P, 0$$

Un altro esempio: un sottogruppo di ordine 6 $(0, P, 2P, 3P, 4P, 5P)$. Multipli del punto $2P$:

$$2P, 4P, 6P, 8P, 10P, 12P \rightarrow 2P, 4P, 0, 2P, 4P, 0$$

Qui $2P$ ha ordine 3. Poiché 6 non è primo, non tutti i punti del sottogruppo ricreano il sottogruppo originale.

Ogni curva ellittica ha un ordine N pari al numero totale di punti sulla curva (incluso il punto di divergenza), e gli ordini di tutti i sottogruppi generati dai punti sono divisori di N (secondo il *teorema di Lagrange*). Possiamo immaginare l'insieme di tutti i punti della curva $\{0, P_1, ..., P_{N-1}\}$. Se N non è primo, alcuni punti formeranno sottogruppi con ordini pari ai divisori di N.

Per trovare l'ordine u del sottogruppo generato da un punto P:

1. Trova N (ad esempio usando il *algoritmo di Schoof*).
2. Trova tutti i divisori di N.
3. Per ogni divisore n di N, calcola nP.
4. Il più piccolo n tale che $nP = 0$ è l'ordine u del sottogruppo.

[10] Si scopre che le curve ellittiche hanno una struttura di *gruppo abeliano* rispetto all'operazione di addizione descritta, dato che il punto di divergenza corrisponde all'elemento identità. Una definizione concisa di questo concetto si trova su https://brilliant.org/wiki/abelian-group/.

[11] Sottogruppo ciclico significa che, per il sottogruppo di P con ordine u, e per ogni intero n, vale $nP = [n \pmod u]P$. Possiamo immaginarci su un punto su una sfera a una certa distanza da una posizione 'zero', e ogni passo sposta di quella distanza. Dopo un certo numero di passi torneremo al punto di partenza, anche se può servire più di una rivoluzione per arrivarci esattamente. Il numero di passi per tornare esattamente al punto iniziale è l''ordine' del gruppo di passi, e tutte le nostre posizioni sono punti unici di quel gruppo. Si consiglia di applicare questo concetto anche ad altre idee discusse qui.

Le curve ellittiche scelte per la crittografia tipicamente hanno $N = hl$, dove l è un numero primo sufficientemente grande (ad esempio 160 bit) e h è il cosiddetto *cofattore*, che può essere piccolo come 1 o 2.[12] Di norma, un punto nel sottogruppo di dimensione l è scelto come generatore G. Per ogni altro punto P in quel sottogruppo esiste un intero $0 < n \leq l$ tale che $P = nG$.

`ge`: elemento del gruppo

Approfondiamo. Supponiamo che esista un punto P' con ordine N, dove $N = hl$. Ogni altro punto P_i può essere trovato come $P_i = n_i P'$ per qualche intero n_i. Se $P_1 = n_1 P'$ ha ordine l, ogni $P_2 = n_2 P'$ con ordine l deve essere nello stesso sottogruppo di P_1, poiché $lP_1 = 0 = lP_2$, e se $l(n_1 P') \equiv l(n_2 P') \equiv NP' = 0$, allora n_1 e n_2 devono essere multipli di h. Dato che $N = hl$, ci sono solo l multipli di h, implicando che è possibile un solo sottogruppo di dimensione l.

In parole semplici, il sottogruppo formato dai multipli di (hP') contiene sempre P_1 e P_2. Inoltre, $h(n'P') = 0$ quando n' è multiplo di l, e ci sono solo h variazioni di n' $\pmod N$ (incluso il punto di divergenza per $n' = hl$) perché quando $n' = hl$ si ritorna a 0: $hlP' = 0$. Quindi, ci sono solo h punti P nella curva tale che $hP = 0$.

Un ragionamento simile vale per qualunque sottogruppo di ordine u. Qualsiasi coppia di punti P_1 e P_2 di ordine u appartiene allo stesso sottogruppo, formato dai multipli di $(N/u)P'$.

Con questa nuova comprensione è chiaro che possiamo usare il seguente algoritmo per trovare punti (diversi dal punto di divergenza) nel sottogruppo di ordine l:

1. Trova N della curva ellittica EC, scegli l'ordine l del sottogruppo, calcola $h = N/l$.
2. Scegli un punto casuale P' in EC.
3. Calcola $P = hP'$.
4. Se $P = 0$ torna al passo 2, altrimenti P è nel sottogruppo di ordine l.

Calcolare il prodotto scalare tra un intero n e un punto P, nP, non è difficile, mentre trovare n tale che $P_1 = nP_2$ è considerato computazionalmente difficile. Per analogia con l'aritmetica modulare, questo problema è chiamato *problema del logaritmo discreto* (DLP). La moltiplicazione scalare può essere vista come una *funzione a senso unico*, che apre la strada all'uso delle curve ellittiche in crittografia.[13]

`sc`: scalare

Il prodotto scalare nP equivale a $(((P + P) + (P + P))\ldots)$. Anche se non sempre è il metodo più efficiente, possiamo usare la tecnica del double-and-add come nella Sezione 2.2.1. Per ottenere la somma $R = nP$, si ricorda la definizione dell'operazione di somma $+$ tra punti discussa nella Sezione 2.3.1.

1. Data una funzione $n_{scalar} \to n_{binary}$; $A = [n_{binary}]$; $R = 0$, il punto di divergenza; $S = P$

[12] Curve con piccoli cofattori permettono un'addizione di punti più veloce, ecc. [35].

[13] Non esistono equazioni o algoritmi noti che risolvano efficientemente (con la tecnologia disponibile) per n in $P_1 = nP_2$, il che significa che potrebbero servire moltissimi anni per risolvere un singolo prodotto scalare.

2. Itera per $i = (0, \ldots, A_{size} - 1)$
   - $b = A_i$ è il bit di indice i di n
   - $R = R + R$
   - Se $b = 1$, $R = R + S$

Si noti che gli scalari delle curve ellittiche per i punti nel sottogruppo di ordine l (che useremo da ora in avanti) appartengono al campo finito $\mathbb{F}_l$. Questo significa che l'aritmetica tra scalari è modulo l.

2.3.2 Crittografia a Chiave Pubblica con Curve Ellittiche

Gli algoritmi di crittografia a chiave pubblica possono essere concepiti in modo analogo all'aritmetica modulare.

Sia k un numero casuale tale che $0 < k < l$, esso corrisponde alla *chiave privata (private key)*.[14] È possibile calcolare la *chiave pubblica (public key)* K (un punto sulla EC) tramite il prodotto scalare $kG = K$.

Per via del *problema del logaritmo discreto (discrete logarithm problem)* (DLP) risulta difficile dedurre k da K soltanto. Ciò consente di utilizzare (k, K) negli algoritmi crittografici a chiave pubblica standard.

2.3.3 Scambio Chiavi Diffie-Hellman con Curve Ellittiche

Uno scambio *Diffie-Hellman* [49] basilare di un segreto condiviso tra *Alice* e *Bob* può avvenire nella seguente modalità:

1. Alice e Bob generano le loro chiavi private e pubbliche (k_A, K_A) e (k_B, K_B). Entrambi pubblicano o scambiano le loro chiavi pubbliche, senza rivelare quelle private.
2. Il segreto condiviso si definisce come
   $$S = k_A K_B = k_A k_B G = k_B k_A G = k_B K_A$$
   Alice può calcolare privatamente $S = k_A K_B$, e Bob $S = k_B K_A$, in modo tale da permettere di utilizzare questo valore come segreto condiviso.

   Ad esempio, se Alice ha un messaggio m da inviare a Bob, può calcolare l'hash del segreto condiviso $h = \mathcal{H}(S)$, calcolare $x = m + h$, ed inviare x a Bob. Bob calcolerà in seguito $h' = \mathcal{H}(S)$, e $m = x - h'$, ottenendo così m.

Un osservatore esterno non sarà in grado di calcolare facilmente il segreto condiviso a causa del 'Problema di Diffie-Hellman' (DHP), il quale afferma che trovare S da K_A e K_B è veramente difficile. Inoltre, il DLP rende impossibile trovare k_A o k_B.[15]

[14] La chiave privata viene chiamata anche *chiave segreta (secret key)*. Ciò ci consente di abbreviare: pk = public key, sk = secret key.

[15] Si ritiene che il DHP abbia almeno una difficoltà simile al DLP, sebbene non sia stato dimostrato. [28]

2.3.4 Firme di Schnorr e la Trasformazione di Fiat-Shamir

Nel 1989 Claus-Peter Schnorr pubblica un protocollo di autenticazione interattivo [120], generalizzato da Maurer nel 2009 [84], che permette a chiunque di dimostrare la conoscenza di una chiave privata k corrispondente ad una certa chiave pubblica K senza rivelare nessuna informazione su di essa [86]. A grandi linee, funziona come segue:

1. Il dimostratore genera un numero intero casuale $\alpha \in_R \mathbb{Z}_l$,[16] calcola αG, ed invia αG al verificatore.
2. Il verificatore genera una *sfida* casuale $c \in_R \mathbb{Z}_l$ ed invia c al dimostratore.
3. Il dimostratore calcola la *risposta* $r = \alpha + c * k$ ed invia r al verificatore.
4. Il verificatore calcola $R = rG$ e $R' = \alpha G + c * K$, e verifica che $R \stackrel{?}{=} R'$.

Il verificatore può calcolare $R' = \alpha G + c * K$ prima del dimostratore, in modo tale che inviare c in grosso modo corrisponda a chiedere "sfida: qual è il logaritmo discreto di R'?." Una sfida a cui il dimostratore può fornire una risposta giusta con una probabilità trascurabile, se non è a conoscenza della chiave k.

Se α è stato scelto in maniera casuale dal dimostratore allora r è distribuita in maniera casuale [121] e k è sicura a livello di teoria dell'informazione all'interno di r (anche se può essere trovata risolvendo DLP per K o αG).[17] Tuttavia, se il dimostratore riutilizza α per dimostrare la conoscenza di k, chiunque sia a conoscenza di entrambe le sfide in $r = \alpha + c * k$ e $r' = \alpha + c' * k$ può calcolare k (due equazioni, due incognite).[18]

$$k = \frac{r - r'}{c - c'}$$

Se il dimostratore è a conoscenza di c sin dall'inizio (ad esempio, gli è stato fornito segretamente dal verificatore), può generare una risposta casuale r e calcolare $\alpha G = rG - cK$. In seguito all'invio di r al verificatore, viene 'dimostrata' la conoscenza di k senza esserne effettivamente a conoscenza. Un osservatore della trascrizione degli scambi tra il proponente e il verificatore non ne ricaverebbe alcuna informazione aggiuntiva. Lo schema non è *pubblicamente verificabile.* [86]

Nel suo ruolo di sfidante, il verificatore sputa fuori un numero casuale dopo aver ricevuto αG, rendendolo equivalente a una *funzione casuale*. Le funzioni casuali, come le funzioni hash, sono note come oracoli casuali perché calcolarne una equivale a richiedere un numero casuale a qualcuno

[16] Notazione: La R in $\alpha \in_R \mathbb{Z}_l$ significa che α è estratto casualmente da $\{1, 2, 3, ..., l - 1\}$. In altre parole, $\mathbb{Z}_l$ sono tutti interi (mod l). Escludiamo 'l' in quanto il punto-ad-infinito non è utile in questo contesto.

src/crypto/crypto.cpp `random32_unbiased()`

[17] Un sistema crittografico con sicurezza a livello di teoria dell'informazione è ingestibile anche da un attaccante dotato di potenza di calcolo virtualmente infinita, e ciò è dovuto dal semplice fatto che l'attaccante non dispone di sufficienti informazioni.

[18] Supponiamo che il dimostratore sia un computer, e immaginiamo che qualcuno cloni/copi il computer dopo che ha generato α, then presenting each copy with a different challenge.

[86].[19]

L'uso di una funzione hash, al posto del verificatore, per generare le sfide è noto come *trasformazione di Fiat-Shamir* [57], perché rende una prova interattiva non interattiva e pubblicamente verificabile [86].[20] [21]

Prove Non interattive

1. Generare un numero casuale $\alpha \in_R \mathbb{Z}_l$, e calcola αG.
2. Calcolare la sfida usando una funzione hash crittografica sicura, $c = \mathcal{H}([\alpha G])$.
3. Definire il risultato $r = \alpha + c * k$.
4. Pubblicare la coppia di prove $(\alpha G, r)$.

Verifica

1. Calcolare la sfida: $c' = \mathcal{H}([\alpha G])$.
2. Calcolare $R = rG$ e $R' = \alpha G + c' * K$.
3. Se $R = R'$ allora il dimostratore conosce k con altissima probabilità.

Come Funziona

$$\begin{aligned} rG &= (\alpha + c * k)G \\ &= (\alpha G) + (c * kG) \\ &= \alpha G + c * K \\ R &= R' \end{aligned}$$

Una parte importante di qualsiasi schema di prova o firma sono le risorse necessarie per la loro verifica. Questo include lo spazio necessario per memorizzare le prove e il tempo richiesto per la verifica. In questo schema memorizziamo un punto su curva ellittica (EC point) e un intero, ed è necessario conoscere la chiave pubblica — ovvero un altro punto sulla curva ellittica. Poiché le funzioni hash sono relativamente veloci da calcolare, è importante tenere presente che il tempo di verifica dipende principalmente dalle operazioni effettuate su curve ellittiche.

src/ringct/rctOps.cpp

[19] Più in generale, "nella crittografia... un oracolo è qualsiasi sistema in grado di fornire informazioni aggiuntive su un sistema, altrimenti non disponibili." [3]

[20] L'output di una funzione hash crittografica $\mathcal{H}$ è distribuito uniformemente sull'insieme dei possibili risultati. In altre parole, per un certo input A, $\mathcal{H}(A) \in_R^D \mathbb{S}_H$, dove $\mathbb{S}_H$ è l'insieme dei possibili output della funzione $\mathcal{H}$. Usiamo $\in_R^D$ per indicare che la funzione è deterministica ma con comportamento casuale: $\mathcal{H}(A)$ produce sempre lo stesso risultato, ma il suo output è equivalente a un numero casuale.

[21] Si noti che le prove (e firme) non interattive in stile Schnorr richiedono l'uso di un generatore G fisso, oppure l'inclusione del generatore nell'hash della sfida. Includerlo in questo modo è noto come *key prefixing*, di cui parleremo più avanti (Sezioni 3.4 e 9.2.3).

2.3.5 Firma dei Messaggi

Tipicamente, una firma crittografica viene eseguita sull'hash crittografico di un messaggio, piuttosto che sul messaggio stesso, per facilitare la firma di messaggi di dimensioni variabili. Tuttavia, in questo documento useremo il termine 'messaggio' in senso lato, e il suo simbolo $\mathfrak{m}$, per riferirci sia al messaggio in senso proprio sia al suo valore di hash, salvo diversa specifica.

La firma dei messaggi è una componente fondamentale della sicurezza su Internet, che consente al destinatario di un messaggio di avere la certezza che il contenuto sia quello inviato dal firmatario. Uno schema di firma molto comune è chiamato ECDSA. Vedi [70], ANSI X9.62 e [67].

Lo schema di firma che presentiamo qui è una formulazione alternativa della prova di Schnorr trasformata vista in precedenza. Pensare alle firme in questo modo prepara il lettore alle firme ad anello trattate nel prossimo capitolo.

Firma

Si supponga che Alice possieda la coppia di chiavi private e pubbliche (k_A, K_A). Per firmare in modo univoco un messaggio arbitrario $\mathfrak{m}$, può eseguire i seguenti passaggi:

1. Generare un numero casuale $\alpha \in_R \mathbb{Z}_l$ e calcolare αG.
2. Calcolare la sfida usando una funzione hash crittograficamente sicura: $c = \mathcal{H}(\mathfrak{m}, [\alpha G])$.
3. Definire la risposta r in modo tale che $\alpha = r + c * k_A$, ovvero $r = \alpha - c * k_A$.
4. Pubblicare la firma (c, r).

Verifica

Qualsiasi terza parte che conosca i parametri di dominio della curva ellittica (che specificano quale curva è stata utilizzata), la firma (c, r) e il metodo di firma, $\mathfrak{m}$ e la funzione hash, nonché la chiave pubblica K_A, può verificare la firma nel seguente modo:

1. Calcolare la sfida: $c' = \mathcal{H}(\mathfrak{m}, [rG + c * K_A])$.
2. Se $c = c'$, allora la firma è valida.

In questo schema di firma sono memorizzati due scalari ed è richiesta una chiave pubblica su curva ellittica.

Perché Funziona

Questo deriva dal fatto che

$$\begin{aligned} rG &= (\alpha - c * k_A)G \\ &= \alpha G - c * K_A \\ \alpha G &= rG + c * K_A \\ \mathcal{H}_n(\mathfrak{m}, [\alpha G]) &= \mathcal{H}_n(\mathfrak{m}, [rG + c * K_A]) \\ c &= c' \end{aligned}$$

Pertanto, la persona in possesso di k_A (Alice) ha generato (c, r) per $\mathfrak{m}$: dunque ha firmato il messaggio. La probabilità che qualcun altro, un falsificatore privo di k_A, possa aver prodotto (c, r) è trascurabile, quindi il verificatore può essere ragionevolmente certo che il messaggio non sia stato manomesso.

2.4 Curva Ed25519

Monero utilizza per le operazioni crittografiche una particolare curva ellittica, chiamata Twisted Edwards, *Ed25519*, che è *birazionalmente equivalente*[22] alla curva di Montgomery *Curve25519*.

Sia la curva 25519 che Ed25519 sono state sviluppate e pubblicate da Bernstein *et al.* [35, 36, 37].[23]

La curva è definita sul campo primo $\mathbb{F}_{2^{255}-19}$ (cioè $q = 2^{255} - 19$) mediante la seguente equazione:

src/crypto/crypto_ops_builder/ref10CommentedCombined/ge.h

$$-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$$

Questa curva affronta molte delle problematiche sollevate dalla comunità crittografica.[24] È risaputo che gli algoritmi standardizzati dal NIST[25] presentano delle criticità. Ad esempio, è emerso recentemente che l'algoritmo di generazione di numeri casuali PNRG (la versione basata su curve ellittiche) presenta delle falle e potrebbe contenere una backdoor [66]. In un contesto più ampio, le autorità di standardizzazione come il NIST portano a una monocultura crittografica, introducendo un punto di centralizzazione. Un esempio lampante si è avuto quando la NSA ha sfruttato la sua influenza sul NIST per indebolire uno standard crittografico internazionale [17].

La curva Ed25519 non è soggetta a brevetti (vedi [77] per una discussione sull'argomento), e il team che l'ha sviluppata ha progettato e adattato gli algoritmi crittografici di base con attenzione all'efficienza [37].

src/crypto/crypto_ops_builder/

src/wallet/ringdb.cpp

[22] Senza entrare nei dettagli, l'equivalenza birazionale può essere intesa come un isomorfismo esprimibile tramite termini razionali.

[23] Il Dr. Bernstein ha anche sviluppato uno schema di cifratura noto come ChaCha [34, 98], che l'implementazione principale di Monero utilizza per cifrare alcune informazioni sensibili relative ai portafogli degli utenti.

[24] Anche se una curva non presenta apparenti problemi di sicurezza crittografica, è possibile che la persona o organizzazione che l'ha creata conosca una debolezza nascosta che si manifesta solo in curve molto rare. Tale persona potrebbe generare casualmente molte curve fino a trovarne una con una vulnerabilità segreta, dunque non nota. Fornire spiegazioni ragionevoli e dettagliati per i parametri della curva rende molto più difficile introdurre debolezze occulte. La curva Ed25519 è nota come per essere 'completamente rigida', ovvero il suo processo di generazione è stato completamente documentato. [119]

[25] National Institute of Standards and Technology, https://www.nist.gov/

Le curve Twisted Edwards hanno un ordine esprimibile come $N = 2^c l$, dove l è un numero primo e c un intero positivo. Nel caso della curva Ed25519, il suo ordine è un numero di 76 cifre (l è lungo 253 bit):[26]

$$2^3 \cdot 7237005577332262213973186563042994240857116359379907606001950938285454250989$$

src/ringct/rctOps.h `curveOrder()`

2.4.1 Rappresentazione Binaria

Gli elementi di $\mathbb{F}_{2^{255}-19}$ sono codificati come interi a 256 bit, quindi possono essere rappresentati usando 32 byte. Poiché ogni elemento richiede solo 255 bit, il bit più significativo è sempre zero.

Mentre, qualsiasi punto su una curva Ed25519 può essere espresso usando 64 byte. Tuttavia, applicando tecniche di *compressione dei punti*, descritte qui di seguito, è possibile ridurre questa dimensione della metà, ovvero a 32 byte.

2.4.2 Compressione dei Punti

I punti di una curva Ed25519 possono essere facilmente compressi, in modo che la rappresentazione di un punto occupi solo lo spazio di una coordinata. Non entreremo nei dettagli matematici necessari per giustificare questo, ma possiamo dare una breve intuizione di come funziona [72]. Lo schema di compressione dei punti per la curva Ed25519 è stato descritto per la prima volta in [36], mentre il concetto è stato introdotto originariamente in [91].

Questo schema di compressione deriva da una trasformazione dell'equazione della curva Twisted Edwards (assumendo $a = -1$, valido per Monero):

$$x^2 = \frac{y^2 - 1}{dy^2 + 1}$$

Questa formula[27] indica che per ogni valore di y esistono due possibili valori di x (positivo o negativo). Gli elementi del campo x e y sono calcolati modulo q, quindi non esistono valori negativi propriamente detti. Tuttavia, calcolare modulo q di $-x$ cambia la parità del numero poiché q è dispari. Per esempio: $3 \pmod 5 = 3$, mentre $-3 \pmod 5 = 2$. In altre parole, gli elementi di campo x e $-x$ differiscono per la loro parità.

Se supponga di conoscere un punto della curva dove x è pari. Calcolando l'equazione trasformata con y si ottiene un numero dispari, allora è possibile dedurre che negando quel numero otterremo il valore corretto di x. Un singolo bit può trasmettere questa informazione, e per comodità la coordinata y ha un bit extra.

Supponiamo di voler comprimere il punto (x, y).

Codifica Impostiamo il bit più significativo di y a 0 se x è pari, e a 1 se x è dispari. Il valore risultante y' rappresenterà il punto sulla curva.

src/crypto/crypto_ops_builder/ref10CommentedCombined/ge_tobytes.c

[26] Ciò significa che le chiavi private EC in Ed25519 sono lunghe 253 bit.

[27] $d = -\frac{121665}{121666}$, in questo caso.

Decodifica

1. Recuperiamo il punto compresso y', quindi copiamo il suo bit più significativo nel bit di parità b prima di azzerarlo. Ora abbiamo di nuovo il valore originale di y.
2. Poniamo $u = y^2 - 1 \pmod q$ e $v = dy^2 + 1 \pmod q$. Ciò significa che $x^2 = u/v \pmod q$.
3. Calcoliamo[28]
$$z = uv^3(uv^7)^{(q-5)/8} \pmod q.$$
   (a) Se $vz^2 = u \pmod q$, allora $x' = z$.
   (b) Se $vz^2 = -u \pmod q$, calcoliamo $x' = z \cdot 2^{(q-1)/4} \pmod q$.
4. Usando il bit di parità b del primo passo, se $b \neq$ il bit meno significativo di x', allora $x = -x' \pmod q$, altrimenti $x = x'$.
5. Restituiamo il punto decompressato (x, y).

ge_from-bytes.c

Le implementazioni di Ed25519 (come quella di Monero) usano tipicamente il generatore $G = (x, 4/5)$ [36], dove x è la variante 'pari', ovvero con $b = 0$, basata sulla decompressione del punto $y = 4/5 \pmod q$.

2.4.3 Algoritmo di Firma EdDSA

Bernstein e il suo team hanno sviluppato numerosi algoritmi basati sulla curva Ed25519.[29] A scopo illustrativo, sarà descritta una versione altamente ottimizzata e sicura dell'algoritmo di firma ECDSA che, secondo gli autori, permette di produrre oltre 100.000 firme al secondo usando un processore Intel Xeon standard [36]. L'algoritmo è descritto anche nell'RFC Internet 8032 [73]. Si noti che si tratta di uno schema di firma molto simile a Schnorr.

Tra le varie caratteristiche, invece di generare interi casuali ogni volta, utilizza un valore hash derivato dalla chiave privata del firmatario e dal messaggio stesso. Questo evita vulnerabilità legate all'implementazione di generatori di numeri casuali. Inoltre, un altro obiettivo dell'algoritmo è evitare l'accesso a locazioni di memoria riservata (o in maniera imprevedibile) per prevenire i cosiddetti *attacchi basati sui tempi di cache* (*cache timing attacks*) [36].

Di seguito è riportata una sintesi dei passi eseguiti dall'algoritmo. Una descrizione completa e un esempio di implementazione in Python sono disponibili in [73].

Firma

1. Sia h_k l'hash $\mathcal{H}(k)$ della chiave privata k del firmatario. Calcoliamo α come hash $\alpha = \mathcal{H}(h_k, \mathfrak{m})$ dell'hash della chiave privata e del messaggio. A seconda dell'implementazione, $\mathfrak{m}$ può essere il messaggio originale oppure il suo hash [73].

[28] Poiché $q = 2^{255} - 19 \equiv 5 \pmod 8$, $(q-5)/8$ e $(q-1)/4$ sono interi.

[29] Si veda [37] per operazioni di gruppo efficienti sulle curve Twisted Edwards (ad esempio somma di punti, raddoppio, somma mista, ecc.). Si veda [33] per aritmetica modulare efficiente.

src/crypto/crypto_ops_builder/ref10CommentedCombined/

2. Calcolare αG e la sfida $ch = \mathcal{H}([\alpha G], K, \mathfrak{m})$.
3. Calcolare la risposta $r = \alpha + ch \cdot k$.
4. La firma è la coppia $(\alpha G, r)$.

Verifica

La verifica si effettua come segue:

1. Calcolare $ch' = \mathcal{H}([\alpha G], K, \mathfrak{m})$.
2. Se vale l'uguaglianza
$$2^c rG \stackrel{?}{=} 2^c \alpha G + 2^c ch' \cdot K,$$
allora la firma è valida.

Il termine 2^c deriva dalla forma generale dell'algoritmo EdDSA descritta da Bernstein *et al.* [36]. Secondo quel documento, anche se non è strettamente necessario per una verifica adeguata, rimuovere 2^c fornisce equazioni più robuste.

La chiave pubblica K può essere un qualsiasi punto della curva ellittica, ma è preferibile utilizzare punti appartenenti al sottogruppo generato da G. Moltiplicare per il cofattore 2^c garantisce che tutti i punti siano in quel sottogruppo. In alternativa, il verificatore può controllare che $lK \stackrel{?}{=} 0$, che è verificata solo se K appartiene effettivamente al sottogruppo. Non si conoscono vulnerabilità legati a questi metodi, e come vedremo, il secondo metodo è importante in Monero (Sezione 3.4).

In questo schema di firma viene memorizzato un punto EC e uno scalare, e si dispone di una chiave pubblica EC.

Perché Funziona

$$\begin{aligned} 2^c rG &= 2^c(\alpha + \mathcal{H}([\alpha G], K, \mathfrak{m}) \cdot k) \cdot G \\ &= 2^c \alpha G + 2^c \mathcal{H}([\alpha G], K, \mathfrak{m}) \cdot K. \end{aligned}$$

Rappresentazione Binaria

Di default, una firma EdDSA richiederebbe 64 + 32 byte per il punto EC αG e lo scalare r. Tuttavia, l'RFC8032 assume che il punto αG sia compresso, riducendo lo spazio necessario a soli 32 + 32 byte. Qui includiamo anche la chiave pubblica K, per un totale di 32 + 32 + 32 byte.

2.5 Operatore Binario XOR

L'operatore binario XOR è uno strumento utile che apparirà nelle Sezioni 4.4 e 5.3. Prende due argomenti e restituisce vero se uno, ma non entrambi, è vero [21]. Ecco la relativa tabella di verità:

A	B	A XOR B
V	V	F
V	F	V
F	V	V
F	F	F

Nel contesto dell'informatica, XOR è equivalente alla somma bit a bit modulo 2. Ad esempio, l'XOR di due coppie di bit:

$$\begin{aligned} \text{XOR}(\{1,1\},\{1,0\}) &= \{1+1, 1+0\} \pmod 2 \\ &= \{0,1\} \end{aligned}$$

Anche queste combinazioni producono $\{0,1\}$: $\text{XOR}(\{1,0\},\{1,1\})$, $\text{XOR}(\{0,0\},\{0,1\})$ e $\text{XOR}(\{0,1\},\{0,0\})$. Per input XOR di b bit, ci sono $2^b - 1$ altre combinazioni di input che producono lo stesso output. Ciò significa che se $C = \text{XOR}(A,B)$ e $A \in_R \{0,...,2^b - 1\}$, un osservatore che conosce C non ottiene alcuna informazione su B.

Allo stesso tempo, chiunque conosca due degli elementi in $\{A,B,C\}$, con $C = \text{XOR}(A,B)$, può calcolare il terzo elemento, ad esempio $A = \text{XOR}(B,C)$. XOR indica se due elementi sono diversi o uguali, quindi conoscere C e B è sufficiente per determinare A. Un'attenta analisi della tabella di verità rivela questa caratteristica fondamentale.[30]

[30] Una interessante applicazione di XOR (non collegata a Monero) è lo scambio di due registri di bit senza un terzo registro. Il simbolo $\oplus$ è usato per indicare l'operazione XOR. Poiché $A \oplus A = 0$, dopo tre operazioni XOR tra i registri si ottiene: $\{A,B\} \to \{[A \oplus B], B\} \to \{[A \oplus B], B \oplus [A \oplus B]\} = \{[A \oplus B], A \oplus 0\} = \{[A \oplus B], A\} \to \{[A \oplus B] \oplus A, A\} = \{B, A\}$.

CAPITOLO 3

Firme Avanzate in Stile Schnorr

Una firma Schnorr di base utilizza una sola chiave. Tuttavia, è possibile sfruttare i suoi concetti fondamentali per creare una varietà di schemi di firma progressivamente più complessi. Uno di questi schemi, MLSAG, sarà di importanza centrale nel protocollo di transazione Monero.

3.1 Dimostrazione della Conoscenza di un Logaritmo Discreto su Basi Multiple

È spesso utile dimostrare che la stessa chiave privata è stata utilizzata per costruire chiavi pubbliche su diverse *basi*. Ad esempio, potremmo avere una chiave pubblica tradizionale kG, e un segreto condiviso Diffie-Hellman kR con la chiave pubblica di un'altra persona (si veda la Sezione 2.3.3), dove le chiavi *base* sono G e R. Come spiegato più avanti, è possibile fornire una prova inconfutabile sulla conoscenza del logaritmo discreto k in kG e in kR, *dimostrando* che k è lo stesso in entrambi i casi, il tutto senza rivelare k.

Dimostrazione Non Interattiva

Si supponga di avere una chiave privata k, e d chiavi base $\mathcal{J} = \{J_1, ..., J_d\}$. Le chiavi pubbliche corrispondenti sono $\mathcal{K} = \{K_1, ..., K_d\}$. Realizziamo una dimostrazione in stile Schnorr (si veda

la Sezione 2.3.4) su tutte le basi.[1] Si supponga inoltre l'esistenza di una funzione hash $\mathcal{H}_n$ che associa gli input del dominio ad interi da 0 a $l-1$.[2]

1. Generare un numero casuale $\alpha \in_R \mathbb{Z}_l$, e calcola, per ogni $i \in (1, ..., d)$, αJ_i.
2. Calcolare la sfida,
$$c = \mathcal{H}_n(\mathcal{J}, \mathcal{K}, [\alpha J_1], [\alpha J_2], ..., [\alpha J_d])$$
3. Definire la risposta $r = \alpha - c * k$.
4. Pubblicare la firma (c, r).

Verifica

Supponendo che il verificatore conosca $\mathcal{J}$ e $\mathcal{K}$, egli effettua i seguenti passaggi:

1. Calcolare la sfida:
$$c' = \mathcal{H}(\mathcal{J}, \mathcal{K}, [rJ_1 + c * K_1], [rJ_2 + c * K_2], ..., [rJ_d + c * K_d])$$
2. Se $c = c'$, allora il firmatario conosce sicuramente il logaritmo discreto su tutte le basi, ed è lo stesso in ciascun caso (come sempre, salvo probabilità trascurabili).

Perché Funziona

Se invece di d chiavi base ce ne fosse solo una, questa dimostrazione sarebbe chiaramente identica alla dimostrazione Schnorr originale (Sezione 2.3.4). È possibile considerare ogni chiave base singolarmente per rendersi conto che la dimostrazione multi-base è semplicemente un insieme di dimostrazioni Schnorr collegate tra loro. Inoltre, usare un'unica sfida e risposta per tutte queste dimostrazioni, comporta la condivisione dello stesso logaritmo discreto k. Per ottenere una singola risposta che funzioni per più chiavi, la sfida dovrebbe essere nota prima di definire un α per ciascuna chiave, ma questo non è possibile in quanto c è funzione di α!

3.2 Più Chiavi Private in una Dimostrazione

Proprio come per la dimostrazione multi-base, possiamo combinare più dimostrazioni Schnorr che utilizzano chiavi private diverse. In questo modo dimostriamo di conoscere tutte le chiavi private associate a un insieme di chiavi pubbliche, e riduciamo i requisiti di memoria generando una sola sfida per tutte le dimostrazioni.

[1] Anche se la chiamiamo 'dimostrazione', può essere resa trivialmente una firma includendo un messaggio $\mathfrak{m}$ nell'hash della sfida. La terminologia è usata in modo intercambiabile in questo contesto.

[2] In Monero, la funzione hash è $\mathcal{H}_n(x) = \text{sc_reduce32}(Keccak(x))$, dove *Keccak* è alla base di SHA3 e sc_reduce32() riduce il risultato a 256 bit nell'intervallo da 0 a $l-1$ (anche se in realtà dovrebbe essere da 1 a $l-1$).

Dimostrazione Non Interattiva

Si supponga di avere d chiavi private $k_1, ..., k_d$, e chiavi base $\mathcal{J} = \{J_1, ..., J_d\}$,[3] con chiavi pubbliche corrispondenti $\mathcal{K} = \{K_1, ..., K_d\}$. Per una dimostrazione tipo Schnorr per tutte le chiavi contemporaneamente, è necessario effettuare i seguenti passaggi:

1. Generare numeri casuali $\alpha_i \in_R \mathbb{Z}_l$ per tutti $i \in (1, ..., d)$, e calcolare tutti gli $\alpha_i J_i$.
2. Calcolare la sfida:
$$c = \mathcal{H}_n(\mathcal{J}, \mathcal{K}, [\alpha_1 J_1], [\alpha_2 J_2], ..., [\alpha_d J_d])$$
3. Definire ogni risposta $r_i = \alpha_i - c * k_i$.
4. Pubblicare la firma $(c, r_1, ..., r_d)$.

Verifica

Supponendo che il verificatore conosca $\mathcal{J}$ e $\mathcal{K}$, egli effettua i seguenti passaggi:

1. Calcolare la sfida:
$$c' = \mathcal{H}(\mathcal{J}, \mathcal{K}, [r_1 J_1 + c * K_1], [r_2 J_2 + c * K_2], ..., [r_d J_d + c * K_d])$$
2. Se $c = c'$, allora il firmatario conosce le chiavi private di tutte le chiavi pubbliche in $\mathcal{K}$ (salvo probabilità trascurabili).

3.3 Firme di Gruppo Anonime Spontanee (Spontaneous Anonymous Group, SAG)

Le firme di gruppo sono un metodo per dimostrare che un firmatario appartiene a un gruppo, senza necessariamente identificarlo. In origine (Chaum in [46]), gli schemi di firme di gruppo richiedevano che il sistema fosse impostato, e in alcuni casi gestito, da una persona fidata, al fine di prevenire firme illegittime e, in alcuni schemi, mediare nelle controversie. Questi schemi si basavano su un *segreto di gruppo*, il che non è auspicabile poiché crea un rischio di divulgazione che potrebbe compromettere l'anonimato. Inoltre, la necessità di coordinazione tra i membri del gruppo (cioè per l'impostazione e la gestione) non è scalabile oltre piccoli gruppi o ambienti aziendali.

Liu *et al.* hanno presentato uno schema più interessante in [83], basandosi sul lavoro di Rivest *et al.* in [118]. Gli autori hanno descritto un algoritmo di firma di gruppo chiamato LSAG, caratterizzato da tre proprietà: *anonimato, collegabilità* e *spontaneità*. In questa sezione verrà trattato il

[3] Non c'è motivo per cui $\mathcal{J}$ non possa contenere chiavi base duplicate, o che tutte le basi non possano essere uguali (ad esempio G). I duplicati sarebbero ridondanti per le dimostrazioni multi-base, ma qui stiamo trattando chiavi private diverse.

SAG, la versione non collegabile di LSAG, per maggiore chiarezza concettuale. Il concetto della collegabilità (linkability) verrà trattato nelle sezioni successive.

Gli schemi con anonimato e spontaneità sono solitamente chiamati *firme ad anello* (ring signatures). Nel contesto di Monero, queste firme permettono la realizzazione di transazioni non falsificabili e *ambigue* rispetto al firmatario, che rendono i flussi di valuta opachi e non tracciabili.

Firma

Le firme ad anello sono composte da un anello e da una firma. Ogni *anello* è un insieme di chiavi pubbliche, una delle quali appartiene al firmatario, mentre le altre non sono correlate. La *firma* viene generata con questo anello di chiavi, e chiunque la verifichi non sarà in grado di determinare quale membro dell'anello abbia firmato effettivamente.

Il nostro schema di firma tipo Schnorr descritto nella Sezione 2.3.5 può essere considerato una firma ad anello con una sola chiave. Possiamo arrivare a due chiavi generando, invece di definire subito r, un valore fittizio r' e creando una nuova sfida per definire r.

Sia $\mathfrak{m}$ il messaggio da firmare, $\mathcal{R} = \{K_1, K_2, ..., K_n\}$ un insieme di chiavi pubbliche distinte (anche detto gruppo o anello), e k_π la chiave privata del firmatario corrispondente alla sua chiave pubblica $K_\pi \in \mathcal{R}$, dove π è un indice segreto. Il firmatario dovrà dunque effettuare i seguenti passaggi:

1. Generare un numero casuale $\alpha \in_R \mathbb{Z}_l$ e risposte finte $r_i \in_R \mathbb{Z}_l$ per $i \in \{1, 2, ..., n\}$ escludendo però $i = \pi$.
2. Calcolare
$$c_{\pi+1} = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [\alpha G])$$
3. Per $i = \pi + 1, \pi + 2, ..., n, 1, 2, ..., \pi - 1$ calcolare, sostituendo $n + 1 \to 1$,
$$c_{i+1} = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_i G + c_i K_i])$$
4. Definire la risposta reale r_π tale che $\alpha = r_\pi + c_\pi k_\pi \pmod l$.

La firma ad anello prodotta conterrà dunque la firma $\sigma(\mathfrak{m}) = (c_1, r_1, ..., r_n)$ e l'anello $\mathcal{R}$.

Verifica

La verifica consiste nel dimostrare che $\sigma(\mathfrak{m})$ è una firma valida creata da una chiave privata corrispondente a una chiave pubblica in $\mathcal{R}$ (senza necessariamente sapere quale), ed è effettuata come segue:

1. Per $i = 1, 2, ..., n$ calcolare iterativamente, sostituendo $n + 1 \rightarrow 1$,
$$c'_{i+1} = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_i G + c_i K_i])$$
2. Se $c_1 = c'_1$ allora la firma è valida. Nota che c'_1 è l'ultimo termine calcolato.

In questo schema di verifica è necessario riservare spazio di memoria per gli $1 + n$ interi e le n chiavi pubbliche.

Perché Funziona

Possiamo convincerci informalmente del corretto funzionamento dell'algoritmo attraverso un esempio. Dato l'anello $R = \{K_1, K_2, K_3\}$ con $k_\pi = k_2$, per generare la firma è necessario effettuare i seguenti passaggi:

1. Generare numeri casuali: α, r_1, r_3
2. Inizializzare il ciclo della firma:
$$c_3 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [\alpha G])$$
3. Iterare:
$$c_1 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_3 G + c_3 K_3])$$
$$c_2 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_1 G + c_1 K_1])$$
4. Al termine del ciclo, calcolare la risposta: $r_2 = \alpha - c_2 k_2 \pmod{l}$

Infine è necessario sostituire α in c_3 per capire da dove proviene il termine "anello":
$$c_3 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [(r_2 + c_2 k_2)G])$$
$$c_3 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_2 G + c_2 K_2 \;])$$

In seguito, eseguire la verifica usando $\mathcal{R}$ e $\sigma(\mathfrak{m}) = (c_1, r_1, r_2, r_3)$:

1. Usare r_1 e c_1 per calcolare
$$c'_2 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_1 G + c_1 K_1])$$
2. Dal momento in cui abbiamo creato la firma, vediamo che $c'_2 = c_2$. Dunque, con r_2 e c'_2 calcolare
$$c'_3 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_2 G + c'_2 K_2])$$
3. È possibile verificare facilmente che $c'_3 = c_3$ sostituendo c_2 a c'_2. Usando r_3 e c'_3 otteniamo
$$c'_1 = \mathcal{H}_n(\mathcal{R}, \mathfrak{m}, [r_3 G + c'_3 K_3])$$

Nessuna sorpresa: $c'_1 = c_1$ se sostituiamo c'_3 con c_3.

3.4 Le Firme bLSAG

Gli schemi di firma ad anello discussi da questa sezione in poi presentano diverse caratteristiche che saranno utili per la creazione di transazioni confidenziali.[4] Si noti che sia l'*ambiguità del firmatario* sia l'*impossibilità di falsificazione* si applicano anche alle firme SAG.

Ambiguità del firmatario Un osservatore deve poter essere in grado di constatare l'appartenenza del firmatario al gruppo dell'anello, senza identificarlo (eccetto con probabilità trascurabile).[5] Monero utilizza questa proprietà per offuscare l'origine dei fondi in ogni transazione.

Collegabilità Se una chiave privata viene usata per firmare due messaggi diversi, allora i messaggi risulteranno collegati.[6] Questa proprietà sarà utile per affrontare e prevenire gli attacchi di doppia spesa (double spending attack) in Monero (eccetto con probabilità trascurabile).

Impossibilità di falsificazione Nessun attaccante può falsificare una firma eccetto con probabilità trascurabile.[7] Questa proprietà è usata per prevenire il furto di fondi Monero da parte di chi non possiede le chiavi private appropriate.

Nello schema di firma LSAG [83], il proprietario di una chiave privata può produrre una firma anonima non collegata per ogni anello[8]. In questa sezione presentiamo una versione migliorata dell'algoritmo LSAG in cui la collegabilità è indipendente dai membri fittizi dell'anello.[9]

Le modifiche apportate sono state sviluppate in [107] basandosi su una pubblicazione di Adam Back [32] riguardante l'algoritmo di firma ad anello CryptoNote [134] (precedentemente usato in Monero e ora deprecato; vedi Sezione 8.1.2), a sua volta ispirato dal lavoro di Fujisaki e Suzuki in [62].

[4] Ricordare che tutti gli schemi di firma robusti hanno modelli di sicurezza che contengono varie proprietà. Le proprietà menzionate qui sono probabilmente quelle più rilevanti per comprendere lo scopo delle firme ad anello di Monero, ma non costituiscono una panoramica completa sulle firme ad anello collegabili.

[5] L'anonimato per un'azione si esprime di solito in termini di un *insieme di anonimato*, ovvero *tutte le persone che potrebbero aver compiuto quell'azione*. Il più grande insieme di anonimato è *l'umanità*, e per Monero è la dimensione dell'anello, o ad esempio il cosiddetto *livello mixin* v più il firmatario reale. Il mixin indica quanti membri fittizi ha ciascuna firma ad anello. Se il mixin è $v = 4$ allora ci sono 5 firmatari possibili. Ampliare gli insiemi di anonimato rende progressivamente più difficile rintracciare i veri attori.

[6] La proprietà di collegabilità non si applica alle chiavi pubbliche non firmanti. Cioè, un membro dell'anello la cui chiave pubblica è stata usata in firme ad anello diverse non causerà collegamenti.

[7] Alcuni schemi di firme ad anello, incluso quello di Monero, sono robusti contro attacchi adattivi con messaggi scelti e attacchi adattivi con chiavi pubbliche scelte. Un attaccante che può ottenere firme legittime per messaggi scelti e corrispondenti a specifiche chiavi pubbliche in anelli a sua scelta non può scoprire come falsificare la firma di nemmeno un messaggio. Questo è chiamato *impossibilità di falsificazione esistenziale*; vedere [107] e [83].

[8] Nello schema LSAG la collegabilità si applica solo a firme che usano anelli con gli stessi membri e nello stesso ordine, cioè *lo stesso anello esatto*. In pratica significa "una firma anonima per ogni membro dell'anello per ogni anello." Le firme possono essere collegate anche se realizzate per messaggi diversi.

[9] LSAG è stato trattato nella prima edizione di questo rapporto. [30]

Firma

Come per SAG, sia $\mathfrak{m}$ il messaggio da firmare, $\mathcal{R} = \{K_1, K_2, ..., K_n\}$ un insieme di chiavi pubbliche distinte, e k_π la chiave privata del firmatario corrispondente alla sua chiave pubblica $K_\pi \in \mathcal{R}$, dove π è un indice segreto. Si assuma l'esistenza di una funzione di hash $\mathcal{H}_p$, che dato un input restituisce un punto della curva ellittica.[10],[11]

1. Calcolare l'immagine della chiave $\tilde{K} = k_\pi \mathcal{H}_p(K_\pi)$.[12]
2. Generare un numero casuale $\alpha \in_R \mathbb{Z}_l$ e numeri casuali $r_i \in_R \mathbb{Z}_l$ per $i \in \{1, 2, ..., n\}$ escluso $i = \pi$.
3. Calcolare
$$c_{\pi+1} = \mathcal{H}_n(\mathfrak{m}, [\alpha G], [\alpha \mathcal{H}_p(K_\pi)])$$
4. Per $i = \pi + 1, \pi + 2, ..., n, 1, 2, ..., \pi - 1$ calcolare, sostituendo $n + 1 \rightarrow 1$,
$$c_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_i G + c_i K_i], [r_i \mathcal{H}_p(K_i) + c_i \tilde{K}])$$
5. Definire $r_\pi = \alpha - c_\pi k_\pi \pmod{l}$.

La firma sarà $\sigma(\mathfrak{m}) = (c_1, r_1, ..., r_n)$, con immagine chiave $\tilde{K}$ e anello $\mathcal{R}$.

Verifica

La verifica consiste nel dimostrare che $\sigma(\mathfrak{m})$ è una firma valida creata da una chiave privata corrispondente a una chiave pubblica in $\mathcal{R}$, e si esegue nel modo seguente:

1. Verificare $l\tilde{K} \stackrel{?}{=} 0$.
2. Per $i = 1, 2, ..., n$ calcolare iterativamente, sostituendo $n + 1 \rightarrow 1$,
$$c'_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_i G + c_i K_i], [r_i \mathcal{H}_p(K_i) + c_i \tilde{K}])$$
3. Se $c_1 = c'_1$ allora la firma è valida.

[10] Non importa se i punti di $\mathcal{H}_p$ siano compressi o meno. Possono sempre essere decompressi.

[11] Monero utilizza una funzione di hash che restituisce direttamente punti della curva, invece di calcolare un intero che viene poi moltiplicato per G. $\mathcal{H}_p$ sarebbe compromessa se qualcuno trovasse un modo per ottenere n_x tale che $n_x G = \mathcal{H}_p(x)$. Una descrizione dell'algoritmo è in [106]. Secondo il whitepaper CryptoNote [134] la sua origine è in questo articolo: [131].

src/ringct/rctOps.cpp `hash_to_p3()`

[12] In Monero è importante usare la funzione hash-to-point per le immagini chiave invece di un altro punto base, così da evitare che la linearità porti al collegamento di firme create dallo stesso indirizzo (anche se per indirizzi monouso diversi). Vedi [134] a pagina 18.

In questo schema sono memorizzati $1+n$ interi, viene prodotta un'immagine chiave EC e utilizzate n chiavi pubbliche.

È necessario verificare che $l\tilde{K} \stackrel{?}{=} 0$ affinché sia possibile aggiungere un punto EC dal sottogruppo di ordine h (il cofattore) a $\tilde{K}$ e, se tutti i c_i sono multipli di h (cosa che potremmo ottenere tramite tentativi automatici con diversi valori di α e r_i), generare h firme valide scollegate usando lo stesso anello e la stessa chiave di firma.[13] Questo perché un punto EC moltiplicato per l'ordine del suo sottogruppo è zero.[14]

src/cryptonote_core/cryptonote_core.cpp `check_tx_inputs_keyimages_domain()`

Per chiarire, dato un punto K nel sottogruppo di ordine l, un punto K^h di ordine h, e un intero c divisibile per h:

$$\begin{aligned} c * (K + K^h) &= cK + cK^h \\ &= cK + 0 \end{aligned}$$

È possibile dimostrare la correttezza (cioè il funzionamento) in modo simile al metodo utilizzato per lo schema di firma SAG.

Questa descrizione tenta di essere fedele alla spiegazione originale del bLSAG, che non include $\mathcal{R}$ nella funzione hash che calcola c_i. Includere le chiavi nell'hash è noto come 'key prefixing'. Ricerche recenti [75] suggeriscono che potrebbe non essere necessario, anche se aggiungere il prefisso è prassi standard per schemi di firma simili (LSAG ad esempio usa il key prefixing).

Collegabilità

Date due firme valide che differiscono in qualche modo (ad esempio risposte false differenti, messaggi diversi, membri complessivi del ring differenti):

$$\begin{aligned} \sigma(\mathfrak{m}) &= (c_1, r_1, ..., r_n) \text{ con } \tilde{K}, \text{ e} \\ \sigma'(\mathfrak{m}') &= (c'_1, r'_1, ..., r'_{n'}) \text{ con } \tilde{K}', \end{aligned}$$

Se $\tilde{K} = \tilde{K}'$ allora banalmente entrambe le firme derivano dalla stessa chiave privata.

Sebbene un osservatore possa collegare σ e σ', non saprebbe necessariamente a quale K_i in $\mathcal{R}$ o $\mathcal{R}'$ corrisponda a meno che non ci fosse una sola chiave comune tra loro. Se ci fosse più di un membro comune del ring, la sua unica possibilità sarebbe risolvere il problema del logaritmo discreto o esaminare i ring in qualche modo (come imparare tutti i k_i con $i \neq \pi$, o imparare k_π).[15]

[13] Non ci preoccupiamo dei punti provenienti da altri sottogruppi perché l'output di $\mathcal{H}_n$ è limitato a $\mathbb{Z}_l$. Per un ordine EC $N = hl$, tutti i divisori di N (e quindi i possibili sottogruppi) sono o multipli di l (un primo) o divisori di h.

[14] Nella storia iniziale di Monero questo controllo non veniva effettuato. Fortunatamente, non è stato sfruttato prima che fosse implementata una correzione nell'aprile 2017 (versione 5 del protocollo) [60].

[15] LSAG, molto simile a bLSAG, è non falsificabile, cioè nessun attaccante può creare una firma valida senza conoscere la chiave privata. Se inventa un falso $\tilde{K}$ e avvia il calcolo della firma con $c_{\pi+1}$, allora, non conoscendo k_π, non può calcolare un numero $r_\pi = \alpha - c_\pi k_\pi$ che produca $[r_\pi G + c_\pi K_\pi] = \alpha G$. Un verificatore rifiuterebbe la firma. Liu *et al.* dimostrano che firme contraffatte in grado di passare la verifica sono estremamente improbabili [83].

3.5 Firme Multilivello MLSAG

Al fine di autorizzare correttamente una o più transazioni, occorre firmare con più chiavi private. In [107], Shen Noether *et al.* descrivono una generalizzazione multilivello dello schema di firma bLSAG applicabile su un insieme di $n \cdot m$ chiavi, overro l'insieme:

$$\mathcal{R} = \{K_{i,j}\} \quad \text{con} \quad i \in \{1,2,...,n\} \quad \text{e} \quad j \in \{1,2,...,m\}$$

Nel quale conosciamo le m chiavi private $\{k_{\pi,j}\}$ corrispondenti al sottoinsieme $\{K_{\pi,j}\}$ per qualche indice $i = \pi$. Tale algoritmo consente la generalizzazione della nozione di collegabilità (linkability):

Collegabilità Se una qualsiasi chiave privata $k_{\pi,j}$ è usata in 2 firme differenti, allora queste firme sono collegate.

Firma

1. Calcolare le immagini delle chiavi $\tilde{K}_j = k_{\pi,j}\mathcal{H}_p(K_{\pi,j})$ per ogni $j \in \{1,2,...,m\}$.
2. Generare numeri casuali $\alpha_j \in_R \mathbb{Z}_l$, e $r_{i,j} \in_R \mathbb{Z}_l$ per $i \in \{1,2,...,n\}$ (escluso $i = \pi$) e $j \in \{1,2,...,m\}$.
3. Calcolare[16]
$$c_{\pi+1} = \mathcal{H}_n(\mathfrak{m}, [\alpha_1 G], [\alpha_1 \mathcal{H}_p(K_{\pi,1})], ..., [\alpha_m G], [\alpha_m \mathcal{H}_p(K_{\pi,m})])$$
4. Per $i = \pi+1, \pi+2, ..., n, 1, 2, ..., \pi-1$ calcolare, sostituendo $n+1 \rightarrow 1$,
$$c_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_{i,1}G + c_i K_{i,1}], [r_{i,1}\mathcal{H}_p(K_{i,1}) + c_i \tilde{K}_1], ..., [r_{i,m}G + c_i K_{i,m}], [r_{i,m}\mathcal{H}_p(K_{i,m}) + c_i \tilde{K}_m])$$
5. Definire tutti i $r_{\pi,j} = \alpha_j - c_\pi k_{\pi,j} \pmod{l}$.

La firma prodotta è $\sigma(\mathfrak{m}) = (c_1, r_{1,1}, ..., r_{1,m}, ..., r_{n,1}, ..., r_{n,m})$, con immagini delle chiavi $(\tilde{K}_1, ..., \tilde{K}_m)$.

Verifica

Per verificare la firma è necessario effettuare i seguenti passaggi:

1. Per ogni $j \in \{1,...,m\}$ controllare se $l\tilde{K}_j \stackrel{?}{=} 0$.
2. Per $i = 1, ..., n$ calcolare, sostituendo $n+1 \rightarrow 1$,
$$c'_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_{i,1}G + c_i K_{i,1}], [r_{i,1}\mathcal{H}_p(K_{i,1}) + c_i \tilde{K}_1], ..., [r_{i,m}G + c_i K_{i,m}], [r_{i,m}\mathcal{H}_p(K_{i,m}) + c_i \tilde{K}_m])$$
3. Se $c_1 = c'_1$ allora la firma è valida.

[16] Monero MLSAG usa il key prefixing. Ogni challenge contiene esplicitamente le chiavi pubbliche così (aggiungendo i termini K assenti in bLSAG; le immagini delle chiavi sono incluse nel messaggio firmato):

$$c_{\pi+1} = \mathcal{H}_n(\mathfrak{m}, K_{\pi,1}, [\alpha_1 G], [\alpha_1 \mathcal{H}_p(K_{\pi,1})], ..., K_{\pi,m}, [\alpha_m G], [\alpha_m \mathcal{H}_p(K_{\pi,m})])$$

src/ringct/rctSigs.cpp `MLSAG_Gen()`

Perché Funziona

Proprio come nell'algoritmo SAG, è possibile notare che:

- Se $i \neq \pi$, allora i valori c'_{i+1} sono calcolati come descritto nell'algoritmo di firma.
- Se $i = \pi$ allora, dato che $r_{\pi,j} = \alpha_j - c_\pi k_{\pi,j}$ termina il ciclo,

$$r_{\pi,j}G + c_\pi K_{\pi,j} = (\alpha_j - c_\pi k_{\pi,j})G + c_\pi K_{\pi,j} = \alpha_j G$$

e

$$r_{\pi,j}\mathcal{H}_p(K_{\pi,j}) + c_\pi \tilde{K}_j = (\alpha_j - c_\pi k_{\pi,j})\mathcal{H}_p(K_{\pi,j}) + c_\pi \tilde{K}_j = \alpha_j \mathcal{H}_p(K_{\pi,j})$$

In altre parole, vale anche che $c'_{\pi+1} = c_{\pi+1}$.

Collegabilità

Se una chiave privata $k_{\pi,j}$ viene riutilizzata per creare una firma, l'immagine della chiave corrispondente $\tilde{K}_j$ fornita nella firma lo rivelerà. Questa osservazione coincide con la definizione generalizzata di linkabilità fornita precedentemente.[17]

Requisiti di Memoria

In questo schema sono memorizzati $1 + m \cdot n$ interi, prodotte m immagini di chiavi EC, e usate $m \cdot n$ chiavi pubbliche.

3.6 Firme Concise CLSAG

CLSAG [65][18] è una via di mezzo tra bLSAG e MLSAG. Supponiamo di avere una chiave *primaria* e ad essa siano associate diverse chiavi *ausiliarie*. È importante dimostrare la conoscenza di tutte le chiavi private, ma la collegabilità si applica solo alla primaria. Questa restrizione consente firme più piccole e veloci rispetto a MLSAG.

Come per MLSAG, si dispone di un insieme di $n \cdot m$ chiavi (dove n è la dimensione dell'anello, m è il numero di chiavi da usare nella firma), e le chiavi primarie si trovano all'indice 1. In altre parole, ci sono n chiavi primarie, e la πesima di queste, insieme alle sue ausiliarie, verrà usata nella firma.

[17] Come per bLSAG, le firme MLSAG collegate non indicano quale chiave pubblica sia stata usata per firmare. Tuttavia, se negli anelli dei sotto-cicli dell'immagine di chiave collegata c'è una sola chiave in comune, il colpevole è evidente. Se il colpevole viene identificato, tutti gli altri membri firmatari di entrambe le firme sono rivelati poiché condividono gli indici del colpevole.

[18] L'articolo su cui si basa questa sezione è un pre-print in fase di finalizzazione per revisione esterna. CLSAG è promettente come sostituto di MLSAG nelle future versioni del protocollo, ma non è stato ancora implementato e potrebbe non esserlo in futuro.

$$\mathcal{R} = \{K_{i,j}\} \quad \text{con} \quad i \in \{1, 2, ..., n\} \quad \text{e} \quad j \in \{1, 2, ..., m\}$$

Conosciamo le chiavi private $\{k_{\pi,j}\}$ corrispondenti al sottoinsieme $\{K_{\pi,j}\}$ per un certo indice $i = \pi$.

Firma

1. Calcolare le immagini delle chiavi $\tilde{K}_j = k_{\pi,j}\mathcal{H}_p(K_{\pi,1})$ per ogni $j \in \{1, 2, ..., m\}$. Si noti che la chiave base è sempre la stessa, quindi le immagini delle chiavi con $j > 1$ sono dette 'immagini delle chiavi ausiliarie'. Per semplicità verranno indicate tutte con $\tilde{K}_j$.

2. Generare numeri casuali $\alpha \in_R \mathbb{Z}_l$ e $r_i \in_R \mathbb{Z}_l$ per $i \in \{1, 2, ..., n\}$ (eccetto $i = \pi$).

3. Calcolare le chiavi pubbliche aggregate W_i per $i \in \{1, 2, ..., n\}$, e l'immagine della chiave aggregata $\tilde{W}$[19]

$$W_i = \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \tilde{K}_1, ..., \tilde{K}_m) * K_{i,j}$$

$$\tilde{W} = \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \tilde{K}_1, ..., \tilde{K}_m) * \tilde{K}_j$$

dove $w_\pi = \sum_j \mathcal{H}_n(T_j, ...) * k_{\pi,j}$ è la chiave privata aggregata.

4. Calcolare

$$c_{\pi+1} = \mathcal{H}_n(T_c, \mathcal{R}, \mathfrak{m}, [\alpha G], [\alpha \mathcal{H}_p(K_{\pi,1})])$$

5. Per $i = \pi + 1, \pi + 2, ..., n, 1, 2, ..., \pi - 1$ calcolare, sostituendo $n + 1 \to 1$,

$$c_{i+1} = \mathcal{H}_n(T_c, \mathcal{R}, \mathfrak{m}, [r_i G + c_i W_i], [r_i \mathcal{H}_p(K_{i,1}) + c_i \tilde{W}])$$

6. Definire $r_\pi = \alpha - c_\pi w_\pi \pmod{l}$.

La firma prodotta è $\sigma(\mathfrak{m}) = (c_1, r_1, ..., r_n)$, con immagine della chiave primaria $\tilde{K}_1$ e immagini ausiliarie $(\tilde{K}_2, ..., \tilde{K}_m)$.

Verifica

Per la verifica della firma è necessario effettuare i seguenti passaggi:

1. Per ogni $j \in \{1, ..., m\}$ controllare $l\tilde{K}_j \stackrel{?}{=} 0$.[20]

[19] L'articolo CLSAG suggerisce di usare funzioni di hash diverse per la separazione del dominio, che modelliamo anteponendo a ogni hash un tag [65], ad esempio T_1 = "CLSAG_1", T_c = "CLSAG_c", ecc. Le funzioni di hash separate per dominio producono output diversi anche con gli stessi input. Utilizziamo anche la prefissazione delle chiavi qui (inclusa $\mathcal{R}$, che contiene tutte le chiavi, nell'hash). La separazione del dominio è una nuova politica per lo sviluppo di Monero, che sarà probabilmente applicata a tutte le future applicazioni delle funzioni hash (versione 13+). Gli usi storici probabilmente resteranno invariati.

[20] In Monero si verifica solo $l * \tilde{K}_1 \stackrel{?}{=} 0$ per l'immagine della chiave primaria. Le chiavi ausiliarie sono memorizzate come $(1/8) * \tilde{K}_j$, e durante la verifica moltiplicate per 8 (vedi Sezione 2.3.1), metodo più efficiente. La discrepanza è una scelta implementativa, dato che le immagini di chiavi linkabili sono molto importanti e non devono essere manipolate aggressivamente, mentre l'altro metodo è stato usato nelle versioni precedenti del protocollo.

2. Calcolare le chiavi pubbliche aggregate W_i per $i \in \{1, ..., n\}$ e l'immagine aggregata $\tilde{W}$

$$W_i = \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \tilde{K}_1, ..., \tilde{K}_m) * K_{i,j}$$

$$\tilde{W} = \sum_{j=1}^{m} \mathcal{H}_n(T_j, \mathcal{R}, \tilde{K}_1, ..., \tilde{K}_m) * \tilde{K}_j$$

3. Per $i = 1, ..., n$ calcolare, sostituendo $n + 1 \to 1$,

$$c_{i+1} = \mathcal{H}_n(T_c, \mathcal{R}, \mathfrak{m}, [r_i G + c_i W_i], [r_i \mathcal{H}_p(K_{i,1}) + c_i \tilde{W}])$$

4. Se $c_1 = c'_1$ allora la firma è valida.

Perché Funziona

Il pericolo principale nelle firme concise come questa è l'annullamento delle chiavi, che si verifica quando le immagini delle chiavi riportate non sono legittime, ma sommano comunque a un valore aggregato legittimo. Qui entrano in gioco i coefficienti di aggregazione $\mathcal{H}_n(T_j, \mathcal{R}, \tilde{K}_1, ..., \tilde{K}_m)$, che fissano ogni chiave al valore atteso. Lasciamo al lettore l'esercizio di seguire le ripercussioni circolari del falsificare un'immagine chiave (magari iniziando con il supporre che quei coefficienti non esistano). Le immagini delle chiavi ausiliarie sono un artefatto della prova della legittimità dell'immagine primaria, dato che la chiave privata aggregata w_π, che contiene tutte le chiavi private, è applicata al punto base $\mathcal{H}_p(K_{\pi,1})$.

Collegabilità

Se una chiave privata $k_{\pi,1}$ viene riutilizzata per creare una qualsiasi firma, l'immagine della chiave primaria corrispondente $\tilde{K}_1$ fornita nella firma lo rivelerà. Le immagini delle chiavi ausiliarie sono ignorate, poiché esistono solo per facilitare la parte 'Concisa' del CLSAG.

Requisiti di Memoria

Sono memorizzati $(1+n)$ interi, prodotte m immagini delle chiavi e utilizzate $m \cdot n$ chiavi pubbliche.

CAPITOLO 4

Indirizzi Monero

In generale, la proprietà di un quantitativo di valuta digitale su blockchain è conferita dai cosiddetti 'indirizzi'. Gli indirizzi contengono le monete (output) ricevute e solo il proprietario ne detiene la capacità di spesa.[1]

Precisamente, un indirizzo detiene gli output ricevuti attraverso transazioni, che sono come delle note che conferiscono al proprietario dell'indirizzo di destinazione il diritto di spendere un certo ammontare di denaro. Ad esempio, una certa nota potrebbe indicare che "l'indirizzo C detiene 5.3 XMR".

Per spendere un output detenuto, il proprietario dell'indirizzo può specificarlo come input di una nuova transazione. Questa nuova transazione produrrà dei nuovi output che saranno assegnati ad altri indirizzi (o agli indirizzi del proprietario, se desiderato). La somma degli input di una transazione è uguale alla somma dei suoi output, inoltre, gli output possono essere spesi soltanto una volta. Carol, proprietaria dell'indirizzo C, può specificare l'output ricevuto in una nuova transazione (ad esempio "In questa transazione vorrei spendere quel determinato output ricevuto.") ed aggiungere una nota la quale indica "l'Indirizzo B adesso detiene 5.3 XMR".

Il saldo di un indirizzo equivale alla somma di tutti gli importi contenuti negli output non spesi detenuti da quel particolare indirizzo.[2]

[1] La probabilità che un soggetto esterno sia in grado di spendere le monete non di sua proprietà è altamente trascurabile.

[2] Le applicazioni software conosciute come portafogli (wallet) sono utilizzati per organizzare gli output detenuti da uno o più indirizzi, gestire le chiavi private per l'autorizzazione di nuove transazioni e di inviarle al network per la verifica e l'inclusione nella blockchain.

Nel Capitolo 5 vengono trattate le modalità di offuscamento degli importi, nel Capitolo 6 la struttura delle transazioni (che include come dimostrare che si sta spendendo un output posseduto e precedentemente non speso, senza nemmeno rivelare quale output venga effettivamente speso), ed infine nel Capitolo 7 il processo di estrazione (mining) della moneta ed il ruolo degli osservatori esterni.

4.1 Chiavi Utente

A differenza di Bitcoin, gli utenti Monero dispongono di due set di chiavi private e pubbliche, (k^v, K^v) e (k^s, K^s). Nella Sezione 2.3.2 vengono trattate le modalità di creazione di questi set di chiavi.

In generale, l'*indirizzo* di un utente coincide con la coppia di chiavi pubbliche (K^v, K^s). Le chiavi private invece, corrispondono alla coppia (k^v, k^s).[3]

src/common/base58.cpp `encode_addr()`

L'utilizzo di due set di chiavi consente la segregazione delle funzioni (separazione dei compiti). La motivazione diventerà chiara più avanti in questo capitolo, ma per il momento ci limitiamo a definire la chiave privata k^v come *chiave di visualizzazione* (view key), e k^s come *chiave di spesa* (spend key). Una persona può usare la propria chiave di visualizzazione per determinare se un output appartiene al proprio indirizzo, mentre la chiave di spesa permette di spendere quel determinato output in una transazione (e, retroattivamente, capire se è stato speso).[4]

4.2 Indirizzi Monouso (One-time)

Un utente Monero, al fine di ricevere denaro, può distribuire i suoi indirizzi ad altri utenti, i quali potranno inviare output attraverso transazioni.

L'indirizzo di destinazione non è mai utilizzato direttamente in una transazione.[5] Invece, viene attuato uno scambio in stile Diffie-Hellman, creando un unico *indirizzo monouso univoco (one-time)*

[3] Per comunicare l'indirizzo ad un altro utente, è di comune uso codificarlo in base58, uno schema di codifica binary-to-text concepito inizialmente per Bitcoin [25]. Vedi [5] per maggiori dettagli.

[4] È di comune uso indicare la chiave di visualizzazione k^v come risultato dell'operazione $\mathcal{H}_n(k^s)$. Questo significa che una persona deve conservare solo la propria chiave di spesa k^s per accedere (visualizzare e spendere) a tutti gli output di cui è proprietaria (spesi e non). La chiave di spesa è solitamente rappresentata come una serie di 25 parole (dove la 25ª parola è un checksum). Altri metodi, meno comuni, includono: generare k^v e k^s come numeri casuali indipendenti, oppure generare una lista mnemonica casuale composta da 12 parole a, in cui

$$k^s = \texttt{sc_reduce32}(Keccak(a)) \quad \text{e} \quad k^v = \texttt{sc_reduce32}(Keccak(Keccak(a))).$$

[5]

src/wallet/api/wallet.cpp `create()` wallet2.cpp `generate()` `get_seed()`

[5] Il metodo qui descritto non è imposto dal protocollo, ma solo dagli standard di implementazione dei portafogli Monero. Ciò significa che un portafoglio alternativo potrebbe seguire lo stile di Bitcoin, dove gli indirizzi dei destinatari sono inclusi direttamente nei dati delle transazioni. Un portafoglio di questo tipo, chiaramente non conforme, produrrebbe transazioni con output inutilizzabili da altri portafogli e ogni indirizzo in stile Bitcoin potrebbe essere utilizzato una sola volta a causa dell'unicità dell'immagine della chiave.

per ogni output della transazione da inviare all'utente. In questa maniera anche gli osservatori esterni che conoscono tutti gli indirizzi dell'utente non saranno in grado di utilizzarli per identificare quali output di transazioni appartengono ad essi.[6]

Iniziamo con una transazione fittizia molto semplice, contenente esattamente un solo output — un pagamento di importo '0' da Alice a Bob.

Bob possiede le chiavi private e pubbliche (k^v_B, k^s_B) e (K^v_B, K^s_B), mentre Alice conosce le chiavi pubbliche (dunque il suo indirizzo) di Bob. La transazione tra Alice e Bob potrebbe svolgersi come segue [134]:

1. Alice genera un numero casuale $r \in_R \mathbb{Z}_l$, e calcola l'indirizzo one-time[7]
$$K^o = \mathcal{H}_n(rK^v_B)G + K^s_B$$
src/crypto/crypto.cpp `derive_public_key()`
2. Alice imposta K^o come destinatario del pagamento, aggiunge l'importo dell'output '0' e il valore rG ai dati della transazione, e la invia alla rete.
3. Bob riceve i dati e vede i valori rG e K^o. Dunque calcola $k^v_B rG = rK^v_B$. Ed in seguito $K'^s_B = K^o - \mathcal{H}_n(rK^v_B)G$. Se Bob verifica che $K'^s_B = K^s_B$, significa che l'output è indirizzato a lui.[8]
src/crypto/crypto.cpp `derive_subaddress_public_key()`
La chiave privata k^v_B è detta chiave di visualizzazione (view key) perché chiunque la possieda (insieme alla chiave pubblica di spesa K^s_B di Bob) può calcolare K'^s_B per ogni output di transazione nella blockchain (registro delle transazioni), e vedere quali di essi appartengono a Bob.
4. Le chiavi one-time per l'output sono:
$$K^o = \mathcal{H}_n(rK^v_B)G + K^s_B = (\mathcal{H}_n(rK^v_B) + k^s_B)G$$
$$k^o = \mathcal{H}_n(rK^v_B) + k^s_B$$

Per spendere il suo output da '0 XMR' [sic] in una nuova transazione, tutto ciò che Bob deve fare è dimostrare di possederlo firmando un messaggio con la chiave one-time K^o. La chiave privata k^s_B è detta chiave privata di spesa (private spend key) poiché è necessaria per dimostrare la proprietà dell'output, mentre k^v_B è la chiave privata di visualizzazione (view key) poiché può essere usata per trovare gli output spendibili da Bob.

[6] Eccetto con trascurabile probabilità.

[7] In Monero, ogni istanza (incluso quando viene usato in altre parti della transazione) di rk^vG viene moltiplicata per il cofattore 8, quindi in questo caso Alice calcola $8 * rK^v_B$ e Bob calcola $8 * k^v_B rG$. Per quanto ne sappiamo, questo non serve a nulla (ma è *comunque* una regola da seguire). Moltiplicare per il cofattore garantisce che il punto risultante appartenga al sottogruppo di G, ma se R e K^v non condividono un sottogruppo all'inizio, allora i logaritmi discreti r e k^v non possono comunque essere usati per creare un segreto condiviso.
src/crypto/crypto.cpp `generate_key_derivation()`

[8] K'^s_B è calcolato con `derive_subaddress_public_key()` perché le chiavi di spesa di un indirizzo normale sono memorizzate all'indice 0 nella tabella di lookup delle chiavi di spesa, mentre gli indirizzi secondari si trovano agli indici 1, 2... Questo avrà senso a breve, vedi Sezione 4.3.

Come sarà chiaro nel Capitolo 6, senza k^o Alice non può calcolare l'immagine chiave (key image) dell'output, quindi non potrà mai sapere con certezza se Bob spende l'output che gli ha inviato.[9]

Bob potrebbe fornire la sua view key a una terza parte. Questa terza parte potrebbe essere un custode di fiducia, un revisore, un'autorità fiscale, ecc. Qualcuno a cui può essere concesso l'accesso in sola lettura alla cronologia delle transazioni dell'utente, senza altri privilegi. Questa terza parte sarebbe anche in grado di decifrare l'importo dell'output (come sarà spiegato nella Sezione 5.3). Vedi il Capitolo 8 per consultare altri modi in cui Bob può fornire informazioni sulla sua cronologia delle transazioni.

4.2.1 Transazioni Multi-Output

La maggior parte delle transazioni conterrà più di un singolo output. Ciò avviene anche solo per restituire il 'resto' al mittente.[10] [11]

I mittenti delle transazioni Monero di solito generano un valore casuale r. Il punto sulla curva rG è comunemente noto come *chiave pubblica della transazione* (tx public key) ed è pubblicato insieme agli altri dati della transazione sulla blockchain.

Per garantire che tutti gli indirizzi one-time in una transazione con p output siano differenti, anche nei casi in cui lo stesso destinatario venga usato due volte, Monero utilizza un indice di output. Ogni output di una transazione ha un indice $t \in \{0, ..., p-1\}$. Aggiungendo questo valore al segreto condiviso prima di calcolare l'hash, si può garantire che gli indirizzi one-time risultanti siano univoci:

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_tx_with_tx_key()`

$$K_t^o = \mathcal{H}_n(rK_t^v, t)G + K_t^s = (\mathcal{H}_n(rK_t^v, t) + k_t^s)G$$
$$k_t^o = \mathcal{H}_n(rK_t^v, t) + k_t^s$$

src/crypto/crypto.cpp `derive_public_key()`

[9] Supponiamo che Alice produca due transazioni, ciascuna contenente lo stesso indirizzo one-time K^o che Bob può spendere. Poiché K^o dipende solo da r e K_B^v, non c'è motivo per cui non possa farlo. Bob può spendere solo uno di quegli output, poiché ogni indirizzo one-time ha una sola immagine di chiave. Se non fa attenzione, Alice potrebbe ingannarlo. Potrebbe creare la transazione 1 con molti soldi per Bob, e poi la transazione 2 con una piccola somma. Se lui spende i fondi della 2, non potrà mai più spendere quelli della 1. In effetti, nessuno potrebbe spendere i fondi della 1, 'bruciandoli' di fatto. I wallet Monero sono progettati per ignorare l'importo più piccolo in questo scenario.

[10] In realtà, a partire dal protocollo v12, sono *richiesti* due output per ogni transazione (non miner), anche se ciò comporta che uno degli output abbia importo 0 (`HF_VERSION_MIN_2_OUTPUTS`). Questo migliora l'indistinguibilità delle transazioni mescolando i casi con un solo output con le più comuni transazioni a due output. Prima della v12, il wallet principale già creava output di valore nullo. L'implementazione principale invia gli output con importo 0 a un indirizzo casuale.

src/wallet/wallet2.cpp `transfer_selected_rct()`

[11] Dopo l'introduzione dei Bulletproof nel protocollo v8, ogni transazione è stata limitata ad un massimo di 16 output (`BULLETPROOF_MAX_OUTPUTS`). In precedenza non esisteva alcun limite, se non un vincolo sulla dimensione della transazione (in byte).

4.3 Sottoindirizzi

Gli utenti Monero possono generare sottoindirizzi (subaddress) da ciascun indirizzo standard [104]. I fondi inviati a un sottoindirizzo possono essere visualizzati e spesi utilizzando le chiavi di visualizzazione e di spesa dell'indirizzo principale. Per analogia: un conto bancario online può avere più saldi corrispondenti a carte di credito e depositi, ma sono tutti accessibili e spendibili da un singolo gestore: il titolare del conto.

I sottoindirizzi sono comodi per ricevere fondi nello stesso portafoglio quando un utente non vuole collegare tra loro le proprie attività pubblicando o usando lo stesso indirizzo. Come vedremo, la maggior parte degli osservatori dovrebbe risolvere il problema del logaritmo discreto (DLP) per determinare se un dato sottoindirizzo è derivato da un particolare indirizzo [104].[12]

Sono utili anche per effettuare una distinzione degli output ricevuti. Ad esempio, se Alice vuole comprare una mela da Bob, Bob potrebbe scrivere una ricevuta per descrivere l'acquisto e creare un sottoindirizzo per quella ricevuta, poi chiedere ad Alice di usare quel sottoindirizzo quando gli invia il denaro. In questo modo, Bob può associare il denaro ricevuto alla mela venduta. Esploreremo un altro modo per distinguere e organizzare gli output ricevuti nella sezione successiva.

Bob genera il suo i^{mo} sottoindirizzo ($i = 1, 2, ...$), a partire dal suo indirizzo principale (anche detto primary address), come coppia di chiavi pubbliche $(K^{v,i}, K^{s,i})$:[13]

src/device/device_default.cpp `get_subaddress_secret_key()`

$$
\begin{aligned}
K^{s,i} &= K^s + \mathcal{H}_n(k^v, i)G \\
K^{v,i} &= k^v K^{s,i}
\end{aligned}
$$

Dunque

$$
\begin{aligned}
K^{v,i} &= k^v(k^s + \mathcal{H}_n(k^v, i))G \\
K^{s,i} &= \quad (k^s + \mathcal{H}_n(k^v, i))G
\end{aligned}
$$

4.3.1 Trasferimento Fondi ad un Sottoindirizzo

Supponiamo che Alice voglia nuovamente inviare un ammontare pari a '0', ma questa volta al sottoindirizzo $(K^{v,1}_B, K^{s,1}_B)$ di Bob.

1. Alice genera un numero casuale $r \in_R \mathbb{Z}_l$, e calcola l'indirizzo one-time

$$K^o = \mathcal{H}_n(rK^{v,1}_B, 0)G + K^{s,1}_B$$

[12] Prima dei sottoindirizzi (introdotti con l'aggiornamento software corrispondente alla versione 7 del protocollo [74]), gli utenti Monero potevano semplicemente generare molti indirizzi normali. Per visualizzare il saldo di ciascun indirizzo era necessaria una scansione separata della blockchain. Questo era molto inefficiente. Con i sottoindirizzi, gli utenti mantengono una tabella di ricerca delle chiavi di spesa (hashed), quindi una singola scansione della blockchain richiede lo stesso tempo per 1 sottoindirizzo o per 10.000 sottoindirizzi.

[13] Si scopre che l'hash del sottoindirizzo è separato per dominio, quindi in realtà è $\mathcal{H}_n(T_{sub}, k^v, i)$ dove $T_{sub} =$"SubAddr". Omettiamo T_{sub} in tutto il documento per brevità.

2. Alice imposta K^o come indirizzo del pagamento, aggiunge l'output di ammontare '0' e il valore $rK_B^{s,1}$ ai dati della transazione, ed infine invia la transazione al network.

3. Bob riceve i dati e vede i valori $rK_B^{s,1}$ e K^o. Può calcolare $k_B^v rK_B^{s,1} = rK_B^{v,1}$. Può quindi calcolare $K_B'^s = K^o - \mathcal{H}_n(rK_B^{v,1}, 0)G$. Quando vede che $K_B'^s = K_B^{s,1}$, sa che la transazione è indirizzata a lui.[14]

   Bob ha bisogno solo della sua chiave privata di visualizzazione k_B^v e della chiave pubblica di spesa del sottoindirizzo $K_B^{s,1}$ per trovare gli output di transazione inviati al suo sottoindirizzo.

src/crypto/crypto.cpp `derive_subaddress_public_key()`

4. Le chiavi one-time per l'output sono:

$$K^o = \mathcal{H}_n(rK_B^{v,1}, 0)G + K_B^{s,1} = (\mathcal{H}_n(rK_B^{v,1}, 0) + k_B^{s,1})G$$
$$k^o = \mathcal{H}_n(rK_B^{v,1}, 0) + k_B^{s,1}$$

Ora, la chiave pubblica della transazione di Alice è insolita per Bob ($rK_B^{s,1}$ invece di rG). Se crea una transazione con p output di cui almeno uno destinato a un sottoindirizzo, Alice dovrà generare una chiave pubblica della transazione unica per ogni output $t \in \{0, ..., p-1\}$. In altre parole, se Alice sta inviando al sottoindirizzo di Bob $(K_B^{v,1}, K_B^{s,1})$ e all'indirizzo di Carol (K_C^v, K_C^s), sarà necessario specificare due chiavi pubbliche della transazione $\{r_1 K_B^{s,1}, r_2 G\}$ nei dati della transazione.[15],[16]

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_tx_with_tx_key()`

[14] Un attaccante avanzato potrebbe riuscire a collegare sottoindirizzi [53] (noto anche come attacco Janus). Con sottoindirizzi (uno dei quali può essere un indirizzo normale) K_B^1 & K_B^2 che l'attaccante pensa siano correlati, crea un output di transazione con $K^o = \mathcal{H}_n(rK_B^{v,2}, 0)G + K_B^{s,1}$ e include la chiave pubblica della transazione $rK_B^{s,2}$. Bob calcola $rK_B^{v,2}$ per trovare $K_B'^{s,1}$ ma non ha modo di sapere che è stata usata la chiave del suo *altro* (sotto)indirizzo! Se dice all'attaccante di aver ricevuto fondi su K_B^1, l'attaccante saprà che K_B^2 è un sottoindirizzo correlato (o indirizzo normale). Poiché i sottoindirizzi sono fuori dallo scopo del protocollo, le mitigazioni sono a carico degli implementatori dei wallet. Nessun wallet noto lo fa, e ogni mitigazione funzionerebbe solo per wallet conformi. Le mitigazioni possibili includono: non informare l'attaccante dei fondi ricevuti, includere la chiave privata di transazione cifrata r nei dati della transazione, una firma Schnorr sul segreto condiviso usando $K^{s,1}$ come punto base invece di G, includere rG nei dati della transazione e verificare il segreto condiviso con $rK^{s,1} \stackrel{?}{=} (k^s + \mathcal{H}_n(k^v, 1)) * rG$ (richiede la chiave privata di spesa). Anche gli output ricevuti da un indirizzo normale devono essere verificati. Vedi [100] per una discussione sull'ultima mitigazione elencata.

[15] I sottoindirizzi di Monero sono prefissati con '8', separandoli dagli indirizzi normali che sono prefissati con '4'. Questo aiuta il mittente a scegliere la procedura corretta durante la costruzione della transazione.

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_account_address_as_str()`

[16] Ci sono alcune complessità sull'uso di chiavi pubbliche della transazione aggiuntive (vedi il percorso del codice `transfer_selected_rct()` → `construct_tx_and_get_tx_key()` → `construct_tx_with_tx_key()` → `generate_output_ephemeral_keys()` e `classify_addresses()`) relative agli output di resto dove l'autore della transazione conosce la chiave di visualizzazione del destinatario (dato che è sé stesso; anche il caso degli output di resto fittizi, creati quando è necessario un output di importo zero, poiché gli autori generano quegli indirizzi). Quando ci sono almeno due output non di resto e almeno uno dei destinatari è un sottoindirizzo, si procede nel modo normale spiegato sopra (un bug attuale nell'implementazione core aggiunge una chiave pubblica della transazione extra ai dati della transazione oltre a quelle aggiuntive, che non viene usata per nulla). Se solo l'output di resto è inviato ad un sottoindirizzo, oppure c'è un solo output non di resto ed è inviato ad un sottoindirizzo, allora si crea una sola chiave pubblica della transazione. Nel primo caso, la chiave pubblica della transazione è rG e la chiave one-time del resto è (indice sottoindirizzo 1, usando il pedice c per indicare le chiavi di resto) $K^o = \mathcal{H}_n(k_c^v rG, t)G + K_c^{s,1}$ usando la chiave di visualizzazione e la chiave di spesa del sottoindirizzo. Nel secondo caso, la chiave pubblica della transazione $rK^{v,1}$ si basa sulla chiave di visualizzazione del sottoindirizzo, e la chiave one-time del resto è $K^o = \mathcal{H}_n(k_c^v * rK^{v,1}, t)G + K_c^s$. Questi dettagli aiutano a mescolare una porzione delle transazioni ai sottoindirizzi fra le più comuni transazioni agli indirizzi normali, e le transazioni a 2 output che compongono circa il 95% del volume di transazioni a oggi.

4.4 Indirizzi Integrati

Per distinguere gli output che riceve, un destinatario può richiedere al mittente di includere un *ID di pagamento (payment ID)* nei dati della transazione.[17] Ad esempio, se Alice vuole comprare una mela da Bob, Bob potrebbe scrivere una ricevuta che descrive l'acquisto e chiedere ad Alice di includere l'identificativo della ricevuta quando gli invia i soldi. In questo modo Bob può associare i soldi ricevuti alla mela che ha venduto.

In passato i mittenti potevano comunicare i payment ID in chiaro, ma includerli manualmente nelle transazioni risultava scomodo, e la trasmissione in chiaro rappresenta un rischio per la privacy dei destinatari, che potrebbero esporre involontariamente le loro attività.[18] Attualmente su Monero, i destinatari possono integrare gli ID di pagamento nei loro indirizzi, fornendo tali *indirizzi integrati*, contenenti (K^v, K^s, ID di pagamento), ai mittenti. Gli ID di pagamento possono tecnicamente essere integrati in qualsiasi tipo di indirizzo, inclusi indirizzi standard, sottoindirizzi e indirizzi multifirme.[19]

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_account_integrated_address_as_str()`

I mittenti che inviano output a indirizzi integrati possono codificare gli ID di pagamento usando il segreto condiviso rK_t^v e un'operazione XOR (ricordare la Sezione 2.5), che i destinatari possono poi decodificare con la chiave pubblica della transazione appropriata e un'altra operazione XOR [9]. La codifica degli ID di pagamento in questo modo consente inoltre ai mittenti di dimostrare di aver creato specifici output di transazione (ad esempio per audit, rimborsi, ecc.).[20]

Codifica

Il mittente codifica l'ID di pagamento per includerlo nei dati della transazione[21]:

src/device/device_default.cpp `encrypt_payment_id()`

$$k_{\text{mask}} = \mathcal{H}_n(rK_t^v, \text{pid_tag})$$
$$k_{\text{payment ID}} = k_{\text{mask}} \rightarrow \text{ridotto alla lunghezza in bit dell'ID di pagamento}$$
$$\text{payment ID codificato} = \text{XOR}(k_{\text{payment ID}}, \text{ID di pagamento})$$

Includiamo pid_tag per garantire che k_{mask} sia diverso dalla componente $\mathcal{H}_n(rK_t^v, t)$ negli indirizzi di output one-time.[22]

[17] Attualmente, le implementazioni di Monero supportano solo un ID di pagamento per transazione, indipendentemente dal numero di output.

[18] Questi ID di pagamento in chiaro di lunghezza (256 bit) sono stati deprecati nella versione 0.15 dell'implementazione core del software (coincidente con la versione 12 del protocollo del novembre 2019). Sebbene altri wallet possano ancora supportarli e permetterne l'inclusione nei dati della transazione, il wallet core li ignorerà.

[19] Per quanto a conoscenza degli autori, gli indirizzi integrati sono stati implementati solo per indirizzi standard.

[20] Poiché un osservatore esterno può riconoscere la differenza tra transazioni con e senza ID di pagamento, il loro utilizzo è considerato causa di una minore uniformità nella storia delle transazioni Monero. Per questo motivo, dal protocollo v10 l'implementazione core aggiunge un ID di pagamento cifrato fittizio a tutte le transazioni con 2 output, la cui decodifica rivelerà una stringa di bit composta da solo zeri (si tratta di una buona pratica e non di una specifica del protocollo).

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_tx_with_tx_key()`

[21] Per convenzione, in Monero gli ID di pagamento per gli indirizzi integrati sono lunghi 64 bit.

[22] In Monero, pid_tag = `ENCRYPTED_PAYMENT_ID_TAIL` = 141. Ad esempio, in transazioni multi-input calcoliamo $\mathcal{H}_n(rK_t^v, t) \pmod{l}$ per garantire che si usi uno scalare minore dell'ordine del sottogruppo EC, ma dato che l è lungo 253 bit e gli ID di pagamento solo 64 bit, applicare il modulo per codificare gli ID di pagamento non avrebbe senso, dunque non viene fatto.

Decodifica

Qualsiasi destinatario per cui l'ID di pagamento è stato creato può trovarlo usando la propria chiave privata di visualizzazione e la chiave pubblica della transazione rG:[23]

src/device/device.hpp `decrypt_payment_id()`

$$
\begin{aligned}
k_{\text{mask}} &= \mathcal{H}_n(k_t^v rG, \text{pid_tag}) \\
k_{\text{payment ID}} &= k_{\text{mask}} \rightarrow \text{ridotto alla lunghezza in bit dell'ID di pagamento} \\
\text{payment ID} &= \text{XOR}(k_{\text{payment ID}}, \text{payment ID codificato})
\end{aligned}
$$

Allo stesso modo, i mittenti possono decodificare gli ID di pagamento che avevano precedentemente codificato ricalcolando il segreto condiviso.

4.5 Indirizzi Multifirma

Può risultare utile condividere la proprietà dei fondi tra più persone o indirizzi. Il Capitolo 9 tratta questo argomento nel dettaglio.

[23] I dati della transazione non indicano a quale output appartenga un ID di pagamento. I destinatari devono identificare i propri ID di pagamento.

CAPITOLO 5

Offuscamento degli Importi

Nella maggior parte delle criptovalute, come Bitcoin, gli output delle transazioni, che conferiscono i diritti di spesa su determinate somme di denaro, comunicano tali importi in chiaro. Questo permette agli osservatori esterni di verificare facilmente che l'importo speso sia uguale all'importo inviato.

Su Monero vengono utilizzate dei *commitment* (impegni crittografici) per nascondere gli importi degli output a qualsiasi entità, eccetto al mittente e al destinatario, pur continuando a garantire agli osservatori che una transazione non invii né più né meno di quanto viene speso. Come vedremo, i commitment sugli importi devono essere accompagnati da una *range proof* (prove di appartenenza all'intervallo), che dimostrano che l'importo nascosto rientra in un intervallo legittimo.

5.1 Commitment

In generale, uno *schema di commitment* crittografico è un insieme di regole da seguire per impegnarsi su un valore scelto senza rivelare il valore stesso. Dopo l'assunzione di un commitment, non è più possibile modificare il valore su cui è stato preso l'impegno.

Ad esempio, in un gioco *testa o croce*, Alice potrebbe impegnarsi privatamente su un esito (cioè "chiamarlo") calcolando l'hash del valore impegnato con dei dati segreti e pubblicando tale hash. Dopo che Bob lancia la moneta, Alice dichiara quale valore aveva scelto e lo dimostra rivelando i dati segreti. Bob può quindi verificare la sua dichiarazione.

In altre parole, supponiamo che Alice abbia una stringa segreta *blah* e il valore su cui vuole impegnarsi sia *heads*. Calcola l'hash $h = \mathcal{H}(blah, heads)$ e lo consegna a Bob. Bob lancia la

moneta, poi Alice dice a Bob la stringa segreta *blah* e che si era impegnata su *heads*. Bob calcola $h' = \mathcal{H}(blah, heads)$. Se $h' = h$, allora sa che Alice aveva scelto *heads* prima del lancio.

Alice utilizza quello che viene chiamato 'sale' (salt) (su *blah*) in modo che Bob non possa semplicemente indovinare $\mathcal{H}(heads)$ e $\mathcal{H}(tails)$ prima del lancio e capire che si era impegnata su *heads*.[1]

5.2 Commitment di Pedersen

Un *commitment di Pedersen* [112] è un impegno che ha la proprietà di essere un *omomorfismo rispetto all'addizione*. Se $C(a)$ e $C(b)$ denotano i commitment sui valori a e b rispettivamente, allora vale $C(a + b) = C(a) + C(b)$.[2] Questa proprietà sarà utile nel nascondere gli importi delle transazioni, poiché si può dimostrare, ad esempio, che gli input sono uguali agli output, senza rivelare gli importi.

Fortunatamente, i commitment di Pedersen sono facili da implementare con la crittografia a curve ellittiche, in quanto vale banalmente:

$$aG + bG = (a + b)G$$

Chiaramente, definendo un impegno semplicemente come $C(a) = aG$, potremmo facilmente creare delle tabelle di commitment pre-calcolati che ci aiutino a riconoscere i valori comuni di a.

Per ottenere una privacy a livello di teoria dell'informazione (cioè anche contro un avversario con potenza computazionale infinita), è necessario aggiungere un *fattore di offuscamento* (blinding factor) segreto e un altro generatore H, tale che non sia noto per quale valore di γ valga: $H = \gamma G$. La difficoltà del problema del logaritmo discreto assicura che calcolare γ a partire da H sia irrealizzabile.[3]

Possiamo quindi definire l'impegno su un valore a come $C(x, a) = xG + aH$, dove x è il fattore di offuscamento (anche detta *maschera*) che impedisce agli osservatori di indovinare a.

[1] Se il valore impegnato è molto difficile da indovinare e verificare, ad esempio se è un punto apparentemente casuale su una curva ellittica, allora non è necessario aggiungere "sale" al commitment.

[2] Omomorfismo rispetto all'addizione, in questo contesto, significa che l'addizione viene preservata quando si trasformano gli scalari in punti sulla curva ellittica applicando, per uno scalare x, $x \to xG$.

[3] Nel caso di Monero, $H = 8 * to_point(\mathcal{H}_n(G))$. Questo differisce dalla funzione di hash $\mathcal{H}_p$ in quanto interpreta direttamente l'output di $\mathcal{H}_n(G)$ come coordinate compresse di un punto, invece di derivare matematicamente un punto sulla curva (vedi [106]). Le ragioni storiche di questa differenza ci sono sconosciute, e in effetti questo è l'unico caso in cui $\mathcal{H}_p$ non viene utilizzata (anche Bulletproofs usa $\mathcal{H}_p$). Nota come viene effettuata un'operazione di moltiplicazione per 8, per assicurarsi che il punto risultante appartenga al sottogruppo di ordine l (anche $\mathcal{H}_p$ lo fa).

src/ringct/rctTypes.h tests/unit_tests/ringct.cpp `TEST(ringct, HPow2)`

Il commitment $C(x, a)$ è privato a livello di teoria dell'informazione perché esistono molte possibili combinazioni di x e a che producono lo stesso C.[4] Se x è veramente casuale, un attaccante non avrebbe letteralmente alcun modo per scoprire a [85, 121].

5.3 Commitment degli Importi

In Monero, gli importi degli output sono memorizzati nelle transazioni come commitment di Pedersen. Definiamo un commitment sull'importo b di un output come:

$$C(y, b) = yG + bH$$

src/ringct/rctOps.cpp `addKeys2()`

I destinatari devono poter sapere quanto denaro è contenuto in ciascun output a loro destinato, in modo tale da ricostruire i commitment sugli importi ed in modo che possano essere usati come input per nuove transazioni. Questo significa che il fattore di offuscamento y e l'importo b devono essere comunicati al ricevente.

La soluzione adottata è un segreto condiviso Diffie-Hellman rK_B^v scambiato tramite la chiave pubblica della transazione (transaction public key) (vedi Sezione 4.2.1). Per ogni transazione sulla blockchain, ciascuno dei suoi output $t \in \{0, ..., p-1\}$ ha una maschera y_t che il mittente e il destinatario possono calcolare privatamente, e un *importo* memorizzato nei dati della transazione. Mentre y_t è uno scalare su curve ellittiche e occupa 32 byte, b è limitato a 8 byte grazie alla range proof, per cui è sufficiente memorizzare solo un valore a 8 byte.[5,6]

$$y_t = \mathcal{H}_n(\text{“}commitment_mask\text{”}, \mathcal{H}_n(rK_B^v, t))$$
$$amount_t = b_t \oplus_8 \mathcal{H}_n(\text{“}amount\text{”}, \mathcal{H}_n(rK_B^v, t))$$

src/ringct/rctOps.cpp `ecdhEncode()`

In questo contesto l'operatore $\oplus_8$ indica un'operazione di XOR (Sezione 2.5) tra i primi 8 byte di ciascun operando (b_t, che è già a 8 byte, e $\mathcal{H}_n(...)$, che è a 32 byte). I destinatari possono eseguire la stessa operazione XOR su $amount_t$ per recuperare b_t.

Il destinatario Bob sarà in grado di calcolare il fattore di offuscamento y_t e l'importo b_t usando la chiave pubblica della transazione rG e la sua chiave di visualizzazione k_B^v. Potrà anche verificare che il commitment $C(y_t, b_t)$ fornito nei dati della transazione, d'ora in poi denotato C_t^b, corrisponda effettivamente all'importo ricevuto.

[4] In sostanza, esistono molti x' e a' tali che $x' + a'\gamma = x + a\gamma$. Chi crea il commitment conosce una combinazione, ma un attaccante non ha modo di sapere quale sia. Questa proprietà è nota anche come 'offuscamento perfetto' (*perfect hiding*) [136]. Inoltre, neanche chi effettua il commitment può trovare un'altra combinazione senza risolvere il problema del logaritmo discreto per γ, una proprietà nota come 'vincolamento computazionale' (*computational binding*) [136].

[5] Come con l'indirizzo monouso K^o della Sezione 4.2, l'indice dell'output t è concatenato al segreto condiviso prima dell'hashing. Questo garantisce che output diretti allo stesso indirizzo non abbiano maschere o importi simili, se non con probabilità trascurabile. Inoltre, come prima, il termine rK_B^v è moltiplicato per 8, quindi è in realtà $8rK_B^v$.

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_tx_with_tx_key()` chiama `generate_output_ephemeral_keys()`

[6] Questa soluzione (implementata nella versione 10 del protocollo) ha sostituito un metodo precedente che utilizzava più dati, provocando così il passaggio dal tipo di transazione v3 (`RCTTypeBulletproof`) a v4 (`RCTTypeBulletproof2`). La prima edizione di questo rapporto discuteva il metodo precedente [30].

Più in generale, qualsiasi terza parte che possieda la chiave di visualizzazione di Bob potrebbe decifrare gli importi dei suoi output, e verificare che corrispondano ai commitment associati.

5.4 Introduzione a RingCT

Una transazione contiene riferimenti agli output di altre transazioni (indicando agli osservatori quali vecchi output sono spesi), e ai propri output. Il contenuto di un output include un indirizzo one-time (a cui assegna la proprietà dell'output) e un commitment che nasconde l'importo (ovvero l'importo cifrato spiegato nella Sezione 5.3).

Sebbene i verificatori di una transazione non conoscano quanto denaro sia contenuto in ciascun input o output, devono comunque poter verificare che la somma degli input sia pari alla somma degli output. Monero utilizza una tecnica chiamata RingCT [107], implementata per la prima volta nel gennaio 2017 (versione 4 del protocollo), per permettere di effettuare questa verifica.

Se abbiamo una transazione con m input che contengono importi $a_1, ..., a_m$, e p output con importi $b_0, ..., b_{p-1}$, allora un osservatore si aspetterebbe giustamente che:[7]

$$\sum_j a_j - \sum_t b_t = 0$$

Nonostante γ sia sconosciuto, data la linearità dell'operazione di somma tra commitment, possiamo facilmente dimostrare agli osservatori che il totale degli input è pari al totale degli output semplicemente imponendo la somma algebrica dei commitment relativi agli input e output pari a zero (ovvero imponendo la somma dei fattori di offuscamento degli output uguale a quella degli input):[8]

$$\sum_j C_{j,in} - \sum_t C_{t,out} = 0$$

Per evitare che il mittente sia identificabile, usiamo un approccio leggermente diverso. Gli importi spesi corrispondono agli output di transazioni precedenti, che avevano commitment del tipo:

$$C_j^a = x_j G + a_j H$$

Il mittente può creare nuovi commitment agli stessi importi, ma con fattori di offuscamento diversi, ovvero:

$$C_j'^a = x_j' G + a_j H$$

Chiaramente, conosce la chiave privata della differenza tra i due commitment:

$$C_j^a - C_j'^a = (x_j - x_j')G$$

[7] Se il totale degli output non corrisponde esattamente a una combinazione di output posseduti, l'autore della transazione può aggiungere un output di "resto" che invia il denaro extra a sé stesso. Per analogia: se hai una banconota da 20$ e spendi 15$, riceverai 5$ di resto dal cassiere.

[8] Si ricorda dalla Sezione 2.3.1 che possiamo sottrarre un punto invertendo le sue coordinate e poi sommando. Se $P = (x, y)$, allora $-P = (-x, y)$. Si ricorda anche che le negazioni in campo finito sono calcolate $\pmod q$, cioè $(-x \pmod q)$.

Pertanto, il mittente può usare questo valore come un *commitment a zero*, poiché può firmare con la chiave privata $(x_j - x'_j) = z_j$ e dimostrare che non c'è alcun termine H nella somma (assumendo che γ sia sconosciuto). In altre parole, può dimostrare che $C^a_j - C'^a_j = z_j G + 0H$, cosa che vedremo nel Capitolo 6, quando discuteremo della struttura delle transazioni RingCT.

Sia C'^a_j un *pseudo-commitment di output.* I pseudo-commitment sono inclusi nei dati della transazione, uno per ciascun input.

Prima di accettare una transazione nella blockchain, la rete verificherà che gli importi in entrata e uscita siano bilanciati. I fattori di offuscamento dei pseudo-commitment e degli output vengono scelti in modo tale che:

$$\sum_j x'_j - \sum_t y_t = 0$$

Questo permette di dimostrare che la somma degli importi in input è pari a quella degli output:

src/ringct/rctSigs.cpp verRct-Semantics-Simple()

$$(\sum_j C'^a_j - \sum_t C^b_t) = 0$$

Fortunatamente, scegliere tali fattori è semplice. Nella versione attuale di Monero, tutti i fattori di offuscamento sono casuali tranne che per il m^{esimo} pseudo-commitment, dove x'_m è banalmente:

genRct-Simple()

$$x'_m = \sum_t y_t - \sum_{j=1}^{m-1} x'_j$$

5.5 Prove di Intervallo

Un problema con la somma dei commitment è che, se abbiamo i commitment $C(a_1)$, $C(a_2)$, $C(b_1)$ e $C(b_2)$ e intendiamo usarli per dimostrare che $(a_1 + a_2) - (b_1 + b_2) = 0$, allora quei commitment risulterebbero comunque validi anche se uno dei valori nell'equazione fosse *negativo*.

Per esempio, potremmo avere $a_1 = 6$, $a_2 = 5$, $b_1 = 21$ e $b_2 = -10$.

$$(6 + 5) - (21 + -10) = 0$$

dove

$$21G + -10G = 21G + (l - 10)G = (l + 11)G = 11G$$

Poiché $-10 = l - 10$, abbiamo effettivamente creato l Moneroj in più (oltre 7,2 $\times$ 10^{74}!) di quanti ne abbiamo inseriti.

La soluzione a questo problema in Monero consiste nel dimostrare che ogni importo in uscita si trovi in un certo intervallo (da 0 a $2^{64} - 1$) usando il sistema di prove Bulletproof (Range Proof),

descritto per la prima volta da Benedikt Bünz *et al.* in [40] (e spiegato anche in [136, 47]).[9] Dato il carattere complesso e articolato delle Bulletproof, non verranno spiegate in questo documento. Inoltre, riteniamo che i materiali citati ne illustrino adeguatamente i concetti.[10]

L'algoritmo di verifica delle Bulletproof prende in input gli importi in uscita b_t e le maschere d'impegno y_t, e produce tutti i C_t^b e un n-upla di prova aggregata strutturata come segue[11,12]:

src/ringct/rctSigs.cpp `proveRangeBulletproof()`

$$\Pi_{BP} = (A, S, T_1, T_2, \tau_x, \mu, \mathbb{L}, \mathbb{R}, a, b, t)$$

Questa singola prova viene usata per dimostrare che tutti gli importi in uscita sono contemporaneamente nel range, sfruttando il vantaggio offerto dall'aggregazione che riduce notevolmente lo spazio richiesto (sebbene aumenti il tempo di verifica).[13] L'algoritmo di verifica prende in input tutti i C_t^b e Π_{BP} e restituisce `true` se tutti gli importi impegnati sono nel range da 0 a $2^{64} - 1$.

L'n-upla Π_{BP} occupa $(2 \cdot \lceil \log_2(64 \cdot p) \rceil + 9) \cdot 32$ byte di memoria.

[9] È teoricamente possibile che, con diverse uscite legittime all'interno del range, la somma dei loro importi vada in overflow e causi un problema simile. Tuttavia, quando il massimo importo di uscita è molto più piccolo di l, serve un numero enorme di uscite perché ciò accada. Per esempio, se l'intervallo è 0-5 e $l = 99$, allora per falsificare denaro usando un input di 2 servirebbe $5 + 5 + \ldots + 5 + 1 = 101 \equiv 2 \pmod{99}$, per un totale di 21 uscite. In Monero, l è circa 2ˆ189 volte più grande dell'intervallo disponibile, il che significa che sarebbero necessarie ben 2ˆ189 uscite per falsificare denaro.

[10] Prima della versione 8 del protocollo, le prove di range venivano effettuate con firme ad anello Borromeane, spiegate nella prima edizione di Da Zero a Monero [30].

[11] I vettori $\mathbb{L}$ e $\mathbb{R}$ contengono ciascuno $\lceil \log_2(64 \cdot p) \rceil$ elementi. $\lceil \ \rceil$ indica che il logaritmo è arrotondato per eccesso. A causa della loro costruzione, alcune Bulletproof includono 'uscite fittizie' come padding per assicurare che p più il numero di uscite fittizie sia una potenza di 2. Queste uscite fittizie possono essere generate durante la verifica e non sono memorizzate con i dati della prova.

[12] Le variabili in una Bulletproof non sono correlate ad altre variabili in questo documento. L'eventuale sovrapposizione di simboli è puramente casuale e non voluta. Nota che gli elementi del gruppo $A, S, T_1, T_2, \mathbb{L}$ e $\mathbb{R}$ sono moltiplicati per 1/8 prima di essere memorizzati, e poi moltiplicati per 8 durante la verifica. Questo garantisce che siano tutti membri del sottogruppo l (si veda la Sezione 2.3.1).

[13] È possibile 'batchare' più Bulletproof distinte cioè verificarle simultaneamente. Questo migliora i tempi di verifica, e attualmente in Monero le Bulletproof vengono verificate in batch per ogni blocco, anche se non vi è alcun limite teorico al numero di prove batchabili. Ogni transazione può contenere solo una Bulletproof.

rct/ringct/bulletproofs.cc `bulletproof_VERIFY()`

CAPITOLO 6

Transazioni Confidenziali ad Anello (RingCT)

Nei Capitoli 4 e 5 sono stati trattati diversi aspetti delle transazioni Monero. A questo punto, una semplice transazione con un input e un output inizializzata da un mittente sconosciuto ed indirizzata ad un destinatario sconosciuto potrebbe, in grosso modo, apparire così:

"La transazione utilizza la chiave pubblica (tx public key) rG. Sarà speso l'output precedente X (nota che ha un importo nascosto A_X, impegnato su C_X). Verrà assegnato un impegno di pseudo-output C'_X. In seguito il mittente genererà un output Y, che potrà essere speso dal proprietario dell'indirizzo one-time K^o_Y. In fine avrà un importo nascosto A_Y impegnato in C_Y, cifrato per il destinatario e con l'appartenenza all'intervallo dimostrata tramite una range proof in stile Bulletproof. Si noti che $C'_X - C_Y = 0$."

Rimangono alcune questioni da risolvere. L'autore possedeva davvero X? L'impegno di pseudo-output C'_X corrisponde effettivamente a C_X, tale che $A_X = A'_X = A_Y$? Qualcuno ha manomesso la transazione, magari reindirizzando l'output verso un destinatario diverso da quello designato dal mittente?

Come menzionato nella Sezione 4.2, è possibile dimostrare la proprietà di un output firmando un messaggio con il relativo indirizzo one-time (chi possiede la chiave privata dell'indirizzo possiede anche l'output). È possibile inoltre dimostrare che tale output ha lo stesso importo di un impegno di pseudo-output, dimostrando la conoscenza della chiave privata dell'impegno a zero ($C_X - C'_X = z_X G$). Se, in aggiunta, il messaggio firmato è *l'intera transazione* (esclusa la firma stessa), allora i verificatori possono essere certi dell'autenticità dell'intento del mittente (la firma risulta valida solo per il messaggio originale). Le firme MLSAG permettono di ottenere

tutto ciò, offuscando al contempo quale output sia stato effettivamente speso tra quelli presenti nella blockchain, impedendo così agli osservatori di determinare quale specifico output sia stato utilizzato.

6.1 Tipologie di Transazione

Monero è una criptovaluta in continua evoluzione. La struttura delle transazioni, i protocolli e gli schemi crittografici sono soggetti a cambiamenti man mano che emergono nuove innovazioni, obiettivi o minacce.

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_tx_with_tx_key()`

In questo documento l'autore ha concentrato l'attenzione sulle *Transazioni Confidenziali ad Anello*, anche dette *RingCT* e sulla loro implementazione nella versione attuale di Monero. Il formato RingCT è obbligatorio per tutte le nuove transazioni Monero, di conseguenza non saranno trattati schemi di transazione deprecati, anche se ancora parzialmente supportati.[1] La tipologia di transazione trattata finora, e che sarà ulteriormente approfondita in questo capitolo, è la `RCTTypeBulletproof2`.[2]

Un riepilogo concettuale delle transazioni Monero è reperibile nella Sezione 6.3.

6.2 Transazioni Confidenziali ad Anello `RCTTypeBulletproof2`

Attualmente (versione 12 del protocollo) tutti i nuovi trasferimenti di moneta devono utilizzare questa tipologia di transazione, in cui ogni input viene firmato separatamente. Un esempio reale di una transazione `RCTTypeBulletproof2`, con una descrizione dettagliata di tutti i suoi campi, è riportato nell'Appendice A.

6.2.1 Commitment sugli Importi e Commissioni di Transazione

Supponiamo che il mittente di una transazione abbia precedentemente ricevuto vari output con importi $a_1, ..., a_m$ destinati a indirizzi one-time $K^o_{\pi,1}, ..., K^o_{\pi,m}$ e con commitment sugli importi $C^a_{\pi,1}, ..., C^a_{\pi,m}$.

Questo mittente conosce le chiavi private $k^o_{\pi,1}, ..., k^o_{\pi,m}$ corrispondenti agli indirizzi one-time (Sezione 4.2). Il mittente conosce anche i fattori di offuscamento x_j utilizzati nei commitment $C^a_{\pi,j}$ (Sezione 5.3).

[1] RingCT è stato implementato per la prima volta a gennaio 2017 (versione 4 del protocollo). È diventato obbligatorio per tutte le nuove transazioni a settembre 2017 (versione 6 del protocollo) [15]. RingCT è la versione 2 del protocollo di transazioni Monero.

[2] Durante l'era RingCT ci sono stati tre tipi di transazione adesso deprecati: `RCTTypeFull`, `RCTTypeSimple` e `RCTTypeBulletproof`. I primi due coesistevano nella prima versione di RingCT e sono trattati nella prima edizione di questo documento [30]. Con l'introduzione dei Bulletproof (versione 8 del protocollo), `RCTTypeFull` è stato deprecato e `RCTTypeSimple` aggiornato a `RCTTypeBulletproof`. In seguito a dei piccoli miglioramenti applicati alla cifratura delle maschere e degli importi degli output (versione 10 del protocollo), è stato introdotto `RCTTypeBulletproof2`.

Tipicamente, il totale degli output di una transazione è *inferiore* rispetto al totale degli input, in quanto il mittente paga una commissione che incentiva i miner a includere la transazione nella blockchain.[3] Le commissioni di transazione f sono memorizzate in chiaro nei dati della transazione trasmessa alla rete. Inoltre, i miner possono creare un output aggiuntivo per sé stessi di importo pari alla commissione (vedi Sezione 7.3.6).

In generale, una transazione è composta da m input $a_1, ..., a_m$ e p output $b_0, ..., b_{p-1}$ tali che:

$$\sum_{j=1}^{m} a_j - \sum_{t=0}^{p-1} b_t - f = 0$$

.[4]

Il mittente calcola dei commitment pseudo-output per gli importi in input, $C'^a_{\pi,1}, ..., C'^a_{\pi,m}$, e crea i commitment per gli importi in output desiderati $b_0, ..., b_{p-1}$. Indichiamo questi nuovi commitment con $C^b_0, ..., C^b_{p-1}$.

Il mittente conosce le chiavi private $z_1, ..., z_m$ relative ai commitment a zero $(C^a_{\pi,1} - C'^a_{\pi,1}), ..., (C^a_{\pi,m} - C'^a_{\pi,m})$.

Affinché i verificatori possano confermare che gli importi della transazione in entrate ed uscita siano bilanciati, l'importo della commissione deve essere convertito in un impegno. La soluzione consiste nel calcolare l'impegno della commissione f senza il fattore di offuscamento (blinding factor). Ovvero, $C(f) = fH$.

Ciò consente di dimostrare che la somma degli importi in input equivale alla somma degli importi in output:

src/ringct/rctSigs.cpp verRctSemanticsSimple()

$$(\sum_j C'^a_j - \sum_t C^b_t) - fH = 0$$

6.2.2 Firma

Il mittente seleziona m insiemi di dimensione v costituiti da indirizzi one-time aggiuntivi e non correlati con i rispettivi commitment dalla blockchain. Questi indirizzi corrispondono ad output

[3] In Monero esiste una commissione di base minima che scala con il peso della transazione. È semi-obbligatoria perché, sebbene sia possibile estrarre blocchi contenenti transazioni con commissioni molto basse, la maggior parte dei nodi Monero non propagherà tali transazioni ad altri nodi. Il risultato è che pochi, se non nessun, miner cercheranno di includerle nei blocchi. Gli autori delle transazioni possono offrire commissioni ai miner superiori al minimo, se lo desiderano. Per maggiori dettagli si rimanda alla Sezione 7.3.4.

[4] Gli output sono mescolati casualmente dall'implementazione core prima di essere indicizzati, così da impedire che osservatori costruiscano euristiche basate sull'ordine. Gli input sono invece ordinati per key image nei dati della transazione.

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_tx_with_tx_key()`

apparentemente non spesi.[5,6] Per firmare l'input j, il mittente inserisce un insieme di dimensione v in un *ring* (anello) assieme al proprio j^{esimo} indirizzo one-time non speso (posizionato all'indice unico π), assieme a falsi commitment a zero, nel modo seguente:[7]

$$\begin{aligned}\mathcal{R}_j = \{&\{K^o_{1,j}, (C_{1,j} - C'^a_{\pi,j})\},\\ &...\\ &\{K^o_{\pi,j}, (C^a_{\pi,j} - C'^a_{\pi,j})\},\\ &...\\ &\{K^o_{v+1,j}, (C_{v+1,j} - C'^a_{\pi,j})\}\}\end{aligned}$$

Alice utilizza una firma MLSAG (Sezione 3.5) per firmare questo ring, dove conosce le chiavi private $k^o_{\pi,j}$ per $K^o_{\pi,j}$ e z_j per l'impegno a zero $(C^a_{\pi,j} - C'^a_{\pi,j})$. Poiché non è necessaria alcuna key image per i commitment a zero, non c'è quindi alcuna componente key image nella costruzione della firma.[8]

src/ringct/ rctSigs.cpp proveRct-MGSimple()

Ogni input di una transazione è firmato individualmente usando un annello $\mathcal{R}_j$ definito come sopra, offuscando così gli output reali spesi, $(K^o_{\pi,1}, ..., K^o_{\pi,m})$, tra altri output non spesi.[9] Poiché parte di ogni anello include un impegno a zero, l'impegno pseudo-output usato deve contenere un importo pari a quello realmente speso. Ciò dimostra il corretto bilanciamento degli input, senza rivelare quale membro dell'anello è l'input reale.

Il messaggio $\mathfrak{m}$ firmato da ciascun input è essenzialmente un hash di tutti i dati della transazione *eccetto* le firme MLSAG.[10] Questo garantisce che le transazioni siano a prova di manomissione sia

[5] In Monero è prassi che gli insiemi di 'indirizzi non correlati aggiuntivi' vengano selezionati tramite una distribuzione gamma sull'intervallo di output storici (per RingCT; per gli output pre-RingCT si utilizza una distribuzione triangolare). Questo metodo utilizza un procedimento chiamato *binning* per uniformare le differenze di densità tra blocchi. Si calcola il tempo medio tra output transazionali fino a un anno prima per output RingCT (tempo medio = [#output/#blocchi] · tempo blocco). Si seleziona un output tramite la distribuzione gamma, e dal suo relativo blocco si infine si prende un output casuale. Questo ultimo output farà parte dell'anello di decoy (esche). [90]

src/wallet/ wallet2.cpp `get_outs()` `gamma_picker::pick()`

[6] Dalla versione 12 del protocollo, tutti gli input delle transazioni devono avere almeno 10 blocchi di età (`CRYPTONOTE_DEFAULT_TX_SPENDABLE_AGE`). Prima della versione 12, l'implementazione *core* richiedeva almeno 10 blocchi per impostazione predefinita, ma non era obbligatorio, quindi un wallet alternativo poteva adottare regole diverse, come alcuni effettivamente fecero [80].

[7] In Monero ogni ring di una transazione deve avere la stessa dimensione, e il protocollo controlla quanti membri può avere ciascun ring per ogni output da spendere. Il valore è cambiato con le versioni del protocollo: v2 marzo 2016 ≥ 3, v6 settembre 2017 ≥ 5, v7 aprile 2018 ≥ 7, v8 ottobre 2018 solo 11. Dalla v6 ogni ring non può contenere membri duplicati, anche se possono esserci duplicati tra ring diversi, per permettere input multipli quando il numero di output storici non è sufficiente a evitare sovrapposizioni tra ring [133].

[8] La costruzione e la verifica della firma escludono il termine $r_{i,2}\mathcal{H}_p(C_{i,j} - C'^a_{\pi,j}) + c_i\tilde{K}_{z_j}$.

[9] Il vantaggio di firmare gli input individualmente è dato dal fatto che l'insieme degli input reali e dei commitment a zero non deve essere posizionato allo stesso indice π, come invece accadrebbe nel caso aggregato. Questo significa che anche se l'origine di un input venisse scoperta, l'origine degli altri input rimarrebbe nascosta. Il vecchio tipo di transazione `RCTTypeFull` usava firme ad anello aggregate, combinando tutti i ring in uno solo, ed è stato deprecato proprio per questo motivo.

src/crypto-note_core/ cryptonote_core.cpp `check_tx_inputs_ring_members_diff()`

[10] Il messaggio effettivo è $\mathfrak{m} = \mathcal{H}(\mathcal{H}(tx_prefix), \mathcal{H}(ss), \mathcal{H}(\text{range proofs}))$ dove:

tx_prefix ={versione dell'era transazionale (es. RingCT = 2), input {offset dei membri del ring, key image}, output {indirizzi one-time}, extra {chiave pubblica della transazione, ID di pagamento o ID codificato, vari}}

ss ={tipo di transazione (`RCTTypeBulletproof2` = '4'), commissione di transazione, commitment pseudo-output

src/ringct/ rctSigs.cpp `get_pre_mlsag_hash()`

per gli autori sia per i verificatori. Viene prodotto un solo messaggio, e ciascun input MLSAG lo firma.

La chiave privata one-time k^o è l'essenza del modello di transazioni di Monero. Firmare $\mathfrak{m}$ con k^o dimostra la proprietà dell'importo impegnato in C^a. I verificatori possono essere sicuri che l'autore della transazione sta spendendo i propri fondi, senza sapere quali fondi, quanto sta spendendo o quali altri fondi possiede!

6.2.3 Il Problema della Doppia Spesa

Una firma MLSAG (Sezione 3.5) contiene le immagini $\tilde{K}_j$ delle chiavi private $k_{\pi,j}$. Una proprietà importante per qualsiasi schema di firma crittografica è che la firma prodotta sia falsificabile con probabilità trascurabile. Pertanto, a tutti gli effetti pratici, possiamo assumere che le immagini chiave (key image) di una firma siano state prodotte in modo deterministico da chiavi private legittime.

src/cryptonote_core/blockchain.cpp `have_tx_keyimges_as_spent()`

La rete deve solo verificare che le immagini chiave incluse nelle firme MLSAG (corrispondenti agli input e calcolate come $\tilde{K}^o_j = k^o_{\pi,j}\mathcal{H}_p(K^o_{\pi,j})$) non siano già comparse in altre transazioni.[11] Se lo sono, allora possiamo essere certi che stiamo assistendo a un tentativo di riutilizzare un output già speso $(C^a_{\pi,j}, K^o_{\pi,j})$.

6.2.4 Requisiti di Memoria

Firma MLSAG (input)

Dalla Sezione 3.5 ricordiamo che una firma MLSAG in questo contesto può essere espressa come

$$\sigma_j(\mathfrak{m}) = (c_1, r_{1,1}, r_{1,2}, ..., r_{v+1,1}, r_{v+1,2}) \text{ con } \tilde{K}^o_j$$

Come eredità di CryptoNote, i valori $\tilde{K}^o_j$ non sono indicati formalmente come parte della firma, ma piuttosto come *immagini* delle chiavi private $k^o_{\pi,j}$. Queste *immagini chiave* (key image) sono normalmente memorizzate in uno spazio apposito nella struttura della transazione, poiché vengono utilizzate per rilevare attacchi di doppia spesa.

Tenendo conto di ciò, e assumendo la compressione dei punti (Sezione 2.4.2), dato che ogni anello $\mathcal{R}_j$ contiene $(v+1)\cdot 2$ chiavi, una firma di input σ_j richiederà $(2(v+1)+1)\cdot 32$ byte. Oltre a ciò, l'immagine chiave $\tilde{K}^o_{\pi,j}$ e l'impegno pseudo-ouput $C'^a_{\pi,j}$ portano il totale a $(2(v+1)+3)\cdot 32$ byte per input.

A questo valore va aggiunto lo spazio necessario per memorizzare gli offset dei membri dell'anello nella blockchain (vedi Appendice A). Questi offset sono usati dai verificatori per trovare le chiavi

per gli input, ecdhInfo (importi cifrati), commitment sugli output}.
Vedi Appendice A per maggiori dettagli sulla terminologia.

[11] I verificatori devono anche controllare che l'immagine di chiave appartenga al sottogruppo del generatore (Sezione 3.4).

degli output e i commitment dei membri dell'anello MLSAG all'interno della blockchain, e sono memorizzati come interi a lunghezza variabile, per cui non è possibile quantificare esattamente lo spazio necessario.[12,13,14]

src/ringct/rctSigs.cpp verRctMGSimple() verRctSemanticsSimple()

Verificare tutte le firme MLSAG di una transazione `RCTTypeBulletproof2` include il calcolo di $(C_{i,j} - C'^{a}_{\pi,j})$ e $(\sum_j C'^{a}_j \stackrel{?}{=} \sum_t C^b_t + fH)$, oltre a verificare che le immagini di chiave appartengano al sottogruppo di G, ovvero che $l\tilde{K} \stackrel{?}{=} 0$.

Prove di Intervallo (output)

Una prova di intervallo aggregata con Bulletproof richiede $(2 \cdot \lceil \log_2(64 \cdot p) \rceil + 9) \cdot 32$ byte totali.

src/ringct/bulletproofs.cpp `bulletproof_VERIFY()`

[12] Vedi [64] o [23] per una spiegazione del tipo di dato varint di Monero. È un tipo di intero che usa fino a 9 byte, e memorizza fino a 63 bit di informazione.

src/common/varint.h

[13] Si supponga che la blockchain contenga una lunga lista di output di transazioni. Vogliamo riportare gli indici degli output da usare negli anelli. Indici più grandi richiedono più spazio, dunque basta riportare la posizione "assoluta" di un solo indice per anello, e le posizioni "relative" degli altri membri. Per esempio, con indici reali {7,11,15,20} basta riportare {7,4,4,5}. I verificatori possono calcolare l'ultimo indice sommando (7+4+4+5 = 20). I membri degli anelli sono organizzati in ordine crescente di indice nella blockchain.

src/cryptonote_basic/cryptonote_format_utils.cpp `absolute_output_offsets_to_relative()`

[14] Una transazione con 10 input usando anelli di 11 membri totali richiederà $((11 \cdot 2 + 3) \cdot 32) \cdot 10 = 8000$ byte per i suoi input, con circa 110–330 byte per gli offset (ci sono 110 membri di anello).

6.3 Riepilogo Concettuale: Transazioni Monero

Per riassumere questo capitolo, e i due precedenti, illustriamo il contenuto principale di una transazione, organizzato per chiarezza concettuale. Un esempio reale si trova nell'Appendice A.

- Tipo: '0' indica `RCTTypeNull` (per i miner), '4' indica `RCTTypeBulletproof2`
- Input: per ogni input $j \in \{1, ..., m\}$ speso dall'autore della transazione
  - **Offset dei membri dell'anello**: una lista di 'offset' che indicano dove un verificatore può trovare i membri dell'anello $i \in \{1, ..., v+1\}$ dell'input j nella blockchain (incluso l'input reale)
  - **Firma MLSAG**: termini σ_j: c_1, e $r_{i,1}$ & $r_{i,2}$ per $i \in \{1, ..., v+1\}$
  - **Key image**: l' immagine della chiave $\tilde{K}_j^{o,a}$ per l'input j
  - **Commitment sullo pseudo-output**: $C_j'^a$ per l'input j
- Output: per ogni output $t \in \{0, ..., p-1\}$ verso l'indirizzo o sottoindirizzo (K_t^v, K_t^s)
  - **Indirizzo monouso**: $K_t^{o,b}$ per l'output t
  - **Commitment dell'output**: C_t^b per l'output t
  - **Importo codificato**: permette al destinatario di calcolare b_t per l'output t
    - *Importo*: $b_t \oplus_8 \mathcal{H}_n(\text{“}amount\text{”}, \mathcal{H}_n(rK_B^v, t))$
  - **Range Proof**: Bulletproof aggregato per tutti gli importi b_t
    - *Prova*: $\Pi_{BP} = (A, S, T_1, T_2, \tau_x, \mu, \mathbb{L}, \mathbb{R}, a, b, t)$
- Commissione di transazione: comunicata in chiaro moltiplicata per 10^{12} (cioè in unità atomiche, vedi Sezione 7.3.1), quindi una commissione pari a 1.0 sarà registrata come 1000000000000
- Extra: contiene la chiave pubblica della transazione rG, oppure, se almeno un output è indirizzato a un sottoindirizzo, $r_t K_t^{s,i}$ per ciascun output a sottoindirizzi t e $r_t G$ per ciascun output a indirizzo standard t, ed eventualmente un payment ID codificato (al massimo uno per transazione)[15]

La precedente transazione di esempio, composta da un solo input e un solo output, può essere descritta come segue:

> La transazione utilizza la chiave pubblica della transazione rG. Spenderà uno degli output presenti nel set $\mathbb{X}$ (si noti che l'importo associato è nascosto come A_X, impegnato nel commitment C_X). L'output speso appartiene all'autore della transazione (ha firmato con un MLSAG sugli indirizzi monouso contenuti in $\mathbb{X}$) e non è stato ancora

[15] Nessuna informazione memorizzata nel campo 'extra' è verificata, anche se è firmata dagli MLSAG degli input, quindi non può essere manomessa (tranne che con probabilità trascurabile). Il campo non ha limiti sulla quantità di dati che può contenere, purché venga rispettato il peso massimo della transazione. Vedi [79] per ulteriori dettagli.

speso in precedenza (la sua key image $\tilde{K}$ non è ancora apparsa sulla blockchain). A questo output viene assegnato uno pseudo-impegno C'_X.

In seguito verrà generato un nuovo output Y, spendibile dall'indirizzo monouso K^o_Y. Questo output nasconde l'importo A_Y, impegnato in C_Y, cifrato per il destinatario e verificato tramite un Bulletproof per garantirne il range.

La transazione include anche una commissione pari a f. Si osservi che vale l'equazione $C'_X - (C_Y + C_f) = 0$, e che l'autore ha firmato anche l'impegno a zero $C'_X - C_X = zG$, a dimostrazione del fatto che l'importo in input è uguale a quello in output ($A_X = A'_X = A_Y + f$). Il MLSAG dell'autore firma l'intera transazione, fornendo così la garanzia agli osservatori che la transazione non sia stata alterata.

6.3.1 Requisiti di Memoria

Per `RCTTypeBulletproof2` servono $(2(v+1)+2)\cdot m \cdot 32$ byte di memoria, e la prova di range Bulletproof aggregata richiede $(2 \cdot \lceil \log_2(64 \cdot p) \rceil + 9) \cdot 32$ byte.[16]

Requisiti vari:

- Immagini delle chiavi di input: $m * 32$ byte
- Indirizzi one-time degli output: $p * 32$ byte
- Commitment degli output: $p * 32$ byte
- Importi cifrati negli output: $p * 8$ byte
- Chiave pubblica della transazione: 32 byte normalmente, $p * 32$ byte se si invia almeno a un sottoindirizzo
- ID di pagamento: 8 byte per un indirizzo integrato. Non dovrebbe essercene più di uno per transazione.
- Commissione di transazione: memorizzata come intero a lunghezza variabile, quindi ≤ 9 byte
- Offset degli input: memorizzati come interi a lunghezza variabile, quindi ≤ 9 byte per offset, per $m * (v+1)$ membri dell'anello
- Tempo di sblocco: memorizzato come intero a lunghezza variabile, quindi ≤ 9 byte[17]
- Tag 'Extra': ogni dato nel campo 'extra' (es. una chiave pubblica) inizia con un byte 'tag', e alcuni hanno anche un byte (o più) di *lunghezza*; vedi Appendice A per i dettagli

[16] La quantità di informazioni contenibili in una transazione è limitata da un peso massimo ('transaction weight'). Prima che i Bulletproof venissero implementati nella versione 8 del protocollo (e attualmente se le transazioni hanno solo due output), il peso e la dimensione in byte della transazione erano equivalenti. Il peso massimo è (0.5*300kB - `CRYPTONOTE_COINBASE_BLOB_RESERVED_SIZE`), dove lo spazio riservato per il blob (600 byte) è dedicato alla transazione di mining all'interno dei blocchi. Prima della versione 8 il moltiplicatore 0.5 non era incluso, e il termine 300kB era più piccolo nelle versioni precedenti del protocollo (20kB v1, 60kB v2, 300kB v5). Approfondiamo questi argomenti nella Sezione 7.3.2.

src/cryptonote_core/tx_pool.cpp `get_transaction_weight_limit()`

[17] L'autore di qualsiasi transazione può bloccare i suoi output, rendendoli inutilizzabili fino a un'altezza di blocco specificata (o fino a un timestamp UNIX). Ha solo l'opzione di bloccare tutti gli output allo stesso blocco. Non è chiaro se questo offra un'utilità concreta (forse per smart contract). Le transazioni di mining hanno un tempo di blocco obbligatorio pari a 60 blocchi. Dal protocollo v12 gli output tradizionali non possono essere spesi prima di 10 blocchi. Se una transazione è pubblicata al blocco 10 con tempo di sblocco pari a 25, potrà essere spesa dal blocco 25 in poi. Il tempo di sblocco è probabilmente la funzionalità meno usata di Monero.

src/cryptonote_core/blockchain.cpp `is_tx_spendtime_unlocked()`

CAPITOLO 7

La Blockchain di Monero

L'era di Internet ha portato una nuova dimensione all'esperienza umana. Possiamo connetterci con persone in ogni angolo del pianeta, oltre ad avere un'immensa quantità di informazioni a portata di mano. Lo scambio di beni e servizi è fondamentale per una società pacifica e prospera [93], e nell'era digitale possiamo offrire la nostra produttività all'intero pianeta.

I mezzi di scambio (denaro) sono essenziali, poiché ci danno un punto di riferimento monetario per una grande varietà di beni economici che altrimenti sarebbe impossibile valutare, e permettono interazioni reciprocamente vantaggiose tra persone che potrebbero non avere nulla in comune [93]. Nel corso della storia ci sono stati molti tipi di moneta, dalle conchiglie, alla carta fino all'oro. Questi mezzi venivano scambiati solo manualmente, a differenza di oggi dove il denaro può essere trasferito elettronicamente.

Nel modello attuale, di gran lunga il più diffuso, le transazioni elettroniche sono gestite da istituzioni finanziarie terze. A queste istituzioni viene affidata la custodia del denaro e le persone si affidano a loro per trasferirlo su richiesta. Tali istituzioni mediano le controversie tra controparti, inoltre, i loro pagamenti sono reversibili e possono essere censurati o controllati da potenti organizzazioni [95].

Per alleviare questi svantaggi sono state ideate le valute digitali decentralizzate.[1]

[1] Questo capitolo include più dettagli di implementazione rispetto ai capitoli precedenti, poiché la natura di una blockchain dipende fortemente dalla sua struttura specifica.

7.1 Valuta Digitale

Progettare una valuta digitale non è semplice. Ne esistono tre tipi: personale, centralizzata o distribuita. Si tenga presente che una valuta digitale è semplicemente una collezione di messaggi, e gli 'importi' registrati in quei messaggi sono interpretati come quantità monetarie.

Nel **modello email** chiunque può creare moneta (ad esempio con la creazione di un messaggio che dice 'posseggo 5 monete'), e chiunque può inviare le proprie monete più e più volte a chiunque abbia un indirizzo email. Non esiste un limite massimo alla quantità di moneta 'stampabile' e non è impedita la spesa ripetuta delle stesse monete (doppia spesa, anche detto *double spending*).

Nel **modello videogame**, dove l'intera valuta è memorizzata/registrata in un database centralizzato, gli utenti devono affidarsi all'onestà del custode. La quantità di valuta prodotta e in circolazione non è verificabile da osservatori esterni, e il custode può cambiare le regole in gioco in qualsiasi momento, o essere censurato da poteri esterni.

7.1.1 Versione Distribuita degli Eventi

Nella valuta digitale "condivisa", o "distribuita", i computer che interagiscono con questa moneta hanno ciascuno una copia del registro di ogni transazione di valuta. Quando viene fatta una nuova transazione su un computer, questa viene trasmessa agli altri computer e accettata se rispetta le regole predefinite.

Gli utenti usufruiscono delle monete solo quando altri utenti le accettano in come pagamento. In generale, gli utenti accettano solo monete che ritengono legittime. Per massimizzare l'utilità delle loro monete, gli utenti sono naturalmente portati a concordare un unico insieme di regole comunemente accettate, senza la presenza e l'intervento di un'autorità centrale[2]:

Regola 1: Il denaro può essere creato solo in scenari chiaramente definiti.

Regola 2: Le transazioni spendono denaro già esistente.

Regola 3: Un utente può spendere una moneta una solo volta.

Regola 4: Solo l'utente che possiede una moneta può spenderla.

Regola 5: Le transazioni producono in uscita una quantità di denaro pari a quella spesa.

Regola 6: Le transazioni sono compilate secondo un formato ben preciso.

Le regole da 2 a 6 sono trattate dallo schema di transazione discusso nel Capitolo 6, che aggiunge i benefici di fungibilità e privacy legati alla *firma ambigua*, alla ricezione anonima dei fondi e al trasferimento di importi nascosti. La Regola 1 sarà successivamente spiegata nel dettaglio in questo

[2] In scienze politiche questo si chiama Punto di Schelling [61], minimo sociale o contratto sociale.

capitolo.[3] Le transazioni su blockchain usano la crittografia, dunque l'oggetto della transazione è denoestratto *criptovaluta*.

Se due computer ricevono due transazioni legittime differenti che spendono la stessa moneta prima che abbiano modo di comunicare tra loro, come viene decisa quale sia quella corretta? In questo caso, si verifica la cosiddetta 'biforcazione' della moneta, perché esistono due copie diverse che rispettano le stesse regole.

Chiaramente, la prima transazione legittima che spende una moneta dovrebbe essere quella canonicamente accetta dagli utenti. Ciò è più facile a dirsi che a farsi. Come vedremo, ottenere il consenso sullo storico delle transazioni costituisce la ragion d'essere della tecnologia blockchain.

7.1.2 Blockchain Semplice

Per prima cosa abbiamo bisogno che tutti i computer, d'ora in poi chiamati *nodi*, concordino sull'ordine delle transazioni.

Supponiamo che una valuta sia creata con una dichiarazione *genesis*: "Che SampleCoin abbia inizio!". Definiamo questo messaggio con il nome di 'blocco', e il suo relativo hash identificativo come

$$BH_G = \mathcal{H}(\text{“Che SampleCoin abbia inizio!”})$$

Ogni volta che un nodo riceve delle transazioni, usa gli hash di quelle transazioni, TH, come messaggi, insieme all'identificativo del blocco precedente, per calcolare nuovi hash di blocco

$$BH_1 = \mathcal{H}(BH_G, TH_1, TH_2, ...)$$
$$BH_2 = \mathcal{H}(BH_1, TH_3, TH_4, ...)$$

E così via, pubblicando ogni nuovo blocco di messaggi appena creato. Ogni nuovo blocco fa riferimento al blocco precedente, in modo tale da creare un ordine chiaro degli eventi che si estende a catena fino al messaggio *genesis*. Otteniamo una 'blockchain' molto semplice.[4]

I nodi possono includere un *timestamp* nei loro blocchi per agevolare la registrazione. Se la maggior parte dei nodi è onesta con i timestamp, allora la blockchain fornisce un quadro abbastanza fedele di quando ogni transazione è stata registrata.

Se diversi blocchi che fanno riferimento allo stesso blocco precedente vengono pubblicati contemporaneamente, allora la rete di nodi entrerà in uno stato di biforcazione, poiché ogni gruppo di nodi riceverà uno dei nuovi blocchi prima dell'altro gruppo (per semplicità, si supponga che circa la metà dei nodi finisca con una parte della biforcazione e l'altra metà con l'altra).

[3] Nelle monte merce come l'oro, queste regole sono rispettate dalla realtà fisica.

[4] Tecnicamente una blockchain è un 'grafo diretto aciclico' (DAG), con le blockchain in stile Bitcoin come variante unidimensionale. I DAG contengono un numero finito di nodi e archi unidirezionali (vettori) che collegano i nodi. Se si parte da un nodo, non si torna mai al punto di partenza indipendentemente dal percorso scelto. [6]

7.2 Difficoltà

Se i nodi potessero pubblicare nuovi blocchi quando vogliono, la rete potrebbe frammentarsi e divergere in molte catene diverse, tutte ugualmente legittime. Supponiamo che ci vogliano 30 secondi affinché tutti i nodi della rete ricevano un nuovo blocco. Cosa succede se i blocchi vengono inviati ad esempio ogni 31, 15, 10 secondi, ecc?

È possibile controllare la velocità con cui l'intera rete produce nuovi blocchi. Se il tempo necessario a creare un nuovo blocco è molto maggiore del tempo necessario al blocco precedente per raggiungere la maggioranza dei nodi, la rete tenderà a rimanere integra.

7.2.1 Estrazione di un Blocco

L'output di una funzione hash crittografica è distribuito uniformemente e apparentemente indipendente dall'input. Ciò significa che la probabilità che, dato un input, venga prodotto un deterestratto hash è ugualmente distribuita tra tutti i valori possibili e producibili in output. Inoltre, calcolare un singolo hash richiede un certa quantità di tempo.

Si supponga l'esistenza di una funzione hash $\mathcal{H}_i(x)$ che produce un numero da 1 a 100: $\mathcal{H}_i(x) \in^D_R \{1, ..., 100\}$.[5] Dato un x, la funzione $\mathcal{H}_i(x)$ seleziona lo stesso numero 'casuale' da $\{1, ..., 100\}$ ogni volta che viene calcolata. Il tempo richiesto per calcolare $\mathcal{H}_i(x)$ è di 1 minuto.

Supponiamo ci venga dato un messaggio $\mathfrak{m}$, e ci venga chiesto di trovare un *nonce* n (un intero) tale che $\mathcal{H}_i(\mathfrak{m}, n)$ produca un numero minore o uguale al *target* $t = 5$ (cioè $\mathcal{H}_i(\mathfrak{m}, n) \in \{1, ..., 5\}$).

Dato che solo 1 su 20 output di $\mathcal{H}_i(x)$ soddisfa il target, ci vorranno in media circa 20 tentativi di n per trovare un nonce valido (quindi circa 20 minuti di calcolo).

La ricerca di un nonce utile si chiama *mining* (estrazione), e la pubblicazione del messaggio con il suo nonce è una *Proof of Work* (prova di lavoro) perché dimostra di aver trovato un nonce valido (anche se per fortuna al primo tentativo, o addirittura pubblicato un nonce valido totalmente a caso), che chiunque può verificare calcolando $\mathcal{H}_i(\mathfrak{m}, n)$.

Ora immaginiamo una funzione hash per generare prove di lavoro, $\mathcal{H}_{PoW} \in^D_R \{0, ..., m\}$, dove m è il suo output massimo possibile. Dato un messaggio $\mathfrak{m}$ (un blocco di informazioni), un nonce n da mining, e un target t, possiamo definire il numero medio atteso di hash, la *difficoltà* d, come:

$$d = m/t$$

Se $\mathcal{H}_{PoW}(\mathfrak{m}, n) * d \leq m$, allora $\mathcal{H}_{PoW}(\mathfrak{m}, n) \leq t$ e n è accettabile.[6]

src/crypto-note_basic/dif
`check_hash()`

Con target più piccoli la difficoltà aumenta e un computer deve calcolare sempre più hash, e quindi impiega tempi sempre più lunghi, per trovare nonce validi.[7]

[5] Usiamo $\in^D_R$ per indicare che l'output è deterministico ma casuale.

[6] In Monero vengono registrate e calcolate solo le difficoltà dato che $\mathcal{H}_{PoW}(\mathfrak{m}, n) * d \leq m$ non richiede t.

[7] Mining e verifica sono asimmetrici perché verificare una Proof of Work (un singolo calcolo dell'algoritmo di proof of work) richiede lo stesso tempo indipendentemente dalla difficoltà.

7.2.2 Velocità di Estrazione

Supponiamo che tutti i nodi stiano estraendo blocchi contemporaneamente, ma interrompano l'estrazione del loro 'blocco corrente' non appena ne ricevono uno nuovo dalla rete. Immediatamente iniziano a estrarre un nuovo blocco che fa riferimento a quello ricevuto.

Consideriamo un insieme di b blocchi recenti della blockchain (con indice $u \in \{1, \ldots, b\}$), ciascuno con difficoltà d_u. Per ora, assumiamo che i nodi che li hanno estratti siano onesti, quindi ogni timestamp TS_u è accurato.[8]
Il tempo totale trascorso tra il blocco più vecchio e quello più recente è

$$totalTime = TS_b - TS_1$$

Il numero approssimativo di hash necessari per estrarre tutti i blocchi è

$$totalDifficulty = \sum_{u=1}^{b} d_u$$

Possiamo quindi stimare la velocità di calcolo degli hash dell'intera rete. Se la potenza di calcolo non è variata in manieria brusca durante la produzione di questi blocchi, otterremo

$$hashSpeed \approx \frac{totalDifficulty}{totalTime}$$

Se vogliamo stimare la potenza di calcolo necessaria per produrre nuovi blocchi ad un ritmo di

$$\frac{UnBlocco}{totalTime}$$

calcoliamo il numero di hash che la rete dovrebbe calcolare in media per raggiungere quel target. Notare che arrotondiamo per eccesso in modo tale che la difficoltà non sia mai zero:

$$newDifficulty = hashSpeed \times targetTime$$

Non vi è alcuna garanzia che il blocco successivo richieda esattamente un numero di hash di rete pari a *newDifficulty*, ma nel tempo, ricalibrando continuamente, la difficoltà seguirà la reale potenza di mining e i blocchi tenderanno a impiegare circa $targetTime$.[9]

7.2.3 Consenso: Difficoltà Cumulativa Maggiore

Ora possiamo risolvere i conflitti tra biforcazioni (fork) della catena.

Per convenzione, la catena con la difficoltà cumulativa maggiore (somma delle difficoltà di tutti i blocchi) — e quindi con il maggior quantitativo di lavoro di rete impegnato — è considerata la versione reale e legittima. Se la catena si divide e ogni fork ha la stessa difficoltà cumulativa, i nodi

[8] I timestamp vengono fissati quando un miner *inizia* a estrarre un blocco, perciò tendono a ritardare rispetto al momento in cui il blocco viene effettivamente pubblicato. Il blocco successivo inizia a essere estratto subito, quindi il timestamp che compare *dopo* un dato blocco indica quanto tempo i miner hanno speso su di esso.

[9] Se assumiamo che la potenza di *hashing* della rete aumenti costantemente e gradualmente, poiché le nuove difficoltà dipendono da hash *passati* (cioè antecedenti all'aumento di potenza), ci aspettiamo che i tempi medi di estrazione dei nuovi blocchi risultino leggermente inferiori a $targetTime$. L'effetto di ciò sullo schedule di emissione (Sezione 7.3.1) potrebbe essere compensato dalle penalità derivanti dall'aumento del peso dei blocchi, che esploriamo in Sezione 7.3.3.

continuano a estrarre sul ramo che hanno ricevuto per primo. Quando un ramo supera l'altro, il ramo più con minor lavoro cumulativo viene scartato (diventa un "orfano").[10]

Se i nodi desiderano modificare o aggiornare il protocollo di base — cioè l'insieme di regole con cui un nodo decide se una copia della blockchain o un nuovo blocco è legittimo — possono farlo facilmente *forkando* la catena. L'impatto sugli utenti dipende da quanti nodi adottano il nuovo ramo e da quanto l'infrastruttura software viene aggiornata.[11]

Per un attaccante, convincere i nodi onesti a modificare la cronologia delle transazioni — per esempio per riutilizzare o "annullare" una spesa — richiede di creare un fork con difficoltà totale superiore a quella della catena principale (che nel frattempo continua a crescere). Questo è molto difficile a meno di non controllare oltre il 50% della potenza di calcolo della rete [95].

7.2.4 Mining in Monero

Per mantenere le varie biforcazioni della blockchain su un piano paritario, in Monero non vengono utilizzati i blocchi più recenti per calcolare la nuova difficoltà, ma l'insieme di b blocchi viene spostato all'indietro di l blocchi. Ad esempio, se la catena ha 29 blocchi $(1, \ldots, 29)$, $b = 10$ e $l = 5$, verranno campionati i blocchi 15–24 per calcolare la difficoltà del blocco 30.

Se i nodi miner disonesti manipolano i timestamp in modo che la difficoltà non rispecchi la reale potenza di hashing, è possibile ovviare al problema ordinando i timestamp in ordine cronologico, quindi rimuovendo i primi o outlier (valori anomali) più piccoli e gli ultimi o outlier più grandi. Otteniamo così una "finestra" di $w = b - 2o$ blocchi. Nell'esempio precedente, se $o = 3$, eliminiamo i blocchi 15–17 e 22–24, lasciando i blocchi 18–21 per calcolare la difficoltà del blocco 30.

Prima di eliminare gli outlier ordiniamo solo i timestamp; l'ordine originale delle difficoltà rimane invariato. Usiamo le difficoltà cumulative di ciascun blocco (difficoltà di quel blocco più tutte le precedenti).

src/cryptonote_core/blockchain.cpp `get_difficulty_for_next_block()`

Dato l'array di w timestamp ordinati e l'array di difficoltà cumulative non ordinate (indicizzati da 1 a w), definiamo:

$$\mathit{totalTime} = \mathit{choppedSortedTimestamps}[w] - \mathit{choppedSortedTimestamps}[1]$$
$$\mathit{totalDifficulty} = \mathit{choppedCumulativeDifficulties}[w] - \mathit{choppedCumulativeDifficulties}[1]$$

In Monero il tempo target è 120 secondi (2 minuti), $l = 15$ (30 min), $b = 720$ (un giorno) e $o = 60$ (2 ore).[12,13]

src/cryptonote_config.h

[10] source?

[11] Gli sviluppatori di Monero hanno cambiato con successo il protocollo 11 volte, con quasi tutti gli utenti e miner che hanno adottato ciascun fork: v1 (genesis) 18 aprile 2014 [125]; v2 marzo 2016; v3 settembre 2016; v4 gennaio 2017; v5 aprile 2017; v6 settembre 2017; v7 aprile 2018; v8 e v9 ottobre 2018; v10 e v11 marzo 2019; v12 novembre 2019. Il README del repository core contiene un riepilogo delle modifiche di protocollo in ogni versione.

src/hardforks/ha `mainnet_hard_f`

[12] A marzo 2016 (v2 del protocollo) Monero è passato da 1 minuto a 2 minuti di tempo target per blocco [13]. Gli altri parametri di difficoltà sono rimasti invariati.

[13] L'algoritmo di difficoltà di Monero potrebbe non essere ottimale rispetto ai migliori algoritmi disponibili [142]. Fortunatamente è "abbastanza resistente al mining egoista" [48], una caratteristica essenziale.

Le difficoltà dei blocchi estratti non sono memorizzate nella catena, dunque chi necessita di queste informazioni dovrà scaricare una copia della blockchain e, dopo aver verificato la legittimità di tutti i blocchi, ricalcolare le difficoltà dai timestamp registrati. Ci sono alcune regole da seguire per i primi $b + l = 735$ blocchi:

src/cryptonote_basic/difficulty.cpp `next_difficulty()`

Regola 1: Ignorare completamente il blocco genesis (blocco 0, con $d = 1$). I blocchi 1 e 2 hanno $d = 1$.

Regola 2: Prima di rimuovere gli outlier, cercare di ottenere la finestra w per calcolare *totalTime* e *totalDifficulty*.

Regola 3: Dopo w blocchi, eliminare gli outlier alti e bassi, scalando il numero rimosso fino ad arrivare a b blocchi. Se il numero di blocchi precedenti (meno w) è dispari, rimuovere un outlier basso in più rispetto agli alti.

Regola 4: Dopo b blocchi, campionare i primi b blocchi fino a $b + l$, dopodiché si procede normalmente con un ritardo di l.

Proof of Work (PoW) di Monero

Monero ha utilizzato vari algoritmi di Proof of Work (output da 32 byte) in diverse versioni del protocollo. L'originale, noto come CryptoNight, è stato progettato per essere relativamente inefficiente su GPU, FPGA e ASIC [123] rispetto a funzioni hash standard come SHA256. Nell'aprile 2018 (versione 7 del protocollo), i nuovi blocchi hanno iniziato a usare una variante leggermente modificata per contrastare l'avvento degli ASIC per CryptoNight [26]. Una successiva variante, chiamata CryptoNight V2, è stata adottata nell'ottobre 2018 (v8) [11], e CryptoNight-R (basata su CryptoNight ma con cambiamenti più sostanziali) è entrata in uso a marzo 2019 (v10) [12]. Infine, a novembre 2019 (v12) è stato introdotto obbligatoriamente un nuovo PoW radicale chiamato RandomX [68], progettato per garantire resistenza agli ASIC nel lungo termine [14].

src/cryptonote_basic/cryptonote_tx_utils.cpp `get_block_longhash()`

7.3 Offerta di Moneta

Esistono due metodi principali per creare moneta in un sistema di criptovaluta basata su blockchain.

Primo: i creatori della criptovaluta possono "coniare" monete e distribuirle alle persone tramite il *messaggio genesis*. Questo viene spesso chiamato "airdrop". Talvolta i creatori assegnano a se stessi una grande quantità moneta sempre tramite il messaggio genesis. In questo caso si configura un atto di "pre-mine", ovvero pre-estrezione. [19].

Secondo: la valuta può essere distribuita automaticamente come ricompensa per il mining di un blocco, proprio come avviene con l'estrazione dell'oro. In questo caso ci sono due modelli: nel modello di Bitcoin la fornitura totale è limitata. Le ricompense per blocco diminuiscono lentamente

fino ad arrivare a zero, dopodiché non viene creata più moneta. Nel modello inflazionistico, l'offerta aumenta indefinitamente.

Monero discende da una valuta chiamata Bytecoin che aveva una pre-mine consistente, seguita da ricompense per blocco [94]. Monero non ha avuto pre-mine e, come vedremo, le sue ricompense per blocco diminuiscono gradualmente fino ad arrivare a un valore minimo. Raggiunto questo valore, tutte le nuove ricompense di estrazione dei blocchi saranno di quell'importo fisso, rendendo Monero intrinsecamente inflazionistico [14].

7.3.1 Ricompensa per Blocco

I miner, prima di cercare il nonce, creano una "transazione miner", anche detta "transazione di mining", senza input e con almeno un output.[15] L'importo totale degli output è pari alla ricompensa per blocco più le commissioni di tutte le transazioni incluse, e questa somma è comunicata in chiaro. I nodi che ricevono un blocco estratto devono verificare che la ricompensa sia corretta e possono calcolare l'offerta di moneta attuale sommando tutte le ricompense dei blocchi passati.

src/cryptonote_core/blockchain.cpp `validate_miner_transaction()`

Oltre a distribuire moneta, le ricompense incentivano l'attività di mining. Se non ci fossero ricompense per blocco (e nessun altro meccanismo), perché qualcuno dovrebbe estrarre nuovi blocchi? Forse per altruismo o curiosità, ma una rete composta da pochi miner renderebbe facile a un attore malintenzionato ottenere più del 50% della potenza di hashing totale e riscrivere la storia recente della catena.[16] Questo è il motivo principale per cui, in Monero, le ricompense sono state programmate per non scendere sotto una certa soglia.

Con questo meccanismo di ricompense per blocco, la competizione tra miner può portare la potenza di hashing a un livello tale che il costo marginale di aggiungere ulteriore potenza supera la potenziale ricompensa marginale per l'estrazione dei blocchi (che procede a velocità costante), senza contare eventuali rischi e costi di opportunità. Ciò significa che, al crescere del valore della criptovaluta, la potenza di hashing totale tende anch'essa ad aumentare, rendendo sempre più difficile raggiungere più del 50% della potenza di rete.

Bit Shifting

Lo scorrimento dei bit (bit shifting) viene usato per calcolare la ricompensa di base per ogni blocco (come vedremo in Sezione 7.3.3, la ricompensa effettiva può anche essere minore di quella base).

[14] Possiamo definire Monero come una criptovaluta debolmente inflazionistica, in quanto la percentuale di inflazione tende a zero con l'aumentare di moneta estratta immessa nel sistema.

[15] Una transazione miner può avere qualunque numero di output, anche se l'implementazione core attuale ne crea soltanto uno. Inoltre, a differenza delle transazioni tradizionali, non ci sono restrizioni esplicite sul peso della transazione miner: è limitato solo dal peso massimo del blocco.

[16] All'aumentare della quota di potenza di hashing di un attaccante (oltre il 50%), diminuisce il tempo necessario per riscrivere blocchi sempre più vecchi. Dato un blocco vecchio x giorni, una potenza controllata v e una potenza *onesta* v_h $(v > v_h)$, serviranno circa $y = x \times \frac{v_h}{v - v_h}$ giorni per riscriverlo o eventualmente rimuoverlo dalla blokchain.

Dato un intero $A = 13$ con rappresentazione binaria [1101], se spostiamo i bit di A verso destra di 2 posizioni (operatore >>), otteniamo [0011].01, cioè 3.25. In pratica la parte decimale viene scartata, lasciando solo [0011] = 3.[17]

Calcolo della Ricompensa di Base

Indichiamo con M l'offerta di moneta attuale e con $L = 2^{64} - 1$ il "limite" dell'offerta (in binario [11...11] a 64 bit).[18] All'inizio di Monero la ricompensa base per blocco era

$$B = (L - M) >> 20$$

Se $M = 0$, allora,

$$L = 18{,}446{,}744{,}073{,}709{,}551{,}615,$$
$$B_0 = (L - 0) >> 20 = 17{,}592{,}186{,}044{,}415.$$

Questi valori sono in *unità atomiche* – 1 unità atomica di Monero non può essere divisa. Chiaramente sono numeri enormi: L supera i 18 quintilioni! Possiamo dividere tutto per 10^{12} per ottenere l'unità standard di Monero (XMR):

$$\frac{L}{10^{12}} = 18{,}446{,}744.073709551615$$
$$B_0 = \frac{(L - 0) >> 20}{10^{12}} = 17.592186044415.$$

Ecco la ricompensa del blocco genesis — distribuita a `thankful_for_today`, che avviò il progetto Monero [125] — di circa 17.6 Monero! Vedi Appendice C per conferma.[19]

Man mano che i blocchi vengono estratti, M cresce e le ricompense diminuiscono. Inizialmente (dal blocco genesis di aprile 2014) i blocchi veniva estratti ogni minuto, ma a marzo 2016 diventarono due minuti per blocco [13]. Per mantenere inalterato il "programma di emissione",[20] le ricompense vennero raddoppiate: da allora si utilizza $(L-M) >> 19$ anziché $>> 20$. Attualmente la ricompensa base è

$$B = \frac{(L - M) >> 19}{10^{12}}$$

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_block_reward()`

7.3.2 Peso Dinamico dei Blocchi

Idealmente, l'inclusione di ogni nuova transazione in un blocco dovrebbe essere immediata. Ma se qualcuno inviasse un numero enorme di transazioni in modo malevolo, la blockchain potrebbe crescere rapidamente a dismisura.

Una mitigazione possibile a questo problema, è impostare un limite fisso alla dimensione dei blocchi (in byte), in modo tale da limitare il numero di transazioni che possono essere incluse in un blocco.

[17] Lo scorrimento a destra di n bit equivale alla divisione intera per 2^n.
[18] Questo spiega perché le proof di range (Sezione 5.5) limitano gli importi delle transazioni a 64 bit.
[19] Gli importi in Monero sono memorizzati in unità atomiche nella blockchain.
[20] Per un confronto interessante tra gli schedule di Monero e Bitcoin, vedi [18].

Cosa succede se il volume di transazioni legittime aumenta? Ogni mittente offrirebbe commissioni più alte per assicurarsi un posto nel blocco successivo. I miner selezionerebbero le transazioni con le commissioni più alte, rendendo proibitive le commissioni per importi piccoli (ad esempio Alice che compra una mela da Bob). Solo chi offre di più riuscirebbe a far includere la propria transazione.[21]

In media stat virtus, dunque Monero evita questi estremi (illimitato vs fissato) impostando dinamicamente il peso dei blocchi.

Dimensione vs Peso

Da quando sono stati introdotti i Bulletproof (versione 8), non si parla più di dimensione in byte ma di *peso* della transazione. Il peso di una transazione miner (vedi Sezione 7.3.6), o di una transazione tradizionale con due output, è pari alla sua dimensione in byte. Se una transazione tradizionale ha più di due output, il peso è leggermente maggiore della dimensione.

src/cryptonote_basic/cryptonote_format_utils.cpp `get_transaction_weight()`

Richiamando la Sezione 5.5, un Bulletproof occupa

$$(2 \cdot \lceil \log_2(64 \cdot p) \rceil + 9) \cdot 32 byte$$

Dunque aggiungendo man mano output alla transazione, lo spazio di memoria richiesto dai range proof cresce in maniera sublineare. Tuttavia, il tempo di verifica dei Bulletproof è lineare, di conseguenza un aumento artificiale del peso comporta anche un tempo di verifica extra, detto *clawback*.

Supponiamo di avere una transazione con p output e, se p non è potenza di 2, aggiungiamo output fittizi per raggiungere la potenza successiva. Calcoliamo la differenza tra la dimensione effettiva dei Bulletproof e la dimensione che avrebbero i Bulletproof se quei p + *output fittizi* fossero stati in transazioni da 2 output (è zero se $p = 2$). Applichiamo solo l'80% di questa differenza:[22]

src/cryptonote_basic/cryptonote_format_utils.cpp `get_transaction_weight_clawback()`

$$\text{transaction_clawback} = 0.8 \times \left[(23 \times \tfrac{p+\text{num_dummy_outs}}{2}) \cdot 32 - (2 \cdot \lceil \log_2(64 \cdot p) \rceil + 9) \cdot 32\right]$$

Di conseguenza, il peso della transazione è:

$$\text{transaction_weight} = \text{transaction_size} + \text{transaction_clawback}.$$

Il peso di un blocco è la somma dei pesi di tutte le sue transazioni più il peso della transazione miner.

src/cryptonote_core/blockchain.cpp `create_block_template()`

Peso di Blocco a Lungo Termine

Se il peso dei blocchi dinamici cresce troppo rapidamente, la blockchain potrebbe diventare ingestibile [81]. Per evitare questo problema, il peso massimo dei blocchi è vincolato dal *peso di*

[21] Bitcoin soffre di una congestione cronica. Questo sito (`https://bitcoinfees.info/`) mostra le commissioni da capogiro pagate ai miner (fino a 35$ per transazione).

[22] Si noti che $\log_2(64 \cdot 2) = 7$ e $2 \cdot 7 + 9 = 23$.

blocco a lungo termine. Ogni blocco ha, oltre al peso normale, un "peso a lungo termine" calcolato in base alle medie dei pesi a lungo termine effettivi dei precedenti 100 000 blocchi (incluso il proprio).[23][24],[25]

src/cryptonote_core/blockchain.cpp `update_next_cumulative_weight_limit()`

$$\begin{aligned}\text{longterm_block_weight} &= \min\{\text{block_weight},\ 1.4 \times \text{previous_effective_longterm_median}\},\\ \text{effective_longterm_median} &= \max\{\ 300\text{kB},\ \text{median_100000blocks_longterm_weights}\}.\end{aligned}$$

Se il peso normale rimane elevato a lungo, ci vogliono almeno 50 000 blocchi (circa 69 giorni) affinché la media a lungo termine effettive salga del 40

Mediana Cumulativa dei Pesi

Il volume delle transazioni può fluttuare rapidamente, ad esempio durante le festività [126]. Per gestire questa variabilità, Monero concede una flessibilità a breve termine. La "mediana cumulativa" di un blocco usa la media dei pesi normali degli ultimi 100 blocchi (incluso il proprio), con un limite minimo e massimo:

$$\begin{aligned}\text{cumulative_weights_median} = max\{&300\text{kB}, min\{max\{300\text{kB}, \text{median_100blocks_weights}\},\\ &50 * \text{effective_longterm_median}\}\}\end{aligned}$$

Il blocco successivo è vincolato da:[26]

$$\text{max_next_block_weight} = 2 * \text{cumulative_weights_median}$$

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_block_reward()`

Sebbene il peso massimo di un blocco possa arrivare fino a 100 volte la media a lungo termine effettiva dopo alcune centinaia di blocchi, esso non può superare il 40% di tale valore nei successivi 50 000 blocchi. Così, il peso a lungo termine regola la crescita e, nel breve, il peso può temporaneamente aumentare rispetto al valore di equilibrio.

7.3.3 Penalità su Ricompense di Blocco

Per estrarre blocchi di peso superiore al mediano cumulativo, i miner pagano una penalità sotto forma di riduzione della ricompensa. È possibile quindi distinguere due "zone" all'interno del peso massimo di blocco: la zona senza penalità e la zona con penalità. La mediana può salire lentamente, permettendo l'estrazione di blocchi progressivamente più pesanti senza penalità.

Se il peso previsto del blocco è maggiore della media cumulativa, allora, data la ricompensa base B, la penalità P è

$$P = B \times \left(\frac{\text{block_weight}}{\text{cumulative_weights_median}} - 1\right)^2.$$

[23] Analogamente alle difficoltà, anche i pesi di blocco e i pesi a lungo termine sono calcolati e memorizzati dai verificatori, non inclusi nei dati della blockchain.

[24] I blocchi antecedenti all'introduzione dei pesi a lungo termine hanno tale peso pari al peso normale.

[25] All'inizio il termine "300 kB" era 20 kB, poi salito a 60 kB a marzo 2016 (v2) [13] e infine fissato a 300 kB ad aprile 2017 (v5) [1]. Questo "pavimento" non nullo aiuta a stabilizzare il peso quando il volume è basso.

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_min_block_weight()`

[26] La mediana cumulativa ha sostituito "M100" nel protocollo v8.

La ricompensa effettiva per blocco è quindi:[27]

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_block_reward()`

$$B^{\text{actual}} = B - P,$$
$$B^{\text{actual}} = B \times \left[1 - \left(\frac{\text{block_weight}}{\text{cumulative_weights_median}} - 1\right)^2\right].$$

L'esponente 2 rende le penalità sub-proporzionali al peso. Un blocco del 10% più pesante della media ha solo l'1% di penalità, del 50% ha il 25%, del 90% l'81% e così via [18].

In genere, i miner dovrebbero produrre blocchi più pesanti della media cumulativa quando la commissione guadagnata aggiungendo un'altra transazione supera la penalità prevista.

7.3.4 Commissione Minima Dinamica

Per evitare che attori malevoli intasino la blockchain con transazioni inutili (che inquinano le firme ad anello) o gonfino eccessivamente la catena, Monero applica una commissione minima per ogni byte di dati di transazione.[28] Inizialmente la commissione era pari a 0.01 XMR/KiB (versione 1)[78], poi 0.002 XMR/KiB a settembre 2016 (versione 3).[29]

src/cryptonote_core/blockchain.cpp `check_fee()`

A gennaio 2017 (versione 4) è stato introdotto un algoritmo di commissione dinamica per KiB[43, 42, 41, 71].[30] Con i Bulletproof (versione 8) si è passati da KiB a byte. Il punto chiave è impedire che la commissione minima cumulativa superi la ricompensa per blocco (anche con ricompense contenute e blocchi pesanti), situazione che crea instabilità[96, 44, 56].[31]

Algoritmo di Calcolo della Commissione

Basiamo l'esecuzione dell'algoritmo su una transazione, riferita qui[71], di peso 3000 byte (simile a una `RCTTypeBulletproof2` con 2 input e 2 output, di circa 2600 byte)[32], e sulle commissioni

[27] Prima dell'introduzione delle transazioni confidenziali (RingCT, v4) gli importi erano in chiaro e, in alcune versioni iniziali del protocollo, suddivisi in chunk (es. 1244 → 1000+200+40+4). Per ridurre la dimensione delle transazioni miner, nelle versioni v2–v3 il core tagliava le cifre meno significative delle ricompense (sotto 0.0001 XMR; `BASE_REWARD_CLAMP_THRESHOLD`). La parte tagliata non andava persa, ma veniva incorporata nelle ricompense future. In generale, fin da v2 il calcolo della ricompensa qui presentato è solo un limite superiore a quanto effettivamente disperso negli output della transazione miner. Inoltre, i primissimi output di transazioni pre-RingCT con importi in chiaro non suddivisi non possono essere usati nelle firme ad anello nell'implementazione attuale: per spenderli vengono migrati in output chunked "mixabili".

src/cryptonote_core/blockc `validate_miner`

[28] Questa soglia è applicata dal protocollo di consenso dei nodi, non dal protocollo blockchain. La maggior parte dei nodi non ritrasmetterà una transazione con commissione al di sotto del minimo (in parte per evitare di rilanciare transazioni che probabilmente non saranno mai estratte [42]), ma accetterà un blocco che la contenga. Non serve quindi retrocompatibilità con algoritmi di commissione passati.

src/cryptonote_core/tx_po `add_tx()`

[29] KiB (kibibyte = 1024 byte) è diverso da kB (kilobyte = 1000 byte).

[30] La commissione base è stata modificata da 0.002 a 0.0004 XMR/KiB ad aprile 2017 (versione 5)[1]. La prima edizione di questo documento descriveva l'algoritmo originale [30].

[31] Il merito dei concetti di questa sezione va a Francisco Cabañas ("ArticMine"), architetto del sistema dinamico di blocchi e commissioni di Monero.

[32] Una transazione base Bitcoin 1-in/2-out pesa 250 byte[24]; 2-in/2-out 430 byte.

necessarie a compensare la penalità quando la media è al minimo (300 kB)[42]. In altre parole, viene indotta una penalità da un peso di 303 kB.

La commissione F necessaria a compensare la penalità marginale MP aggiungendo una transazione di peso TW a un blocco di peso BW è:

$$F = MP = B \times \left[\left(\frac{BW+TW}{\text{cumulative_median}} - 1 \right)^2 - \left(\frac{BW}{\text{cumulative_median}} - 1 \right)^2 \right].$$

Definendo il fattore di peso di blocco $\text{WF}_b = (BW/\text{cumulative_median} - 1)$ e il fattore di peso transazione $\text{WF}_t = (TW/\text{cumulative_median})$, otteniamo:

$$F = B \times \left(2\, \text{WF}_b\, \text{WF}_t + \text{WF}_t^2 \right)$$

Usando un blocco di 300 kB (media pari a 300 kB) e la transazione di riferimento (3000 byte):

$$\begin{aligned} F_{\text{ref}} &= B \times \left(2 \times 0 \times \text{WF}_t + \text{WF}_t^2 \right) \\ &= B \times \text{WF}_t^2 = B \times \left(\frac{TW_{\text{ref}}}{\text{cumulative_median}_{\text{ref}}} \right)^2. \end{aligned}$$

Questa commissione si distribuisce sull'1% della zona penalità (3000 su 300 000). Possiamo dunque distribuire la stessa commissione sull'1% di qualunque zona penalità con una generica transazione di riferimento:

$$\begin{aligned} \frac{TW_{\text{ref}}}{\text{cumulative_median}_{\text{ref}}} &= \frac{TW_{\text{general-ref}}}{\text{cumulative_median}_{\text{general}}}, \\ F_{\text{general-ref}} &= B \times \frac{TW_{\text{general-ref}}}{\text{cumulative_median}_{\text{general}}} \times \frac{TW_{\text{ref}}}{\text{cumulative_median}_{\text{ref}}}. \end{aligned}$$

A questo punto possiamo scalare la commissione in base al peso effettivo TW_{general} a un dato mediano:

$$\begin{aligned} F_{\text{general}} &= F_{\text{general-ref}} \times \frac{TW_{\text{general}}}{TW_{\text{general-ref}}} \\ &= B \times \frac{TW_{\text{general}}}{\text{cumulative_median}_{\text{general}}} \times \frac{TW_{\text{ref}}}{\text{cumulative_median}_{\text{ref}}}. \end{aligned}$$

Dividendo per TW_{general} otteniamo la commissione di base per byte desiderata:

$$f^B_{\text{default}} = \frac{F_{\text{general}}}{TW_{\text{general}}} = B \times \frac{1}{\text{cumulative_median}_{\text{general}}} \times \frac{3000}{300000}.$$

Quando il volume è sotto la media non serve che le commissioni siano al livello di riferimento [71]. Fissiamo la minima a 1/5 di quella di base:

$$\begin{aligned} f^B_{\text{min}} &= B \times \frac{1}{\text{cumulative_weights_median}} \times \frac{3000}{300000} \times \frac{1}{5} \\ &= B \times \frac{1}{\text{cumulative_weights_median}} \times 0.002. \end{aligned}$$

Mediana delle Commissioni

È emerso che usare la mediana cumulativa per le commissioni apre la porta ad attacchi spam. Alzando la mediana a breve termine al massimo (50 volte la mediana a lungo termine), un attaccante può pagare commissioni minime per mantenere blocchi pesanti con costi molto bassi.

Per evitarlo, vengono limitate le commissioni accettate per un nuovo blocco alla *mediana più piccola* disponibile, favorendo in ogni caso commissioni più alte.[33]

$$\text{smallest_median} = \max\{300\text{kB}, \min\{\text{median_100blocks_weights}, \text{effective_longterm_median}\}\}.$$

src/cryptonote_core blockchain.cpp `check_fee()`

Favorire commissioni più alte durante l'aumento del volume di transazioni aiuta a regolare la mediana a breve termine e a evitare che le transazioni rimangano in sospeso, poiché i miner tenderanno a includere anche quelle nella zona di penalità.

La commissione minima effettiva è quindi[34,35]

$$f^B_{min-actual} = \text{B} * (\frac{1}{\text{smallest_median}}) * 0.002$$

src/cryptonote_core blockchain.cpp `get_dynamic_base_fee()`

Commissioni di Transazione

Come ha spiegato Cabañas nella sua presentazione [42], ≪le commissioni indicano al miner fino a quale profondità nella zona di penalità l'autore della transazione è disposto a pagare per farsi includere≫. I miner riempiono i blocchi inserendo le transazioni in ordine decrescente di commissione (assumendo peso uguale), quindi per entrare nella zona di penalità devono esserci molte transazioni con commissioni elevate. Ciò implica che il peso di blocco massimo si raggiunge solo se le commissioni totali sono almeno 3 o 4 volte la ricompensa base (a quel punto la ricompensa effettiva scende a zero).[36]

Per calcolare le commissioni, il core wallet di Monero usa moltiplicatori di 'priorità'. Una transazione "lenta" paga la commissione minima, "normale" la commissione di base (5×), "veloce" (25×) può occupare il 2.5% della zona di penalità, e "super urgente" (1000×) può riempirla completamente.

src/wallet/wa `get_fee_mult`

Un'importante conseguenza dei blocchi dinamici è che le commissioni totali medie tenderanno a essere inferiori o al più pari alla ricompensa per blocco (si prevede parità attorno al 37% della zona

[33] Un attaccante potrebbe spendere il minimo necessario per portare la mediana a breve termine a 50 volte la mediana a lungo termine. Con le ricompense attuali di 2 XMR, un attaccante ottimizzato può aumentare la mediana a breve termine del 17% ogni 50 blocchi e raggiungere il massimo dopo circa 1300 blocchi (43 h), spendendo 0.78 XMR per blocco (65 kUSD totali), per poi tornare alla commissione minima. Appena la mediana delle commissioni raggiunge la zona senza penalità, la commissione minima totale per riempire quella zona sarà di 0.004 XMR (0.26 USD). Se la mediana delle commissioni raggiunge la mediana a lungo termine, in uno scenario dove la blockchain è sotto un attacco spam, sarebbe 1/50 della zona senza penalità, ossia 0.2 XMR a blocco (13 USD). Significa 2.88 XMR/giorno vs 144 XMR/giorno (per 69 giorni) per mantenere blocchi a 50 volte la mediana a lungo termine. Il costo iniziale di 1000 XMR vale la pena nel primo caso, ma non nel secondo. All'estremità della curva di emissione il setup cala a 300 XMR e la manutenzione a 43 XMR.

[34] Per verificare se una commissione è corretta, applichiamo un margine del 2% a $f^B_{min\text{-}actual}$ per gestire possibili overflow interi (la commissione va calcolata prima che il peso della tx sia completamente noto). Ciò significa che la minima effettiva è $0.98 * f^B_{min\text{-}actual}$.

[35] Sono in corso ricerche per migliorare ulteriormente le commissioni minime [129].

[36] La penalità marginale degli ultimi byte necessari per riempire un blocco può essere considerata una "transazione" come le altre. Per acquistare quello spazio, tutte le commissioni individuali devono superare la penalità, altrimenti il miner preferirà mantenere la ricompensa marginale. Questo richiede almeno 4 volte la ricompensa base in commissioni se il blocco è riempito con transazioni piccole; se il peso è 150 kB (50% della zona senza penalità) e la mediana minima (300 kB), basta un 3×.

di penalità, quando la penalità è al 13%). La competizione per lo spazio di blocco con commissioni più alte porta a un'offerta di spazio maggiore e commissioni più basse.[37] Questo meccanismo di retroazione contrasta efficacemente la minaccia del "selfish miner" [56].

7.3.5 Emissione Residua

Supponiamo una criptovaluta con offerta massima fissa e peso di blocco dinamico. Dopo un certo periodo le ricompense per blocco scendono a zero. Senza più penalità per blocchi più grandi, i miner includono qualsiasi transazione con fee non nulle nei loro blocchi.

I pesi dei blocchi si stabilizzano intorno al volume medio di transazioni inviate, e gli autori di transazioni non hanno incentivo a pagare fee superiori al minimo, che secondo la Sezione 7.3.4 sarebbe zero.

Questo crea una situazione instabile e insicura. I miner hanno scarso o nullo incentivo a estrarre nuovi blocchi, con conseguente calo della potenza di hashing di rete man mano che il ritorno sull'investimento diminuisce. I tempi di estrazione di un blocco restano costanti, ma il costo di un attacco double-spend può diventare sostenibile.[38] Se le fee minime sono forzate a rimanere non nulle, la minaccia del "selfish miner" [56] diventa reale [44].

Monero evita questo non permettendo che la ricompensa per blocco scenda sotto 0.6 XMR (0.3 XMR al minuto). Quando si verifica:

src/cryptonote_basic/cryptonote_basic_impl.cpp `get_block_reward()`

$$\begin{aligned} 0.6 &> ((L - M) >> 19)/10^{12}, \\ M &> L - 0.6 \times 2^{19} \times 10^{12}, \\ M/10^{12} &> L/10^{12} - 0.6 \times 2^{19}, \\ M/10^{12} &> 18\,132\,171.273709551615, \end{aligned}$$

la catena di Monero entrerà in uno stato di *emissione residua* (emission tail), con ricompense costanti di 0.6 XMR (0.3 XMR/minuto) per sempre.[39] Ciò corrisponde a un'inflazione iniziale di circa 0.9% annua, in lento declino.

7.3.6 Transazione Miner: `RCTTypeNull`

Il miner di un blocco ha il diritto di incassare le commissioni delle transazioni incluse e di coniare nuova moneta come ricompensa. Questo processo costruisce una transazione miner (anche detta transazione *coinbase*), simile a una transazione normale.[40]

src/cryptonote_core/cryptonote_tx_utils.cpp `construct_miner_tx()`

[37] Con il calo delle ricompense e l'aumento della media per maggiore adozione, le commissioni dovrebbero diventare progressivamente più piccole in termini reali.

[38] Anche il caso di offerta fissa e peso di blocco fisso, come in Bitcoin, è ritenuto instabile. [44]

[39] L'emission tail di Monero è stimato per maggio 2022 [16]. Il limite L sarà raggiunto a maggio 2024, ma poiché l'emissione non dipende più dall'offerta non avrà effetto. Grazie alle range proof di Monero, non sarà possibile inviare più di L in un singolo output anche accumulando più di quella somma e disponendo di wallet adeguati.

[40] In passato le transazioni miner potevano usare formati deprecati e contemporaneamente includere componenti RingCT. Questo problema è stato corretto nella versione 12 [8].

La somma degli importi degli output di una transazione miner non può superare la somma delle commissioni e ricompensa, ed è registrata in chiaro.[41] Invece degli input, viene registrata l'altezza del blocco (es. "richiedo ricompensa e fee per il blocco n-esimo").

La proprietà degli output appena coniati è assegnata a un normale indirizzo one-time[42], con la chiave pubblica della transazione memorizzata in `extra`. I fondi sono bloccati fino a 60 blocchi dopo la pubblicazione[20].[43]

Da quando è stato implementato RingCT (gennaio 2017, v4) [15], chi sincronizza una nuova copia della blockchain calcola l'impegno sull'importo della transazione miner (commitment) $C = 1G + aH$ e lo conserva. Questo permette ai miner di spendere i loro output appena coniati come normali output in anelli MLSAG.

src/cryptonote_core/blockchain.cpp `is_tx_spendtime_unlocked()`

I verificatori della blockchain memorizzano ciascun impegno degli output miner post-RingCT (32 byte ciascuno).

7.4 Struttura della Blockchain

La blockchain di Monero è tutto sommato semplice.

src/cryptonote_core/cryptonote_tx_utils.cpp `generate_genesis_block()`

Inizia con un messaggio *genesis* (nel caso di Monero si tratta di una transazione miner che distribuisce la prima ricompensa), che costituisce il *blocco genesis* (vedi Appendice C). Il blocco successivo contiene un riferimento a quello precedente, sotto forma di *Block ID*.

Un Block ID è l'hash dell'intestazione del blocco (una lista di informazioni sul blocco), di una cosiddetta "radice di Merkle" che incorpora tutti le Transaction ID (hash di ciascuna transazione), e del numero di transazioni (inclusa la transazione miner).[44]

src/cryptonote_basic/cryptonote_format_utils.cpp `get_block_hashing_blob()`

$$\text{Block ID} = \mathcal{H}_n\big(\text{Block header},\ \text{Merkle root},\ \#\text{transactions} + 1\big).$$

Per estrarre un nuovo blocco si producono prove di lavoro (proof of work) variando il nonce nell'header finché i target di difficoltà non sono soddisfatti.[45] La proof of work e il Block ID calcolano l'hash delle stesse informazioni, ma con funzioni diverse. I blocchi sono estratti iterando finché

$$(\text{PoW}_{\text{output}} \times \text{difficulty}) > 2^{256} - 1,$$

variando il nonce e ricalcolando

$$\text{PoW}_{\text{output}} = \mathcal{H}_{\text{PoW}}\big(\text{Block header},\ \text{Merkle root},\ \#\text{transactions} + 1\big).$$

src/cryptonote_basic/cryptonote_format_utils.cpp `calculate_block_hash()` `get_block_longhash()`

[41] Nell'implementazione corrente il miner può reclamare meno della ricompensa calcolata; l'eccedenza viene reinserita nel programma di emissione (emission schedule).

[42] L'output del miner può teoricamente essere inviato a un sottoindirizzo, usare multisig o un payment ID integrato; non è noto se esistano implementazioni che lo facciano.

[43] Non è possibile bloccarli per più o meno di 60 blocchi. Se pubblicata al blocco 10, lo sblocco avviene al 70.

[44] +1 accounts for the transazione miner.

[45] Un miner tipico di Monero (dati di https://monerobenchmarks.info/) esegue meno di 50 000 hash/s, quindi meno di 6 M hash per blocco. Il nonce a 4 byte (max 4.3 G) è sufficiente.

7.4.1 ID della Transazione

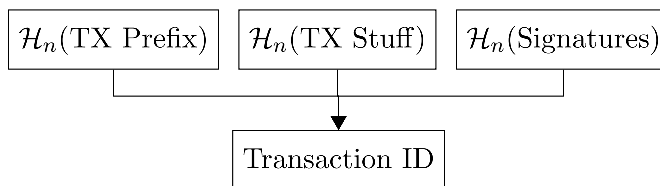
I Transaction ID includono, oltre ai dati firmati dagli input MLSAG (Sez. 6.2.2), anche le stesse firme MLSAG e le range proof.

L'hash viene calcolato attraverso:

- **TX Prefix** = {versione, inputs {key offsets, key images}, outputs {indirizzi one-time}, extra {chiave pubblica transazione, payment ID codificato, var.}}
- **TX Stuff** = {tipo firma (`RCTTypeNull` o `RCTTypeBulletproof2`), commissione, pseudo-commitments input, ecdhInfo, output commitments}
- **Signatures** = {MLSAG, range proofs}

src/cryptonote_basic/cryptonote_format_utils.cpp `calculate_transaction_hash()`

Diagramma gerarchico:



Nelle transazioni miner, al posto degli input si registra l'altezza del blocco. Così il Transaction ID della transazione miner (che è un normale Transaction ID sostituendo $\mathcal{H}_n$(Signatures) con 0) è sempre unico.

7.4.2 Albero di Merkle

Alcuni nodi potrebbero voler risparmiare spazio di archiviazione eliminando dati dalla propria copia della blockchain (es. dopo aver verificato range proof e firme, conservano solo $\mathcal{H}_n$(Signatures) per le Transaction ID).

Per facilitare la "potatura" (pruning) delle transazioni e organizzarle, si ricorre all'albero di Merkle [89], un albero binario di hash. Qualunque ramo può essere potato purché si abbia la radice.[46]

Esempio con quattro transazioni e una transazione di mining (Figura 7.1):

[46] Il pruning è stato introdotto nella versione 0.14.1 (marzo 2019, protocollo v10). I nodi completi, anche detti nodi *full*, possono eliminare le firme e le proof per 7/8 delle transazioni, mantenendo le $\mathcal{H}_n$(Signatures), riducendo i requisiti di archiviazione di circa 2/3. [52]

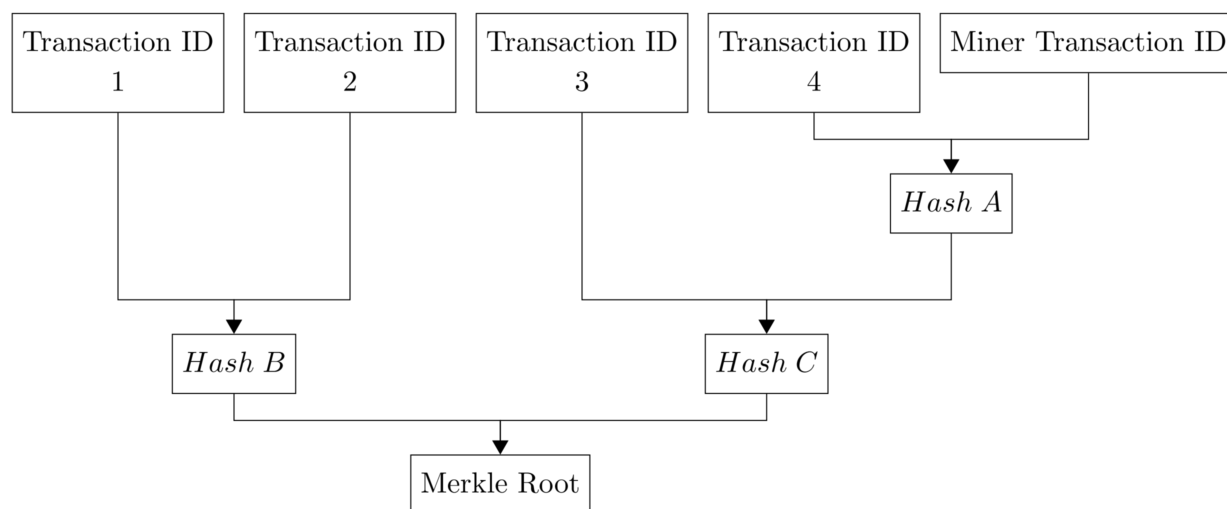


Figura 7.1: Merkle Tree

La radice di Merkle (Merkle Root) è quindi un riferimento a tutte le transazioni incluse nell'albero.

7.4.3 Blocchi

Un blocco è fondamentalmente una struttura dati composta da un'intestazione (header) e da una lista di transazioni. Le intestazioni registrano informazioni importanti su ciascun blocco. Le transazioni di un blocco possono essere riferite tramite la loro radice di Merkle. Di seguito presentiamo la struttura di massima del contenuto di un blocco. Un esempio reale di blocco è disponibile nell'Appendice B.

- Header del blocco:
  - **Major version**: utilizzata per tracciare le hard fork (modifiche al protocollo).
  - **Minor version**: un tempo usata per un sistema di voto, adesso svolge le stesse funzioni di major version.
  - **Timestamp**: orario UTC (Coordinated Universal Time) del blocco. Viene inserito dai miner; i timestamp non sono verificati ma non saranno accettati se inferiori al timestamp medio degli ultimi 60 blocchi.
  - **ID del blocco precedente**: riferimento al blocco precedente, caratteristica essenziale di una blockchain.
  - **Nonce**: un intero a 4 byte che i miner modificano ripetutamente affinché l'hash di Proof of Work soddisfi il target di difficoltà. I verificatori possono ricalcolare facilmente l'hash PoW.

src/cryptonote_core/blockchain.cpp `check_block_timestamp()`

- Transazione di mining: distribuisce la ricompensa del blocco e le commissioni al miner.
- ID delle transazioni: riferimenti alle transazioni diverse da quella del miner aggiunte da questo blocco alla blockchain. Gli ID delle transazioni, insieme all'ID della transazione del miner, possono essere usati per calcolare la radice di Merkle e per trovare le transazioni reali ovunque siano archiviate.

Oltre ai dati contenuti in ciascuna transazione (Sezione 6.3), sono memorizzate anche le seguenti informazioni:

- Major e minor version: interi variabili di lunghezza fino a 9 byte
- Timestamp: intero variabile fino a 9 byte
- ID del blocco precedente: 32 byte
- Nonce: 4 byte, la dimensione effettiva può essere estesa tramite il campo extra nonce nella sezione extra della transazione di mining[47].
- Transazione di mining: 32 byte per un indirizzo one-time, 32 byte per una chiave pubblica della transazione (+1 byte per il tag extra), e interi variabili per il tempo di sblocco, altezza del blocco corrispondente e l'importo. Dopo il download della blockchain, è inoltre necessario memorizzare 32 byte per il commitment dell'importo $C = 1G + aH$ (solo per gli importi post-RingCT della transazione di mining).
- ID delle transazioni: 32 byte ciascuno.

[47] All'interno di ogni transazione è presente un campo 'extra' che può contenere dati più o meno arbitrari. Se un miner necessita di un range di nonce più ampio di 4 byte, può aggiungere o modificare dati nel campo extra della propria transazione miner per estendere la dimensione del nonce. [79]

Parte II

Estensioni

CAPITOLO 8

Prove di Conoscenza Relative alle Transazioni Monero

Monero è una valuta e, come qualsiasi valuta, i suoi usi possono essere complessi. Dalla contabilità aziendale, al mercato di scambio, all'arbitrato legale, diverse parti possono essere interessate e voler conoscere informazioni dettagliate sulle transazioni effettuate.

Com'è possibile stabilire con certezza che il denaro ricevuto proviene da una persona specifica? O dimostrare di aver effettivamente inviato un certo output o transazione a qualcuno nonostante affermazioni contrarie? I mittenti e i destinatari nel registro pubblico di Monero sono *opachi*. Come si può dimostrare di possedere una certa somma di denaro, senza compromettere le chiavi private? Gli importi in Monero sono completamente nascosti agli osservatori.

In questo capitolo verranno trattati metodi per produrre diverse tipologie di prove sulle transazioni, alcune delle quali sono implementate in Monero e disponibili con strumenti integrati nel wallet. Inoltre, verrà presentato un quadro di riferimento per verificare il saldo completo posseduto da una persona o organizzazione, senza richiedere la divulgazione di informazioni sulle transazioni future che potrebbero effettuare.

8.1 Prove di Transazione Monero

Le *prove di transazione* Monero sono in continua fase di sviluppo e aggiornamento [102]. Le prove attualmente implementate fanno tutte riferimento alla 'versione 1', e non includono la separazione del dominio. In questo documento saranno descritte solo le prove di transazione più avanzate, siano esse già implementate, previste per le prossime versioni [103], o ipotetiche implementazioni che potrebbero essere attuate (Sezioni 8.1.5 [128] e 8.2.1).

8.1.1 Prove di Transazione Monero Multi-Base

Ci sono alcuni dettagli di cui è necessario essere consapevoli procedendo nella lettura di questo capitolo. La maggior parte delle prove di transazione Monero coinvolge l'utilizzo di prove multi-base (Sezione 3.1). Dove necessario, il separatore di dominio sarà specificato come $T_{txprf2} = \mathcal{H}_n(\text{“TXPROOF_V2”})$.[1] Il messaggio firmato è solitamente (a meno che non sia specificato diversamente) $\mathfrak{m} = \mathcal{H}_n(\texttt{tx_hash, message})$, dove `tx_hash` è l'ID della transazione presa in considerazione (Sezione 7.4.1), e `message` è un messaggio opzionale che dimostratori o terze parti possono fornire per assicurare la legittimità dell'autore della prova, ovvero che non sia *rubata*.

src/wallet/wallet2.cpp `get_tx_proof()`

Le prove sono codificate in base-58, uno schema di codifica binario-testo (binary-to-text) introdotto per la prima volta per Bitcoin [25]. La verifica di queste prove prevede sempre prima la decodifica dalla base-58 al formato binario. Si noti che i verificatori necessitano anche dell'accesso alla blockchain, così da poter usare i riferimenti all'identificativo della transazione per ottenere informazioni come gli indirizzi one-time.[2]

La struttura del prefisso delle chiavi nelle prove di transazione è piuttosto sgangherata, dovuto in parte all'accumulo di aggiornamenti non ancora riorganizzati. Le sfide per le prove *2-base* riferite alla 'versione 2' sono assemblate nel seguente formato:

$$c = \mathcal{H}_n(\mathfrak{m}, \text{public key 2, proof part 1, proof part 2, } T_{txprf2}, \text{public key 1, base key 2, base key 1})$$

Se la 'base key 1' corrisponde a G allora la sua posizione nella sfida viene riempita con 32 byte a zero.

8.1.2 Prova di Creazione di Input di Transazione (`SpendProofV1`)

Si supponga di voler dimostrare di aver effettuato una transazione. Chiaramente, riproducendo la firma di un input di transazione su un nuovo messaggio, un verificatore non potrebbe dubitare in alcun modo della legittimità del messaggio originale. Riprodurre *tutte* le firme degli input di una transazione significa che l'iniziatore deve aver creato l'intera transazione (Sezione 6.2.2), o almeno averla finanziata completamente.[3]

Una cosiddetta *prova di spesa* (SpendProof) contiene le firme riprodotte per tutti gli input di una transazione. È importante tenere conto che le firme ad anello delle SpendProof riutilizzano i membri originali per evitare di identificare il vero firmatario tramite intersezioni dell'anello.

src/wallet/wallet2.cpp `get_spend_proof()`

Le SpendProof sono implementate in Monero, e al fine di codificarne una per la trasmissione ai verificatori, il dimostratore concatena la stringa prefisso "`SpendProofV1`" con la lista delle firme. Si noti che la stringa prefisso non è in base-58 e non deve essere codificata/decodificata, poiché il suo scopo è dedito alla leggibilità umana.

[1] Come nella Sezione 3.6, le funzioni hash dovrebbero essere separate per dominio tramite prefissi etichettati. L'implementazione attuale delle prove di transazione Monero non ha separazione di dominio, quindi tutti i tag in questo capitolo descrivono caratteristiche *non* ancora implementate.

[2] Gli ID delle transazioni sono solitamente comunicati separatamente dalle prove.

[3] Come vedremo nel Capitolo 11, chi effettua la firma di un input non necessariamente ha prodotto tutte le firme degli input.

Le SpendProof

Inaspettatamente, le SpendProof non usano MLSAG, ma piuttosto lo schema originale di ring signature di Monero che veniva usato nel primissimo protocollo di transazione (pre-RingCT) [134]. Di seguito sono elencati i passaggi per produrre una prova di spesa:

src/crypto/crypto.cpp `generate_ring_signature()`

1. Calcolare l'immagine della chiave $\tilde{K} = k^o_\pi \mathcal{H}_p(K^o_\pi)$.
2. Generare un numero casuale $\alpha \in_R \mathbb{Z}_l$ e numeri casuali $c_i, r_i \in_R \mathbb{Z}_l$ per $i \in \{1, 2, ..., n\}$ escludendo però $i = \pi$.
3. Calcolare
$$c_{tot} = \mathcal{H}_n(\mathfrak{m}, [r_1 G + c_1 K^o_1], [r_1 \mathcal{H}_p(K^o_1) + c_1 \tilde{K}], ..., [\alpha G], [\alpha \mathcal{H}_p(K^o_\pi)], ..., \text{ecc.})$$
4. Definire la sfida reale
$$c_\pi = c_{tot} - \sum_{i=1, i \neq \pi}^{n} c_i$$
5. Definire $r_\pi = \alpha - c_\pi * k^o_\pi \pmod{l}$.

La firma sarà dunque $\sigma = (c_1, r_1, c_2, r_2, ..., c_n, r_n)$.

Verifica

Per verificare una SpendProof su una transazione specifica, il verificatore deve controllare che tutte le ring signature siano valide usando le informazioni trovate nella transazione presa in considerazione (ad esempio immagini chiave e offset degli output per ottenere gli indirizzi one-time da altre transazioni).

src/wallet/wallet2.cpp `check_spend_proof()`

1. Calcolare
$$c_{tot} = \mathcal{H}_n(\mathfrak{m}, [r_1 G + c_1 K^o_1], [r_1 \mathcal{H}_p(K^o_1) + c_1 \tilde{K}], ..., [r_n G + c_n K^o_n], [r_n \mathcal{H}_p(K^o_n) + c_n \tilde{K}])$$
2. Verificare che
$$c_{tot} \stackrel{?}{=} \sum_{i=1}^{n} c_i$$

Perché Funziona

Si noti come questo schema sia identico al bLSAG (Sezione 3.4) quando vi è un solo membro nell'anello. Per aggiungere un membro fittizio, invece di passare la sfida $c_{\pi+1}$ per creare un nuovo hash di sfida, il membro viene aggiunto all'hash originale. Poiché l'equazione seguente

$$c_s = c_{tot} - \sum_{i=1, i \neq s}^{n} c_i$$

vale banalmente per ogni indice s, un verificatore non avrà modo di identificare la vera sfida. Inoltre, senza la conoscenza di k^o_π, il dimostratore non sarebbe mai stato in grado di definire correttamente r_π (tranne che con probabilità trascurabile).

8.1.3 Prova di Creazione di Output di Transazione (`OutProofV2`)

Si supponga di aver inviato denaro a qualcuno (un output) e di volerlo dimostrare. Gli output di una transazione contengono essenzialmente tre campi: l'indirizzo del destinatario, la quantità inviata e la chiave privata della transazione. Le quantità sono cifrate, quindi sono necessari solo l'indirizzo e la chiave privata della transazione per cominciare. Chiunque cancelli o perda la propria chiave privata della transazione non potrà creare una OutProof, quindi in questo senso le OutProof sono le prove di transazione Monero meno affidabili.[4]

L'obiettivo è dimostrare che l'indirizzo one-time è stato creato a partire dall'indirizzo del destinatario, e permettere ai verificatori di ricostruire l'impegno sull'output. È possibile produrre questa prova fornendo il segreto condiviso mittente-destinatario rK^v, poi dimostrando di averlo creato e che corrisponde alla chiave pubblica della transazione e all'indirizzo del destinatario firmando una firma 2-base (Sezione 3.1) sulle chiavi base G e K^v. I verificatori possono usare il segreto condiviso per controllare il destinatario (Sezione 4.2), decodificare la quantità (Sezione 5.3) e ricostruire l'impegno dell'output (Sezione 5.3). In questa sezione sono forniti tutti i dettagli che riguardano indirizzi standard e sottoindirizzi.

src/wallet/wallet2.cpp `check_tx_proof()`

Le OutProof

Di seguito sono elencati i passaggi da effettuare al fine di produrre una prova per un output diretto a un indirizzo (K^v, K^s) o sottoindirizzo $(K^{v,i}, K^{s,i})$, con chiave privata della transazione r, dove il segreto condiviso mittente-destinatario è rK^v. Si ricorda che la chiave pubblica della transazione viene memorizzata nei dati della transazione, dunque corrisponde a rG oppure a $rK^{s,i}$ a seconda che la destinazione sia o meno un sottoindirizzo (Sezione 4.3).

src/crypto/crypto.cpp `generate_tx_proof()`

1. Generare un numero casuale $\alpha \in_R \mathbb{Z}_l$, e calcolare
   (a) *Indirizzo normale*: αG e αK^v
   (b) *Sottoindirizzo*: $\alpha K^{s,i}$ e $\alpha K^{v,i}$
2. Calcolare la sfida
   (a) *Indirizzo normale*:[5]
   $$c = \mathcal{H}_n(\mathfrak{m}, [rK^v], [\alpha G], [\alpha K^v], [T_{txprf2}], [rG], [K^v], [0])$$

[4] Possiamo pensare a una 'OutProof' come alla dimostrazione che un output è 'uscente' dal dimostratore. Le corrispondenti 'InProof' (Sezione 8.1.4) mostrano output 'entranti' all'indirizzo del dimostratore.

[5] Qui il valore '0' è una codifica di 32 byte di zeri.

(b) *Sottoindirizzo*:

$$c = \mathcal{H}_n(\mathfrak{m}, [rK^{v,i}], [\alpha K^{s,i}], [\alpha K^{v,i}], [T_{txprf2}], [rK^{s,i}], [K^{v,i}], [K^{s,i}])$$

3. Definire la risposta[6] $r^{resp} = \alpha - c * r$.

4. La firma prodotta è $\sigma^{outproof} = (c, r^{resp})$.

Un dimostratore può generare una serie di OutProof ed inviarle successivamente tutte insieme a un verificatore. Egli concatena la stringa prefisso "`OutProofV2`" con una lista di prove, dove ogni elemento (codificato in base-58) consiste nel segreto condiviso tra mittente e destinatario rK^v (o $rK^{v,i}$ per un sottindirizzo), e la sua corrispondente $\sigma^{outproof}$. Supponendo che il verificatore conosca l'indirizzo appropriato per ciascuna prova, la verifica si svolge come segue:

src/wallet/wallet2.cpp `get_tx_proof()`

Verifica

1. Calcolare la sfida

src/crypto/crypto.cpp `check_tx_proof()`

(a) *Indirizzo normale*:

$$c' = \mathcal{H}_n(\mathfrak{m}, [rK^v], [r^{resp}G + c * rG], [r^{resp}K^v + c * rK^v], [T_{txprf2}], [rG], [K^v], [0])$$

(b) *Sottindirizzo*:

$$c' = \mathcal{H}_n(\mathfrak{m}, [rK^{v,i}], [r^{resp}K^{s,i}+c*rK^{s,i}], [r^{resp}K^{v,i}+c*rK^{v,i}], [T_{txprf2}], [rK^{s,i}], [K^{v,i}], [K^{s,i}])$$

2. Se $c = c'$ allora il dimostratore conosce r, e rK^v è un segreto condiviso legittimo tra rG e K^v (eccetto che con probabilità trascurabili).

3. Il verificatore deve verificare che l'indirizzo del destinatario fornito possa essere utilizzato per creare un indirizzo one-time dalla transazione presa in considerazione (è lo stesso calcolo per indirizzi normali e sottindirizzi)

$$K^s \stackrel{?}{=} K_t^o - \mathcal{H}_n(rK^v, t)$$

src/wallet/wallet2.cpp `check_tx_key_helper()`

4. Deve anche decifrare l'ammontare dell'output b_t, calcolare la maschera dell'output y_t, e tentare di ricostruire il corrispondente commitment dell'output[7]

$$C_t^b \stackrel{?}{=} y_tG + b_tH$$

[6] A causa del numero limitato di simboli disponibili, è stata usato con dispiacere la dicitura r sia per le risposte che per la chiave privata della transazione. Il pedice 'resp' per 'response' sarà usato per differenziare i due quando necessario.

[7] Una firma OutProof valida non implica necessariamente che il destinatario considerato sia il vero destinatario. Un dimostratore malevolo potrebbe generare una chiave di visualizzazione casuale K'^v, calcolare $K'^s = K^o - \mathcal{H}_n(rK'^v, t) * G$, e fornire (K'^v, K'^s) come destinatario nominale. Ricalcolando il commitment dell'output, i verificatori possono essere più sicuri che l'indirizzo del destinatario in questione sia legittimo. Tuttavia, dimostratore e destinatario potrebbero collaborare per codificare il commitment usando K'^v, mentre l'indirizzo one-time usa (K^v, K^s). Poiché il destinatario dovrebbe conoscere la chiave privata k'^v (supponendo che l'output sia ancora spendibile), l'utilità di tale inganno è discutibile. Perché il destinatario non userebbe direttamente (K'^v, K'^s) (o un altro indirizzo monouso) per tutto l'output? Poiché il calcolo di C_t^b è legato al destinatario, consideriamo adeguato il processo di verifica OutProof descritto. In altre parole, il dimostratore non può ingannare i verificatori senza coordinarsi con il destinatario.

8.1.4 Dimostrare la Proprietà di un Output (`InProofV2`)

Una OutProof dimostra che l'autore di una transazione ha inviato un output a un indirizzo, mentre un InProof dimostra che un output è stato ricevuto a un certo indirizzo. È essenzialmente l'altra 'faccia della medaglia' del segreto condiviso tra mittente e destinatario rK^v. Questa volta l'utente dimostra la conoscenza di k^v relativa a K^v, e, in combinazione con la chiave pubblica della transazione rG, costruisce il segreto condiviso $k^v * rG$.

Una volta che un verificatore ha rK^v, può controllare se l'indirizzo one-time corrispondente è di proprietà dell'indirizzo del dimostratore con $K^o - \mathcal{H}_n(k^v * rG, t) * G \stackrel{?}{=} K^s$ (Sezione 4.2.1). Producendo una InProof per tutte le chiavi pubbliche di transazione sulla blockchain, un dimostratore rivelerà tutti gli output di sua proprietà.

src/wallet/wallet2.cpp `check_tx_proof()`

Dare direttamente la chiave di visualizzazione a un verificatore porterebbe allo stesso risultato voluto, ma una volta ottenuta la chiave, il verificatore sarebbe in grado di identificare la proprietà di output da creare in futuro. Con gli InProof il dimostratore può mantenere il controllo delle sue chiavi private, al costo del tempo necessario a dimostrare (e poi verificare) ogni output come posseduto o meno.

L'InProof

Una InProof si costruisce nello stesso modo di una OutProof, cambiano solo le chiavi base $\mathcal{J} = \{G, rG\}$, le chiavi pubbliche $\mathcal{K} = \{K^v, rK^v\}$, e la chiave di firma che adesso è k^v invece di r. Mostreremo solo il passo di verifica per chiarire il significato. Si noti che l'ordine del prefisso delle chiavi cambia (rG e K^v si scambiano di posto) per coincidere con il diverso ruolo di ciascuna chiave.

src/crypto/crypto.cpp `generate_tx_proof()`

Più di una InProof, relative a vari output posseduti dallo stesso indirizzo, possono essere inviate tutte insieme al verificatore. Sono preceduti dalla stringa "`InProofV2`", e ogni elemento (codificato in base-58) contiene il segreto condiviso mittente-destinatario rK^v (o $rK^{v,i}$), e la corrispondente $\sigma^{inproof}$.

src/wallet/wallet2.cpp `get_tx_proof()`

Verifica

src/crypto/crypto.cpp `check_tx_proof()`

1. Calcolare la sfida
   (a) *Indirizzo normale*:
   $$c' = \mathcal{H}_n(\mathfrak{m}, [rK^v], [r^{resp}G + c * K^v], [r^{resp} * rG + c * k^v * rG], [T_{txprf2}], [K^v], [rG], [0])$$
   (b) *Sottindirizzo*:
   $$c' = \mathcal{H}_n(\mathfrak{m}, [rK^{v,i}], [r^{resp}K^{s,i} + c * K^{v,i}], [r^{resp} * rK^{s,i} + c * k^v * rK^{s,i}], [T_{txprf2}], [K^{v,i}], [rK^{s,i}], [K^{s,i}])$$
2. Se $c = c'$ allora il dimostratore conosce k^v, e $k^v * rG$ è un segreto condiviso legittimo tra K^v e rG (salvo probabilità trascurabili).

Dimostrare la Proprietà Completa con la Chiave dell'Indirizzo One-Time

Mentre una InProof dimostra che un indirizzo one-time è stato costruito con un indirizzo specifico (salvo probabilità trascurabili), ciò non significa necessariamente che il dimostratore possa *spendere* il relativo output. Solo chi può spendere un output ne è effettivamente proprietario.

Dimostrare la proprietà, una volta completata una InProof, è semplice come firmare un messaggio con la chiave di spesa.[8]

8.1.5 Dimostrare che un Output Posseduto Non è Stato Speso in una Transazione (UnspentProof)

Potrebbe sembrare che dimostrare se un output è stato speso o meno sia semplice come ricreare la sua key image con una prova multi-base su $\mathcal{J} = \{G, \mathcal{H}_p(K^o)\}$ e $\mathcal{K} = \{K^o, \tilde{K}\}$. Sebbene ciò funzioni questo comporta che i verificatori devono conoscere la key image, il che rivela anche quando un output non speso viene speso *in futuro*.

In realtà è possibile dimostrare che un output non è stato speso in una specifica transazione senza rivelare la key image. Inoltre, è possibile dimostrare che è attualmente non speso *punto e basta*, estendendo questa UnspentProof [128] a ≪tutte le transazioni in cui (l'output) è stato incluso come membro dell'anello≫.[9]

Più precisamente, la UnspentProof afferma che una data key image di una transazione sulla blockchain corrisponde, o meno, a uno specifico indirizzo one-time appartenente al suo anello corrispondente. Per inciso, come vedremo, le UnspentProof vanno di pari passo con le InProof.

Preparazione di una UnspentProof

Il verificatore di una UnspentProof deve conoscere rK^v, il segreto condiviso mittente-destinatario per un output posseduto dato con indirizzo one-time K^o e la chiave pubblica della transazione rG. Egli o conosce la chiave di visualizzazione k^v, che gli ha permesso di calcolare $k^v * rG$ e verificare che $K^o - \mathcal{H}_n(k^v * rG, t) * G \stackrel{?}{=} K^s$, quindi sa che l'output testato appartiene al dimostratore (ricordare la Sezione 4.2), oppure il dimostratore ha fornito rK^v. Qui entrano in gioco le InProof, poiché con una InProof il verificatore può essere sicuro che rK^v provenga legittimamente dalla chiave di visualizzazione del dimostratore e corrisponda a un output posseduto, senza apprendere la chiave privata di visualizzazione.

Prima di verificare una UnspentProof, il verificatore apprende la key image da testare $\tilde{K}_?$ e controlla che il suo anello corrispondente includa l'indirizzo one-time K^o dell'output posseduto dal dimostratore. Quindi calcola l'immagine parziale di 'spesa' $\tilde{K}^s_?$.

$$\tilde{K}^s_? = \tilde{K}_? - \mathcal{H}_n(rK^v, t) * \mathcal{H}_p(K^o)$$

Se la key image testata è stata creata da K^o allora il punto risultante sarà $\tilde{K}^s_? = k^s * \mathcal{H}_p(K^o)$.

[8] La possibilità di fornire una firma del genere direttamente non sembra essere disponibile in Monero, anche se, come vedremo, le ReserveProof (Sezione 8.1.6) le includono.

[9] Le UnspentProof non sono state implementate in Monero.

Le UnspentProof

Il dimostratore crea due prove multi-base (ricordare la Sezione 3.1). Il suo indirizzo, che possiede l'output in questione, è (K^v, K^s) oppure $(K^{v,i}, K^{s,i})$.[10]

1. Una prova a 3 basi, dove la chiave di firma è k^s, e
$$\begin{aligned} \mathcal{J}_3^{unspent} &= \{[G], [K^s], [\tilde{K}^s_?]\} \\ \mathcal{K}_3^{unspent} &= \{[K^s], [k^s * K^s], [k^s * \tilde{K}^s_?]\} \end{aligned}$$

2. Una prova a 2 basi, dove la chiave di firma è $k^s * k^s$, e
$$\begin{aligned} \mathcal{J}_2^{unspent} &= \{[G], [\mathcal{H}_p(K^o)]\} \\ \mathcal{K}_2^{unspent} &= \{[k^s * K^s], [k^s * k^s * \mathcal{H}_p(K^o)]\} \end{aligned}$$

Insieme alle prove $\sigma_3^{unspent}$ e $\sigma_2^{unspent}$, il dimostratore si assicura di comunicare le chiavi pubbliche $k^s * K^s$, $k^s * \tilde{K}^s_?$ e $k^s * k^s * \mathcal{H}_p(K^o)$.

Verifica

1. Verificare che $\sigma_3^{unspent}$ e $\sigma_2^{unspent}$ siano legittime.
2. Assicurarsi che la stessa chiave pubblica $k^s * K^s$ sia stata usata in entrambe le prove.
3. Verificare se $k^s * \tilde{K}^s_?$ e $k^s * k^s * \mathcal{H}_p(K^o)$ siano uguali. Se lo sono, l'output è speso, e se no è non speso (eccetto con probabilità trascurabile).

Perché Funziona

Questo approccio apparentemente tortuoso impedisce al verificatore di apprendere $k^s * H_p(K^o)$ per un output non speso, che potrebbe usare in combinazione con rK^v per calcolare la sua immagine chiave reale, pur lasciandolo fiducioso che l'immagine chiave testata non corrisponda a quell'output.

La prova $\sigma_2^{unspent}$ può essere riutilizzata per qualsiasi numero di UnspentProof che coinvolgono lo stesso output, anche se se fosse stato effettivamente speso ne è necessaria solo una (cioè le UnspentProof possono essere usate anche per dimostrare che un output è speso). Produrre UnspentProof su tutte le firme ad anello in cui un dato output non speso è stato referenziato non dovrebbe essere computazionalmente costoso. È probabile che con il passare del tempo, un output, venga incluso come esca in circa 11 (dimensione attuale dell'anello) anelli diversi.

[10] Le UnspentProof si costruiscono allo stesso modo sia per sottindirizzi che per indirizzi normali. È richiesta la chiave completa di spesa di un sottindirizzo, ad esempio $k^{s,i} = k^s + \mathcal{H}_n(k^v, i)$ (Sezione 4.3).

8.1.6 Provare che un Indirizzo ha un Saldo Non Speso Minimo (`ReserveProofV2`)

Nonostante il problema di privacy che deriva dal rivelare l'immagine chiave di un output quando non è ancora stato speso, è comunque qualcosa in qualche modo utile e le *prove di riserva* sono state implementate in Monero [**?**] prima che le UnspentProof fossero inventate [128]. Le cosiddette 'ReserveProof' di Monero vengono utilizzate per dimostrare che un indirizzo possiede una quantità minima di denaro creando immagini chiave per alcuni output non spesi.

src/wallet/wallet2.cpp `get_reserve_proof()`

Più specificamente, dato un saldo minimo, il dimostratore trova abbastanza output non spesi per coprirlo, dimostra la proprietà con le InProof, crea immagini chiave per essi e dimostra che sono legittimamente basate su quegli output con prove a 2 basi (usando un prefisso della chiave diverso), e poi dimostra la conoscenza delle chiavi private di spesa utilizzate con normali firme Schnorr (potrebbero essercene più di una se alcuni output sono posseduti da sotto-indirizzi diversi). Un verificatore può controllare che le immagini chiave non siano apparse sulla blockchain, e quindi i loro output devono essere non spesi.

Le ReserveProof

Tutte le sotto-prove all'interno di una ReserveProof firmano un messaggio diverso rispetto ad altre prove (ad es. OutProof, InProof o SpendProof). Questa volta $\mathfrak{m} = \mathcal{H}_n(\texttt{message}, \texttt{address}, \tilde{K}^o_1, ..., \tilde{K}^o_n)$, dove `address` è la forma codificata (vedi [5]) dell'indirizzo standard del dimostratore (K^v, K^s), e le immagini chiave corrispondono agli output non spesi da includere nella prova.

1. Ogni output ha una InProof, che dimostra che l'indirizzo del dimostratore (o uno dei suoi sotto-indirizzi) possiede l'output.
2. L'immagine chiave di ogni output è firmata con una prova a 2 basi, dove la sfida è calcolata come segue:
$$c = \mathcal{H}_n(\mathfrak{m}, [rG + c * K^o], [r\mathcal{H}_p(K^o) + c * \tilde{K}])$$
src/crypto/crypto.cpp `generate_ring_signature()`
3. Ogni indirizzo (e sottoindirizzo) che possiede almeno un output ha una normale firma Schnorr (Sezione 2.3.5), e la sfida è simile a (è la stessa per indirizzi normali e sotto-indirizzi):
$$c = \mathcal{H}_n(\mathfrak{m}, K^{s,i}, [rG + c * K^{s,i}])$$
src/crypto/crypto.cpp `generate_signature()`

src/wallet/wallet2.cpp `get_reserve_proof()`

Per poter inviare una ReserveProof a qualcun altro, il dimostratore concatena la stringa prefisso "`ReserveProofV2`" con due liste codificate in base-58 (ad es. "`ReserveProofV2`, lista 1, lista 2"). Ogni elemento della lista 1 è correlato a un output specifico e contiene il relativo hash di transazione (Sezione 7.4.1), l'indice dell'output in quella transazione (Sezione 4.2.1), il segreto condiviso rK^v, la sua immagine chiave, la sua InProof $\sigma^{inproof}$, e la prova dell'immagine chiave. Gli elementi della lista 2 sono gli indirizzi che possiedono quegli output insieme alle relative firme Schnorr.

Verifica

1. Controllare che le immagini chiave della ReserveProof non siano apparse nella blockchain.
2. Verificare la InProof per ogni output, e che uno degli indirizzi forniti nella lista 2 possieda ciascuno di essi.
3. Verificare le firme dell'immagine chiave a 2 basi.
4. Usare i segreti condivisi mittente-destinatario per decodificare gli importi degli output (Sezione 5.3).
5. Controllare la firma di ogni indirizzo.

src/wallet/wallet2.cpp `check_reserve_proof()`

Se la verifica va a buon fine, allora il dimostratore dispone di un importo non speso, pari ad almeno l'importo totale contenuto negli output della ReserveProof (eccetto con probabilità trascurabile).[11]

8.2 Framework di Audit Monero

Negli USA la maggior parte delle aziende affronta audit annuali dei propri bilanci [127], che includono il conto economico, lo stato patrimoniale e il rendiconto finanziario. Di questi, i primi due riguardano in gran parte la contabilità interna di un'azienda, mentre l'ultimo riguarda ogni transazione che influisce su quanto denaro l'azienda ha attualmente. Le criptovalute sono denaro digitale, quindi qualsiasi audit del rendiconto finanziario di un utente di criptovaluta deve riguardare le transazioni memorizzate sulla blockchain.

Il primo compito di una persona sottoposta ad audit è identificare tutti gli output che possiede attualmente (spesi e non spesi). Questo può essere fatto con le InProof utilizzando tutti i suoi indirizzi. Una grande azienda potrebbe avere una moltitudine di sotto-indirizzi, specialmente i rivenditori che operano nei mercati online (vedi Capitolo 10). La creazione di InProof su tutte le transazioni per ogni singolo sottoindirizzo potrebbe comportare enormi requisiti computazionali e di archiviazione sia per i dimostratori che per i verificatori.

È possibile però creare InProof solo per gli indirizzi standard del dimostratore (su tutte le transazioni). L'auditor utilizza quei segreti condivisi mittente-destinatario per verificare se qualche output è posseduto dall'indirizzo principale del dimostratore o dai suoi sotto-indirizzi correlati. Richiamando la Sezione 4.3, la chiave di visualizzazione di un utente è sufficiente per identificare tutti gli output posseduti dai sotto-indirizzi di un indirizzo.

Per assicurarsi che il dimostratore non stia ingannando un auditor nascondendo l'indirizzo primario sotto ad alcuni dei suoi sotto-indirizzi, deve anche dimostrare che tutti i sotto-indirizzi corrispondono a uno dei suoi indirizzi standard conosciuti.

[11] Le ReserveProof, pur dimostrando la piena proprietà dei fondi, non includono prove che i dati sotto-indirizzi corrispondano effettivamente all'indirizzo standard del dimostratore.

8.2.1 Dimostrare la Corrispondenza tra Indirizzo e Sottoindirizzo (SubaddressProof)

Le SubaddressProof mostrano che la chiave di visualizzazione di un indirizzo primario può essere usata per identificare gli output posseduti da un dato sottoindirizzo.[12]

Le SubaddressProof

Le SubaddressProof possono essere create in modo molto simile alle OutProof e alle InProof. Le chiavi base sono $\mathcal{J} = \{G, K^{s,i}\}$, le chiavi pubbliche $\mathcal{K} = \{K^v, K^{v,i}\}$, e la chiave di firma è k^v. Ancora una volta, mostriamo solo il passo di verifica per chiarire il concetto.

Verifica

Un verificatore conosce l'indirizzo del dimostratore (K^v, K^s), il sottoindirizzo $(K^{v,i}, K^{s,i})$, e ha la SubaddressProof $\sigma^{subproof} = (c, r)$. Dopodiché, effettua i seguenti passaggi:

1. Calcolare la sfida
$$c' = \mathcal{H}_n(\mathfrak{m}, [K^{v,i}], [rG + c * K^v], [rK^{s,i} + c * K^{v,i}], [T_{txprf2}], [K^v], [K^{s,i}], [0])$$
2. Se $c = c'$ allora il dimostratore conosce k^v per K^v, e $K^{s,i}$ in combinazione con quella chiave di visualizzazione crea $K^{v,i}$ (eccetto con probabilità trascurabile).

8.2.2 Il Framework di Audit

Ora siamo pronti ad apprendere il più possibile sulla cronologia delle transazioni di un utente.[13]

1. Il dimostratore raccoglie un elenco di tutti i suoi account, dove ogni account consiste in un indirizzo normale e vari sotto-indirizzi. Crea SubaddressProof per tutti i sotto-indirizzi. Proprio come le ReserveProof, crea anche una firma con la chiave di spesa di ogni indirizzo e sottoindirizzo, dimostrando di avere i diritti di spesa su tutti gli output posseduti da quegli indirizzi.
2. Il dimostratore genera, per ciascuno dei suoi indirizzi standard, InProof su tutte le transazioni (ad es. tutte le chiavi pubbliche di transazione) nella blockchain. Questo rivela all'auditor tutti gli output posseduti dagli indirizzi del dimostratore poiché possono controllare tutti gli

[12] Le SubaddressProof non sono state implementate in Monero.

[13] Questo framework di audit non è completamente disponibile in Monero. Le SubaddressProof e le UnspentProof non sono implementate, le InProof non sono predisposte per l'ottimizzazione relativa ai sotto-indirizzi che abbiamo spiegato, e non esiste una vera struttura per ottenere o organizzare facilmente tutte le informazioni necessarie sia per i dimostratori che per i verificatori.

indirizzi monouso con i segreti condivisi mittente-destinatario. Possono essere sicuri che gli output posseduti dai sotto-indirizzi sono identificabili, grazie alle SubaddressProof.[14]

3. Il dimostratore genera, per ciascuno dei suoi output posseduti, UnspentProof su tutti gli input di transazione in cui appaiono come membri di un anello. Ora l'auditor conoscerà il saldo del dimostratore, e potrà indagare ulteriormente sugli output spesi.[15]

4. *Opzionale*: Il dimostratore genera, per ogni transazione in cui ha speso un output, una OutProof per rivelare all'auditor il destinatario e l'importo. Questo passaggio è possibile solo per le transazioni in cui il dimostratore ha salvato le chiavi private della transazione.

È importante notare che un dimostratore non ha modo di rivelare direttamente l'origine dei fondi. La sua unica risorsa è richiedere un insieme di prove alle persone che gli inviato tali fondi.

1. Per una transazione che invia fondi al dimostratore, il suo autore crea una SpendProof che dimostra di averla effettivamente inviata.

2. Il finanziatore del dimostratore crea anche una firma con una chiave pubblica identificativa, ad esempio la chiave di spesa del proprio indirizzo standard. Sia la SpendProof che questa firma dovrebbero firmare un messaggio contenente quella chiave pubblica identificativa, per assicurarsi che la SpendProof non sia stata rubata o che sia stata in realtà creata da qualcun altro.

[14] Questo passaggio può essere completato anche fornendo le chiavi private di visualizzazione, sebbene abbia ovvie implicazioni sulla privacy.

[15] In alternativa, potrebbe creare ReserveProof per tutti gli output posseduti. Ancora una volta, rivelare le immagini chiave degli output non spesi ha ovvie implicazioni sulla privacy.

CAPITOLO 9

Multifirme in Monero

In generale, le transazioni di criptovaluta non sono reversibili. Se un malintenzionato riesce ad entrare in possesso delle chiavi private o riesce a truffare un utente, il denaro potrebbe non essere più recuperabile. La distribuzione delle capacità di firma tra più persone può ridurre il potenziale pericolo causato da un malintenzionato.

Si supponga che un utente depositi del denaro in un conto corrente dotato di un servizio di alert fornito da una società di sicurezza che monitora attività sospette relative al suo account. Le transazioni possono essere autorizzate solo se vi è collaborazione tra l'utente e l'azienda nella costruzione della firma. Se si subisce il furto delle proprie chiavi, è possibile notificare la società del problema ed essa smetterà di firmare transazioni per l'account dell'utente. Questo tipo di servizio viene solitamente chiamato come 'deposito a garanzia' (escrow).[1]

Le criptovalute utilizzano una tecnica chiamata 'multifirma' per ottenere una firma congiunta attraverso la cosiddetta 'multisig M-su-N'. Nella modalità M-su-N, N persone possono cooperare per creare una chiave privata congiunta, e solo M persone ($M \leq N$) sono necessarie per firmare con quella chiave. Questo capitolo inizia introducendo le basi e in seguito i concetti più avanzati della firma multisig N-su-N, arrivando alla generalizzazione della multisig M-su-N. Si conclude con una spiegazione finale sulle tecniche di annidamento delle chiavi multisig all'interno di altre chiavi multisig.

[1] Le multifirme hanno una varietà di applicazioni reali, dai conti aziendali, abbonamenti a giornali ai mercati online.

Questo capitolo si incentrerà su come l'autore ha ritenuto che la firma multisig *debba* essere implementata. Questa interpretazione è basata sulle raccomandazioni fornite in [110] e su varie osservazioni di implementazioni ottimizzate. Le note a piè di pagina evidenziano a grandi linee quanto l'implementazione attuale si discosta da quanto descritto.[2] Questo documento estende la descrizione della multisig M-su-N nel dettaglio e fornisce un approccio innovativo all'annidamento delle chiavi multisig.

9.1 Comunicazione tra Co-Firmatari

La creazione di chiavi e transazioni congiunte richiede la comunicazione di informazioni riservate tra persone che potrebbero trovarsi in qualsiasi parte del mondo. Per mantenere tali informazioni al sicuro da osservatori esterni, i co-firmatari cifrano i messaggi che si inviano a vicenda.

Lo scambio Diffie-Hellman (ECDH) è un modo molto semplice per crittografare i messaggi utilizzando la crittografia a curva ellittica. Abbiamo già menzionato questo nella Sezione 5.3, dove gli importi degli output di Monero vengono comunicati ai destinatari tramite il segreto condiviso rK^v. Ovvero:

$$amount_t = b_t \oplus_8 \mathcal{H}_n(\text{``}amount\text{''}, \mathcal{H}_n(rK^v_B, t))$$

È possibile estendere facilmente questo concetto a qualsiasi tipologia di messaggio. Per prima cosa è necessario codificare il messaggio sotto forma di stringa di bit, in seguito si procede con la divisione in blocchi di dimensioni pari all'output di $\mathcal{H}_n$. Dopodiché, è necessario generare un numero casuale $r \in \mathbb{Z}_l$ ed eseguire uno scambio Diffie-Hellman su tutti i blocchi del messaggio utilizzando la chiave pubblica K del destinatario. Inviando quei blocchi crittografati insieme alla chiave pubblica rG, il destinatario può quindi decifrare il messaggio con il segreto condiviso krG. I mittenti del messaggio dovrebbero anche firmare il loro messaggio crittografato (o l'hash del messaggio crittografato per semplicità) in modo che i destinatari possano verificare che i messaggi non siano stati manomessi (una firma è verificabile solo sul messaggio $\mathfrak{m}$ per cui è stata generata).

Poiché la crittografia non è essenziale per comprendere il funzionamento di una criptovaluta come Monero, non è stato ritenuto necessario approfondire maggiori dettagli. I lettori curiosi possono consultare questa eccellente panoramica concettuale [4], o consultare una descrizione tecnica del popolare schema di crittografia AES qui [22]. Inoltre, il Dr. Bernstein ha sviluppato uno schema di crittografia noto come ChaCha [34, 98], utilizzato dall'implementazione iniziale di Monero per

src/wallet/wallet2.cpp `export_multisig()`

[2] Al momento della stesura di questo documento sono disponibili tre implementazioni della funzionalità multisig. La prima si tratta di un processo manuale molto basilare che utilizza la CLI (Command Line Interface) [113]. La seconda è MMS (Multisig Messaging System) che genera una multisig sicura e automatizzata tramite la CLI [114, 115]. La terza è il 'Exa Wallet', portafoglio di criptovalute commerciale, che ha il codice sorgente iniziale disponibile sul loro repository Github all'URL `https://github.com/exantech` (non sembra aggiornato con l'attuale versione di rilascio). Tutte e tre si basano sulla stessa codebase del team principale, il che significa essenzialmente che esiste una sola implementazione.

crittografare determinate informazioni sensibili relative ai portafogli degli utenti (come le immagini chiave per gli output posseduti).

src/wallet/ringdb.cpp

9.2 Aggregazione delle Chiavi per gli Indirizzi

9.2.1 Approccio Ingenuo

Si supponga che N persone vogliano creare un indirizzo multifirma di gruppo, che denotiamo come $(K^{v,grp}, K^{s,grp})$. I fondi possono essere inviati a quell'indirizzo proprio come a qualsiasi indirizzo standard, ma, come vedremo più avanti, per spendere quei fondi è necessaria la collaborazione di tutte le N persone per firmare e autorizzare le transazioni.

Poiché tutti gli N partecipanti devono poter essere in grado di visualizzare i fondi ricevuti dall'indirizzo di gruppo, possiamo far conoscere a tutti la chiave di visualizzazione di gruppo $k^{v,grp}$ (si rimanda alle Sezioni 4.1 e 4.2). Per conferire a tutti i partecipanti pari diritti, la chiave di visualizzazione di gruppo può essere una somma di componenti delle chiavi di visualizzazione che tutti i partecipanti si scambiano in modo sicuro. Dato il partecipante $e \in \{1, ..., N\}$ con rispettiva componente base della chiave di visualizzazione $k_e^{v,base} \in_R \mathbb{Z}_l$, tutti i partecipanti possono calcolare la chiave privata di visualizzazione di gruppo come segue:

$$k^{v,grp} = \sum_{e=1}^{N} k_e^{v,base}$$

src/multisig/multisig.cpp `generate_multisig_view_secret_key()`

In modo analogo, la chiave di spesa di gruppo $K^{s,grp} = k^{s,grp}G$ potrebbe essere anch'essa una somma di componenti base di chiavi private di spesa. Tuttavia, se qualcuno conosce tutte le componenti della chiavi private di spesa, allora potrà calcolare facilmente l'intera chiave privata di spesa di gruppo. Se si aggiunge anche la chiave privata di visualizzazione, quella persona potrà anche firmare transazioni autonomamente. In questo caso non si tratterebbe quindi di una firma multisig, ma semplicemente di una firma tradizionale.

Otteniamo lo stesso effetto se la chiave di spesa di gruppo è una somma di chiavi pubbliche di spesa. Supponiamo che i partecipanti abbiano chiavi pubbliche di spesa base $K_e^{s,base}$ che si inviano in modo sicuro. Ora facciamo in modo che ognuno di essi calcoli:

$$K^{s,grp} = \sum_{e} K_e^{s,base}$$

src/multisig/multisig.cpp `generate_multisig_N_N()`

Chiaramente questo equivale a calcolare:

$$K^{s,grp} = (\sum_{e} k_e^{s,base}) * G$$

9.2.2 Svantaggi dell'Approccio Ingenuo

L'uso di una somma di chiavi pubbliche di spesa è intuitivo e apparentemente semplice, ma porta a scontrarsi con un paio di problemi.

Test di Aggregazione delle Chiavi

Un osservatore esterno che conosce tutte le chiavi pubbliche di spesa base $K_e^{s,base}$ può facilmente testare l'aggregazione delle chiavi su un dato indirizzo pubblico (K^v, K^s) calcolando $K^{s,grp} = \sum_e K_e^{s,base}$ ed in seguito verificando che $K^s \stackrel{?}{=} K^{s,grp}$. Questo si lega a un requisito più generale che prevede che le chiavi aggregate siano indistinguibili dalle chiavi normali, in modo tale da evitare che gli osservatori esterni siano in grado di ottenere informazioni sulle attività degli utenti in base al tipo di indirizzo che pubblicano.[3]

Possiamo aggirare questo problema creando nuove chiavi di spesa base per ogni indirizzo multifirma, o mascherando le vecchie chiavi. Il primo caso è facile, ma può risultare scomodo.

Il secondo caso si svolge nel seguente modo: data la vecchia coppia di chiavi del partecipante e, (K_e^v, K_e^s), con chiavi private (k_e^v, k_e^s) e maschere casuali μ_e^v, μ_e^s,[4] i nuovi componenti base della chiave privata dell'indirizzo del gruppo sono definiti come:

$$k_e^{v,\text{base}} = \mathcal{H}_n(k_e^v, \mu_e^v)$$
$$k_e^{s,\text{base}} = \mathcal{H}_n(k_e^s, \mu_e^s)$$

Se i partecipanti non vogliono che gli osservatori raccolgano le nuove chiavi e ne testino l'aggregazione, è necessario comunicare i loro nuovi componenti chiave in modo sicuro.[5]

Se i test di aggregazione non costituiscono un problema, i partecipanti possono pubblicare le loro componenti base della chiave pubblica $(K_e^{v,base}, K_e^{s,base})$ come indirizzi standard. Qualsiasi terza parte potrebbe quindi calcolare l'indirizzo del gruppo da quegli indirizzi individuali e inviare fondi ad esso, senza interagire con nessuno dei partecipanti dell'indirizzo multifirma [87].

Annullamento della Chiave

Se la chiave di spesa di gruppo è una somma di chiavi pubbliche, un partecipante disonesto che apprende in anticipo i componenti base della chiave di spesa dei suoi collaboratori potrebbe *annullarli.*

Ad esempio, si supponga che Alice e Bob vogliono creare un indirizzo di gruppo. Alice, in buona fede, comunica a Bob i suoi componenti chiave di base $(k_A^{v,base}, K_A^{s,base})$. Bob crea privatamente i suoi componenti chiave di base $(k_B^{v,base}, K_B^{s,base})$ ma non lo comunica subito ad Alice. Invece,

[3] Se almeno un partecipante onesto utilizza componenti selezionate casualmente da una distribuzione uniforme, allora le chiavi aggregate da una semplice somma sono indistinguibili [121] dalle chiavi normali.

[4] Le maschere casuali possono essere facilmente derivate da una password. Ad esempio, $\mu^s = \mathcal{H}_n(\text{password})$ e $\mu^v = \mathcal{H}_n(\mu^s)$. Oppure, come avviene in Monero, si possono mascherare le chiavi di spesa e di visualizzazione con una stringa fissa come ad esempio $\mu^s, \mu^v =$ "Multisig". Questo implica che Monero supporta solo una chiave di spesa base multisig per indirizzo normale, anche se in pratica rendere un portafoglio multisig comporta la perdita dell'accesso al portafoglio originale da parte dell'utente [113]. Gli utenti devono creare un nuovo portafoglio con il loro indirizzo normale per poter accedere ai fondi, a meno che la multisig non sia stata creata direttamente da un nuovo indirizzo normale.

src/multisig/multisig.cpp `get_multisig_blinded_secret_key()`

[5] Come vedremo nella Sezione 9.6, l'aggregazione delle chiavi non funziona sulla multisig M-su-N quando M < N a causa della presenza di segreti condivisi.

calcola $K'^{s,base}_B = K^{s,base}_B - K^{s,base}_A$ e invia ad Alice $(k^{v,base}_B, K'^{s,base}_B)$. Dunque, l'indirizzo multifirma di gruppo può essere calcolato come segue:

$$\begin{aligned} K^{v,grp} &= (k^{v,base}_A + k^{v,base}_B)G \\ &= k^{v,grp}G \\ K^{s,grp} &= K^{s,base}_A + K'^{s,base}_B \\ &= K^{s,base}_A + (K^{s,base}_B - K^{s,base}_A) \\ &= K^{s,base}_B \end{aligned}$$

Ciò porta alla creazione di un indirizzo di gruppo $(k^{v,grp}G, K^{s,base}_B)$ dove Alice conosce la chiave privata di visualizzazione di gruppo, mentre Bob conosce sia la chiave privata di visualizzazione *che* la chiave privata di spesa! Bob può firmare transazioni da solo, ingannando Alice, che crede che i fondi detenuti dall'indirizzo di gruppo possano essere spesi solo con il suo consenso.

È possibile risolvere questo problema richiedendo a ogni partecipante, prima di aggregare le chiavi, di creare una firma che dimostri di conoscere la propria chiave privata relativa alla componente della chiave di spesa [108].[6] Ciò risultava molto scomodo e soggetto ad errori di implementazione. Fortunatamente è disponibile un'alternativa solida.

9.2.3 Aggregazione Robusta delle Chiavi

Per resistere facilmente all'annullamento della chiave privata, apportiamo una piccola modifica all'aggregazione della chiave di spesa (lasciando invariata l'aggregazione della chiave di visualizzazione). Sia $\mathbb{S}^{base} = \{K^{s,base}_1, ..., K^{s,base}_N\}$ l'insieme dei componenti base della chiave di spesa dei N firmatari, ordinati secondo un determinato criterio (ad esempio, dal più piccolo al più grande numericamente, cioè lessicograficamente).[7] La *chiave di spesa aggregata robusta* è definita come segue: [110][8,9]

$$K^{s,grp} = \sum_e \mathcal{H}_n(T_{agg}, \mathbb{S}^{base}, K^{s,base}_e) K^{s,base}_e$$

In questo modo, se Bob cerca di annullare la chiave di Alice, si ritrova con un problema molto difficile da risolvere:

$$\begin{aligned} K^{s,grp} &= \mathcal{H}_n(T_{agg}, \mathbb{S}, K^s_A)K^s_A + \mathcal{H}_n(T_{agg}, \mathbb{S}, K'^s_B)K'^s_B \\ &= \mathcal{H}_n(T_{agg}, \mathbb{S}, K^s_A)K^s_A + \mathcal{H}_n(T_{agg}, \mathbb{S}, K'^s_B)K^s_B - \mathcal{H}_n(T_{agg}, \mathbb{S}, K'^s_B)K^s_A \\ &= [\mathcal{H}_n(T_{agg}, \mathbb{S}, K^s_A) - \mathcal{H}_n(T_{agg}, \mathbb{S}, K'^s_B)]K^s_A + \mathcal{H}_n(T_{agg}, \mathbb{S}, K'^s_B)K^s_B \end{aligned}$$

[6] L'attuale (e prima) specifica del multisig su Monero, resa disponibile nell'aprile 2018 [59] (con l'integrazione M-su-N successiva nell'ottobre 2018 [11]), utilizzava questa aggregazione ingenua delle chiavi, e richiedeva agli utenti di firmare le loro componenti della chiave di spesa.

[7] $\mathbb{S}^{base}$ deve essere ordinato coerentemente in modo tale che i partecipanti possano essere sicuri di eseguire l'hashing della stessa cosa.

[8] Si rimanda alla Sezione 3.6, le funzioni hash dovrebbero essere separate per dominio prefissandole con tag, ad esempio T_{agg} = "Multisig_Aggregation". Lasciamo fuori i tag per esempi come le firme di Schnorr della prossima sezione.

[9] È importante includere $\mathbb{S}^{base}$ negli hash di aggregazione per evitare attacchi sofisticati di cancellazione della chiave che coinvolgono la soluzione generalizzata di Wagner al paradosso del compleanno [135]. [137] [87]

src/wallet/wallet2.cpp `get_multisig_info()`

Lasciamo la frustrazione di Bob all'immaginazione del lettore.

Proprio come nell'approccio ingenuo, qualsiasi terza parte che conosce $\mathbb{S}^{base}$ e le corrispondenti chiavi pubbliche di visualizzazione può calcolare l'indirizzo del gruppo.

Poiché i partecipanti non hanno bisogno di dimostrare di conoscere le loro chiavi di spesa private, o di interagire prima di firmare le transazioni, l'aggregazione robusta delle chiavi descritta in questo documento soddisfa il cosiddetto *modello a chiave pubblica semplice*, dove "l'unico requisito è che ogni potenziale firmatario abbia una chiave pubblica"[87].[10]

Funzioni `premerge` e `merge`

Più formalmente, e per chiarezza in futuro, definiamo un'operazione detta `premerge`, che prende un insieme di chiavi base $\mathbb{S}^{base}$ e produce un insieme di chiavi di aggregazione $\mathbb{K}^{agg}$ di uguale dimensione, dove l'elemento e^{esimo} può essere espresso come segue:[11]

$$\mathbb{K}^{agg}[e] = \mathcal{H}_n(T_{agg}, \mathbb{S}^{base}, K_e^{s,base}) K_e^{s,base}$$

Le chiavi private di aggregazione k_e^{agg} sono usate nelle firme di gruppo.[12]

Vi è un'altra operazione detta `merge`, che prende le chiavi di aggregazione da `premerge` e costruisce la chiave di firma di gruppo (ad es. chiave di spesa per Monero):

$$K^{grp} = \sum_e \mathbb{K}^{agg}[e]$$

È possibile generalizzare queste funzioni per i casi (N-1)-su-N e M-su-N come descritto nella Sezione 9.6.2, e per il caso di multisig annidata come descritto nella Sezione 9.7.2.

9.3 Firme in Stile Schnorr con Soglia

Affinché una firma multisig funzioni è necessaria la collaborazione di più co-firmatari, quindi è possibile affermare che esiste una 'soglia' di firmatari al di sotto della quale la firma non può essere prodotta. Una firma multisig, o multifirma, con N partecipanti che richiede che tutte le N persone collaborino nella costruzione della firma, solitamente indicata come *multisig N-su-N*, ha una soglia pari a N. Più avanti estenderemo il concetto di soglia alla multisig M-su-N (M $\leq$ N) dove N partecipanti creano l'indirizzo del gruppo ma solo M persone sono necessarie per creare le firme.

Facciamo un passo indietro da Monero. Tutti gli schemi di firma in questo documento derivano dall'esempio di zkProof di Maurer [84], il quale permette una descrizione essenziale della forma delle firme multisig con soglia attraverso una semplice firma in stile Schnorr (si rimanda alla Sezione 2.3.5 per maggiori dettagli) [108].

[10] Come vedremo più avanti, l'aggregazione delle chiavi soddisfa il modello a chiave pubblica semplice solo per multisig N-su-N e 1-su-N.

[11] Nota: $\mathbb{K}^{agg}[e]$ è l'e^{esimo} elemento dell'insieme.

[12] L'aggregazione robusta delle chiavi non è ancora stata implementata in Monero, ma poiché i partecipanti possono memorizzare e usare la chiave privata k_e^{agg} (per l'aggregazione ingenua delle chiavi, $k_e^{agg} = k_e^{base}$), l'aggiornamento di Monero per usare l'aggregazione robusta delle chiavi cambierà solo il processo di premerge.

Firma

Si supponga che ci sia un numero N di persone, dove ogni persona $e \in \{1, ..., N\}$ dispone di una chiave pubblica nell'insieme $\mathbb{K}^{agg}$ e conosce la chiave privata k_e^{agg}. La loro chiave pubblica di gruppo N-su-N, che useranno per firmare i messaggi, è denotata con K^{grp}. Si supponga inoltre che queste N persone vogliano firmare congiuntamente un messaggio $\mathfrak{m}$. Potrebbero avviare una collaborazione su una firma in stile Schnorr di base come segue:

1. Ogni partecipante $e \in \{1, ..., N\}$ effettua i seguenti passaggi:
   (a) sceglie una componente casuale $\alpha_e \in_R \mathbb{Z}_l$,
   (b) calcola $\alpha_e G$
   (c) si impegna sul valore $C_e^\alpha = \mathcal{H}_n(T_{com}, \alpha_e G)$,
   (d) e invia C_e^α agli altri partecipanti in modo sicuro.
2. Una volta raccolti tutti gli impegni C_e^α, ogni partecipante invia il proprio $\alpha_e G$ agli altri partecipanti in modo sicuro. Ogni partecipante deve verificare che $C_e^\alpha \stackrel{?}{=} \mathcal{H}_n(T_{com}, \alpha_e G)$ sia valido per tutti gli altri partecipanti.
3. Ogni partecipante calcola
$$\alpha G = \sum_e \alpha_e G$$
4. Ogni partecipante $e \in \{1, ..., N\}$ effettua i seguenti passaggi:[13]
   (a) calcola la sfida $c = \mathcal{H}_n(\mathfrak{m}, [\alpha G])$,
   (b) definisce la propria componente di risposta $r_e = \alpha_e - c * k_e^{agg} \pmod{l}$,
   (c) e invia r_e agli altri partecipanti in modo sicuro.
5. Ogni partecipante calcola
$$r = \sum_e r_e$$
6. Qualsiasi partecipante può pubblicare la firma $\sigma(\mathfrak{m}) = (c, r)$.

Verifica

Dati K^{grp}, $\mathfrak{m}$, e $\sigma(\mathfrak{m}) = (c, r)$, la verifica della firma può essere attuamente come segue:

1. Calcola la sfida $c' = \mathcal{H}_n(\mathfrak{m}, [rG + c * K^{grp}])$.
2. Se $c = c'$ allora la probabilità che la firma non sia legittima è trascurabile.

[13] Come nella Sezione 2.3.4, è importante non riutilizzare α_e per diverse sfide c. Ciò significa che per resettare un processo multifirma in cui le risposte sono state inviate, i partecipanti dovrebbero ricominciare dall'inizio con nuovi valori α_e.

Abbiamo incluso l'apice *grp* per chiarezza, ma in realtà il verificatore non ha modo di sapere che K^{grp} è una chiave congiunta a meno che un partecipante non glielo dica, o a meno che non conosca le componenti della chiave base o di aggregazione.

Perché Funziona

La risposta r è il cuore della firma congiunta. Il partecipante e conosce due segreti in r_e (α_e e k_e^{agg}), quindi la sua chiave privata k_e^{agg} è teoricamente sicura dagli altri partecipanti (supponendo che non riutilizzi mai α_e). Inoltre, i verificatori usano la chiave pubblica di gruppo K^{grp}, quindi tutti i componenti della chiave sono necessari per la costruzione di firme.

$$\begin{aligned}
rG &= (\sum_e r_e)G \\
&= (\sum_e (\alpha_e - c * k_e^{agg}))G \\
&= (\sum_e \alpha_e)G - c * (\sum_e k_e^{agg})G \\
&= \alpha G - c * K^{grp} \\
\alpha G &= rG + c * K^{grp} \\
\mathcal{H}_n(\mathfrak{m}, [\alpha G]) &= \mathcal{H}_n(\mathfrak{m}, [rG + c * K^{grp}]) \\
c &= c'
\end{aligned}$$

Passo Aggiuntivo di Commit-and-Reveal

Il lettore potrebbe chiedersi da dove provenga il Passo 2. Senza *commit-and-reveal* [110], un co-firmatario malevolo potrebbe apprendere tutti gli $\alpha_e G$ *prima* che la sfida venga generata. Questo gli permette di raggiungere un certo grado di controllo sulla sfida prodotta, modificando il proprio $\alpha_e G$ prima di inviarlo agli altri co-firmatari. In seguito può usare le componenti di risposta raccolte da più firme controllate per derivare le chiavi private k_e^{agg} degli altri co-firmatari in tempo sub-esponenziale [50], e questo comporta una seria minaccia alla sicurezza. Questo *threat model* si basa sulla generalizzazione di Wagner [135] (vedi anche [137] per una spiegazione più intuitiva) del paradosso del compleanno [139].[14]

9.4 Firme Confidenziali ad Anello MLSTAG per Monero

Le transazioni confidenziali ad anello con soglia di Monero aggiungono una certa complessità poiché le chiavi di firma MLSTAG (MLSAG con soglia) consistono in pratica in indirizzi one-time e impegni a zero (per gli importi di input).

[14] Il commit-and-reveal non è utilizzato dall'attuale implementazione multisig di Monero, sebbene sia in fase di studio per futuri rilasci. [101]

Come spiegato nella Sezione 4.2.1, un indirizzo monouso, o one-time, che assegna la proprietà del t-esimo output di una transazione a chiunque abbia l'indirizzo pubblico (K_t^v, K_t^s) si compone come segue:

$$K_t^o = \mathcal{H}_n(rK_t^v, t)G + K_t^s = (\mathcal{H}_n(rK_t^v, t) + k_t^s)G$$
$$k_t^o = \mathcal{H}_n(rK_t^v, t) + k_t^s$$

Possiamo aggiornare la notazione per gli output ricevuti da un indirizzo di gruppo $(K_t^{v,grp}, K_t^{s,grp})$ come segue:

$$K_t^{o,grp} = \mathcal{H}_n(rK_t^{v,grp}, t)G + K_t^{s,grp}$$
$$k_t^{o,grp} = \mathcal{H}_n(rK_t^{v,grp}, t) + k_t^{s,grp}$$

Proprio come prima, chiunque sia in possesso di $k_t^{v,grp}$ e $K_t^{s,grp}$ può verificare che $K_t^{o,grp}$ è l'output posseduto dal proprio indirizzo, e può decifrare il termine Diffie-Hellman per l'importo dell'output e ricostruire la corrispondente maschera di impegno (Sezione 5.3).

Ciò implica anche la possibilità di implementare sottoindirizzi multisig (Sezione 4.3). Le transazioni multisig che utilizzano fondi ricevuti a un sottoindirizzo richiedono alcune modifiche abbastanza semplici agli algoritmi descritti in questo documento e menzionati nelle note a piè di pagina.[15]

9.4.1 `RCTTypeBulletproof2` con Multisig N-su-N

La gran parte delle operazioni per effettuare una transazione multisig può essere completata da chiunque l'abbia avviata. Solo le transazioni multisig a firme MLSTAG richiedono collaborazione tra i partecipanti. In genere, l'iniziatore esegue i seguenti passaggi per preparare una transazione `RCTTypeBulletproof2` (Sezione 6.2):

1. Generare la chiave privata della transazione $r \in_R \mathbb{Z}_l$ (Sezione 4.2) e calcolare la corrispondente chiave pubblica rG (o più chiavi di questo tipo se il destinatario si tratta di sottoindirizzo; Sezione 4.3).

2. Decidere gli input da spendere (tra gli $j \in \{1, ..., m\}$ output posseduti con indirizzi monouso $K_j^{o,grp}$ e importi $a_1, ..., a_m$), e i destinatari a cui inviare i fondi (che genereranno $t \in \{0, ..., p-1\}$ nuovi output con importi $b_0, ..., b_{p-1}$ e indirizzi monouso K_t^o). Questo include la commissione della rete f e il suo impegno fH. In seguito, selezionare gli input per il set di *esche* (decoy) per l'anello.

3. Cifrare l'importo di ogni output $amount_t$ (Sezione 5.3), e calcolare i relativi impegni C_t^b.

4. Selezionare, per ogni input $j \in \{1, ..., m-1\}$, le componenti della maschera dell'impegno pseudo-output $x'_j \in_R \mathbb{Z}_l$, e calcolare la m-esima maschera come segue (Sezione 5.4):
$$x'_m = \sum_t y_t - \sum_{j=1}^{m-1} x'_j$$
ed in seguito calcolare gli impegni pseudo-output $C_j'^a$.

[15] I sottoindirizzi multisig sono supportati in Monero.

5. Produrre la range proof Bulletproof aggregata per tutti gli output. Si rimanda alla Sezione 5.5 per maggiori dettagli.

6. Preparare per le firme MLSTAG generando, per gli impegni a zero, le componenti del seme (seed) $\alpha_j^z \in_R \mathbb{Z}_l$, e calcolando $\alpha_j^z G$.[16]

Infine l'iniziatore invia tutte queste informazioni agli altri partecipanti in modo sicuro. Ora il gruppo di firmatari è pronto a costruire firme di input con le loro chiavi private $k_e^{s,agg}$, e calcolare gli impegni a zero $C^a_{\pi,j} - C'^a_{\pi,j} = z_j G$.

MLSTAG RingCT

Nell'implementazione *core* possiamo notare che per prima cosa vengono costruite le immagini chiave di gruppo per tutti gli input $j \in \{1, ..., m\}$ con indirizzi monouso $K^{o,grp}_{\pi,j}$.[17]

src/wallet/wallet2.cpp `sign_multisig_tx()`

1. Per ogni input j ogni partecipante e effettua i seguenti passaggi:
   (a) calcola l'immagine chiave parziale $\tilde{K}^o_{j,e} = k_e^{s,agg} \mathcal{H}_p(K^{o,grp}_{\pi,j})$,
   (b) e invia $\tilde{K}^o_{j,e}$ agli altri partecipanti in modo sicuro.

2. Ogni partecipante può ora calcolare, usando u_j come indice dell'output nella transazione in cui $K^{o,grp}_{\pi,j}$ è stato inviato all'indirizzo multisig:[18]
$$\tilde{K}^{o,grp}_j = \mathcal{H}_n(k^{v,grp} rG, u_j)\mathcal{H}_p(K^{o,grp}_{\pi,j}) + \sum_e \tilde{K}^o_{j,e}$$

In seguito viene generata la firma MLSTAG per ogni input j.

1. Ogni partecipante e effettua i seguenti passaggi:
   (a) generare le componenti seme (seed) $\alpha_{j,e} \in_R \mathbb{Z}_l$ e calcolare $\alpha_{j,e} G$, e $\alpha_{j,e}\mathcal{H}_p(K^{o,grp}_{\pi,j})$,
   (b) generare, per ogni $i \in \{1, ..., v+1\}$ eccetto $i = \pi$, componenti casuali $r_{i,j,e}$ e $r^z_{i,j,e}$,
   (c) calcolare l'impegno
$$C^\alpha_{j,e} = \mathcal{H}_n(T_{com}, \alpha_{j,e} G, \alpha_{j,e}\mathcal{H}_p(K^{o,grp}_{\pi,j}), r_{1,j,e}, ..., r_{v+1,j,e}, r^z_{1,j,e}, ..., r^z_{v+1,j,e})$$

src/wallet/wallet2.cpp `get_multisig_kLRki()`

[16] Non è necessario eseguire il commit-and-reveal di questi poiché gli impegni a zero sono noti a tutti i co-firmatari.

[17] Se $K^{o,grp}_{\pi,j}$ è costruito da un sottoindirizzo multisig indicizzato i, allora (dalla Sezione 4.3) la sua chiave privata è composta come segue:
$$k^{o,grp}_{\pi,j} = \mathcal{H}_n(k^{v,grp} r_{u_j} K^{s,grp,i}, u_j) + \sum_e k_e^{s,agg} + \mathcal{H}_n(k^{v,grp}, i)$$

[18] Se l'indirizzo monouso corrisponde a un sottoindirizzo multisig di indice i, aggiungere
$$\tilde{K}^{o,grp}_j = ... + \mathcal{H}_n(k^{v,grp}, i)\mathcal{H}_p(K^{o,grp}_{\pi,j})$$

src/cryptonote_basic/cryptonote_format_utils.cpp `generate_key_image_helper_precomp()`

(d) e inviare $C^{\alpha}_{j,e}$ agli altri partecipanti in modo sicuro.

2. Alla ricezione di tutti i $C^{\alpha}_{j,e}$ dagli altri partecipanti, inviare tutti gli $\alpha_{j,e}G$, $\alpha_{j,e}\mathcal{H}_p(K^{o,grp}_{\pi,j})$, e $r_{i,j,e}$ e $r^z_{i,j,e}$, e verificare che l'impegno originale di ogni partecipante fosse valido.

3. Ogni partecipante può calcolare i seguenti valori:

$$\alpha_j G = \sum_e \alpha_{j,e} G$$
$$\alpha_j \mathcal{H}_p(K^{o,grp}_{\pi,j}) = \sum_e \alpha_{j,e}\mathcal{H}_p(K^{o,grp}_{\pi,j})$$
$$r_{i,j} = \sum_e r_{i,j,e}$$
$$r^z_{i,j} = \sum_e r^z_{i,j,e}$$

4. Ogni partecipante può costruire il ciclo di firma (vedi Sezione 3.5).

5. Per finire chiudendo la firma, ogni partecipante e effettua i seguenti passaggi:

   (a) definire $r_{\pi,j,e} = \alpha_{j,e} - c_\pi k^{s,agg}_e \pmod l$,

   (b) e inviare $r_{\pi,j,e}$ agli altri partecipanti in modo sicuro.

6. Tutti possono calcolare (si rircorda che $\alpha^z_{j,e}$ è stato creato dall'iniziatore)[19]

src/ringct/rctSigs.cpp `signMultiSig()`

$$r_{\pi,j} = \sum_e r_{\pi,j,e} - c_\pi * \mathcal{H}_n(k^{v,grp} rG, u_j)$$
$$r^z_{\pi,j} = \alpha^z_{j,e} - c_\pi z_j \pmod l$$

La firma per l'input j è $\sigma_j(\mathfrak{m}) = (c_1, r_{1,j}, r^z_{1,j}, ..., r_{v+1,j}, r^z_{v+1,j})$ con $\tilde{K}^{o,grp}_j$.

Poiché in Monero il messaggio $\mathfrak{m}$ e la sfida c_π dipendono da tutte le altre parti coinvolte nella transazione, qualsiasi partecipante disonesto che cerchi di imbrogliare inviando ai suoi co-firmatari il valore sbagliato, farà fallire il processo di firma. La risposta $r_{\pi,j}$ è utile e significativa solo per il messaggio $\mathfrak{m}$ e la sfida c_π per cui è stata definita.

9.4.2 Comunicazione Semplificata

Sono necessarie non poche operazioni per costruire correttamente una transazione Monero multifirma. Possiamo riorganizzare e semplificare alcuni di questi passaggi in modo tale che le interazioni dei firmatari siano comprese in due parti essenziali per un totale di cinque passaggi:

[19] Se l'indirizzo monouso $K^{o,grp}_{\pi,j}$ corrisponde a un sottoindirizzo multisig di indice i, aggiungere anche:

$$r_{\pi,j} = ... - c_\pi * \mathcal{H}_n(k^{v,grp}, i)$$

src/cryptonote_basic/cryptonote_format_utils.cpp `generate_key_image_helper_precomp()`

1. *Aggregazione delle chiavi per un indirizzo pubblico multisig*: Chiunque disponga di un insieme di indirizzi pubblici può eseguire `premerge` su di essi e poi effettuare il `merge` su un indirizzo N-su-N, ma nessun partecipante potrà conoscere la chiave di visualizzazione di gruppo a meno che non apprenda tutte le relative componenti. Dunque ogni partecipante del gruppo inizia scambiandosi k_e^v e $K_e^{s,base}$ in modo sicuro. Qualsiasi partecipante può eseguire `premerge` e `merge` e pubblicare $(K^{v,grp}, K^{s,grp})$, consentendo a tutti i partecipanti di ricevere fondi all'indirizzo di gruppo. L'aggregazione M-su-N richiede ulteriori passaggi, descritti nella Sezione 9.6.

   src/wallet/wallet2.cpp `pack_multisignature_keys()`

2. *Transazioni*:

   (a) l'iniziatore (un partecipante o una sotto-coalizione del gruppo) decide di avviare una collaborazione su una transazione. Sceglie m input con indirizzi monouso $K_j^{o,grp}$ e impegni di importo C_j^a, m insiemi di v indirizzi monouso aggiuntivi e impegni da usare come esche (decoy) nell'anello, sceglie p destinatari di output con indirizzi pubblici (K_t^v, K_t^s) e importi b_t da inviare loro, decide una commissione di transazione f, genera la chiave privata della transazione r,[20] genera le maschere di impegno pseudo-output x'_j con $j \neq m$, costruisce il termine ECDH $amount_t$ per ogni output, produce una prova di intervallo aggregata e genera i valori α_j^z per gli impegni a zero di tutti gli input e scalari casuali $r_{i,j}$ e $r_{i,j}^z$ con $i \neq \pi_j$.[21] Tutto ciò verrà utilizzato nel prossimo step di comunicazione.

   src/wallet/wallet2.cpp `transfer_selected_rct()`

   L'iniziatore invia tutte queste informazioni agli altri partecipanti in modo sicuro.[22] Gli altri partecipanti possono confermare proseguendo nel prossimo step di comunicazione, oppure negoziare delle modifiche.

   src/wallet/wallet2.cpp `save_multisig_tx()`

   (b) Ogni partecipante sceglie le proprie componenti di apertura per la/e firma/e MLSTAG, si impegna con esse, calcola le proprie immagini chiave parziali e invia tali impegni e immagini parziali agli altri partecipanti in modo sicuro.

   Firma/e MLSTAG: l'immagine chiave è denotata con $\tilde{K}_{j,e}^o$, e l'aleatorietà della firma con $\alpha_{j,e}G$, e $\alpha_{j,e}\mathcal{H}_p(K_{\pi,j}^{o,grp})$. Le immagini chiave parziali non necessitano di essere incluse nei dati impegnati, poiché non possono essere usate per derivare le chiavi private dei firmatari. Sono però utili per visualizzare quali output posseduti sono stati spesi, quindi per il bene di un design modulare dovrebbero essere gestite separatamente.

   (c) Alla ricezione di tutti gli impegni di firma, ogni partecipante invia le informazioni impegnate agli altri partecipanti in modo sicuro.

[20] O più chiavi private r_t se si invia ad almeno un sottoindirizzo.

[21] Si noti che semplifichiamo il processo di firma lasciando che l'iniziatore generi scalari casuali $r_{i,j}$ e $r_{i,j}^z$, invece di sommare le componenti generate da ogni co-firmatario.

[22] Non ha bisogno di inviare direttamente gli importi di output b_t, poiché possono essere calcolati da $amount_t$. Monero adotta l'approccio ragionevole di creare una transazione parziale riempita con le informazioni selezionate dall'iniziatore, e inviandola ad altri co-firmatari insieme a un elenco di informazioni correlate come le chiavi private della transazione, indirizzi di destinazione, gli input reali, ecc.

(d) Ogni partecipante chiude la propria parte della/e firma/e MLSTAG, inviando tutti gli $r_{\pi_j,j,e}$ agli altri partecipanti in modo sicuro.[23]

Supponendo che l'intero processo sia andato a buon fine, tutti i partecipanti possono finire di scrivere la transazione e trasmetterla autonomamente. Le transazioni autorizzate da una coalizione multisig sono indistinguibili da quelle autorizzate da singoli individui.

9.5 Calcolo delle Immagini Chiave

Se un utente perde i propri registri e vuole calcolare il saldo del proprio indirizzo (fondi ricevuti meno fondi spesi), deve farlo con l'aiuto della blockchain. Le chiavi di visualizzazione sono utili solo per visualizzare i fondi ricevuti, quindi gli utenti devono calcolare le immagini chiave per tutti gli output posseduti per vedere se sono stati spesi, confrontandoli con le immagini chiave memorizzate nella blockchain. Poiché i membri di un indirizzo di gruppo non possono calcolare le immagini chiave da soli, hanno bisogno dell'aiuto degli altri partecipanti.

Il calcolo delle immagini chiave da una semplice somma di componenti potrebbe fallire se partecipanti disonesti riportano chiavi false.[24] Dato un output ricevuto con indirizzo monouso $K^{o,grp}$, il gruppo può produrre una semplice prova in stile Schnorr 'collegabile' (essenzialmente bLSTAG a chiave singola, richiamare le Sezioni 3.1 e 3.4) per dimostrare che l'immagine chiave $\tilde{K}^{o,grp}$ è legittima senza rivelare i loro componenti privati di chiave di spesa o aver bisogno di fidarsi l'uno dell'altro.

Prova

1. Ogni partecipante e effettua i seguenti passaggi:
   (a) calcola $\tilde{K}^o_e = k^{s,agg}_e \mathcal{H}_p(K^{o,grp})$,
   (b) genera la componente seme (seed) $\alpha_e \in_R \mathbb{Z}_l$ e calcola $\alpha_e G$ e $\alpha_e \mathcal{H}_p(K^{o,grp})$,
   (c) si impegna con i dati con $C^\alpha_e = \mathcal{H}_n(T_{com}, \alpha_e G, \alpha_e \mathcal{H}_p(K^{o,grp}))$,
   (d) e invia C^α_e e $\tilde{K}^o_e$ agli altri partecipanti in modo sicuro.
2. Alla ricezione di tutti i C^α_e, ogni partecipante invia le informazioni impegnate e verifica che gli impegni degli altri partecipanti fossero legittimi.

[23] È imperativo che ogni tentativo di firma da parte di un firmatario utilizzi un $\alpha_{j,e}$ univoco, per evitare di divulgare la sua chiave di spesa privata ad altri firmatari (come spiegato nella Sezione 2.3.4) [110]. I portafogli dovrebbero fondamentalmente imporre questo eliminando sempre $\alpha_{j,e}$ ogni volta che una risposta che lo utilizza è stata trasmessa al di fuori del portafoglio.

[24] Attualmente Monero sembra utilizzare una semplice somma.

src/wallet/ wallet2.cpp save_multisig()

src/wallet/ wallet2.cpp export_multisig()

3. Ogni partecipante può calcolare:[25]

$$\tilde{K}^{o,grp} = \mathcal{H}_n(k^{v,grp} rG, u)\mathcal{H}_p(K^{o,grp}) + \sum_e \tilde{K}^o_e$$

$$\alpha G = \sum_e \alpha_e G$$

$$\alpha \mathcal{H}_p(K^{o,grp}) = \sum_e \alpha_e \mathcal{H}_p(K^{o,grp})$$

4. Ogni partecipante calcola la sfida:[26]

$$c = \mathcal{H}_n([\alpha G], [\alpha \mathcal{H}_p(K^{o,grp})])$$

5. Ogni partecipante effettua i seguenti passaggi:
   (a) definisce $r_e = \alpha_e - c * k_e^{s,agg} \pmod l$,
   (b) e invia r_e agli altri partecipanti in modo sicuro.

6. Infine, ogni partecipante può calcolare[27]:

$$r^{resp} = \sum_e r_e - c * \mathcal{H}_n(k^{v,grp} rG, u)$$

La prova è (c, r^{resp}) con $\tilde{K}^{o,grp}$.

Verifica

1. Verificare che $l\tilde{K}^{o,grp} \stackrel{?}{=} 0$.
2. Calcolare $c' = \mathcal{H}_n([r^{resp}G + c * K^{o,grp}], [r^{resp}\mathcal{H}_p(K^{o,grp}) + c * \tilde{K}^{o,grp}])$.
3. Se $c = c'$ allora l'immagine chiave $\tilde{K}^{o,grp}$ corrisponde all'indirizzo monouso $K^{o,grp}$ (tranne che con probabilità trascurabile).

9.6 Soglie più Piccole

All'inizio di questo capitolo sono stati discussi i servizi di escrow, che utilizzavano la multisig 2-su-2 per distribuire la capacità di firma tra un utente e una società di sicurezza. Questa configurazione

[25] Se l'indirizzo monouso corrisponde a un sottoindirizzo multisig indicizzato i, aggiungi

$$\tilde{K}^{o,grp} = ... + \mathcal{H}_n(k^{v,grp}, i)\mathcal{H}_p(K^{o,grp})$$

[26] Questa prova dovrebbe includere la separazione di dominio e la prefissazione della chiave, che omettiamo per brevità.

[27] Se l'indirizzo monouso $K^{o,grp}$ corrisponde a un sottoindirizzo multisig indicizzato i, includi

$$r^{resp} = ... - c * \mathcal{H}_n(k^{v,grp}, i)$$

non è ideale, perché se la società di sicurezza viene compromessa, o si rifiuta di cooperare, i fondi dell'utente potrebbero rimanere bloccati per sempre.

Possiamo aggirare questa questo problema con un indirizzo multisig 2-su-3, che ha tre partecipanti ma ne richiede solo due per firmare le transazioni. Un servizio di escrow fornisce una chiave e gli utenti forniscono due chiavi. Gli utenti possono archiviare una chiave di backup in un luogo sicuro (come una cassetta di sicurezza) e utilizzare l'altra per gli acquisti quotidiani. Se il servizio di escrow fallisce, un utente può sempre utilizzare la chiave di backup assieme alla chiave degli acquisti giornalieri per prelevare i fondi.

Le multifirme con soglie inferiori a N dispongono di un ampio ventaglio di casi d'uso.

9.6.1 Aggregazione delle Chiavi 1-su-N

Si supponga che un gruppo di persone voglia creare una chiave multisig K^{grp} con cui tutti possano firmare. La soluzione è banale: far conoscere a tutti la chiave privata k^{grp} e ci sono tre modi per farlo:

1. Un partecipante o una sotto-coalizione seleziona una chiave e la invia a tutti gli altri in modo sicuro.
2. Tutti i partecipanti calcolano le componenti della chiave privata e se le scambiano a vicenda in modo sicuro, usando la semplice somma per la costruzione della chiave di gruppo.[28]
3. I partecipanti estendono la multisig M-su-N a 1-su-N. Questo potrebbe risultare utile se un avversario ha accesso alle comunicazioni del gruppo.

In questo caso, per Monero, tutti i partecipanti sarebbero a conoscenza delle chiavi private $(k^{v,grp,\text{1xN}}, k^{s,grp,\text{1xN}})$. Si tiene a precisare che nelle sezioni precedenti tutte le chiavi di gruppo erano indicate come N-su-N, mentre in questa sezione la dicitura 1xN è usata per denotare le chiavi relative alla tipologia di firma 1-su-N.

9.6.2 Aggregazione delle Chiavi (N-1)-su-N

In una chiave di gruppo (N-1)-su-N, come 2-su-3 o 4-su-5, qualsiasi insieme di (N-1) partecipanti può produrre una firma valida. Ciò è possibile grazie ai segreti condivisi Diffie-Hellman. Si supponga che ci sono partecipanti $e \in \{1, ..., N\}$, con chiavi pubbliche base K_e^{base} di cui sono tutti a conoscenza.

Ogni partecipante e calcola, per $w \in \{1, ..., N\}$ ma $w \neq e$

$$k_{e,w}^{sh,\text{(N-1)xN}} = \mathcal{H}_n(k_e^{base} K_w^{base})$$

src/multisig/multisig.cpp `generate_multisig_N1_N()`

[28] Si noti che il problema dell'annullamento della chiave non si presenta qui, in quanto tutti conoscono la chiave privata completa.

Poi calcola tutti i $K_{e,w}^{sh,\text{(N-1)xN}} = k_{e,w}^{sh,\text{(N-1)xN}} G$ e li invia agli altri partecipanti in modo sicuro. D'ora in poi la dicitura *sh* verrà utilizzata per denotare le chiavi condivise da un sotto-gruppo di partecipanti.

Ogni partecipante avrà dunque (N-1) componenti di chiave privata condivisa corrispondenti a ciascuno degli altri partecipanti, per un totale di N*(N-1) chiavi tra tutti i membri del gruppo. Tutte le chiavi sono condivise da due partecipanti ad uno scambio Diffie-Hellman, quindi ci sono in realtà [N*(N-1)]/2 chiavi univoche, che vanno a costituire l'insieme $\mathbb{S}^{\text{(N-1)xN}}$.

Generalizzazione di `premerge` e `merge`

Trattiamo adesso la generalizzazione della definizione di `premerge` (Sezione 9.2.3): Il suo input sarà l'insieme $\mathbb{S}^{\text{MxN}}$, dove M è la *soglia* per cui le chiavi dell'insieme sono state *preparate*. Nel caso particolare $M = N$ avremo che $\mathbb{S}^{\text{NxN}} = \mathbb{S}^{base}$, mentre se $M < N$ allora $\mathbb{S}^{\text{MxN}}$ contiene chiavi condivise. Il risultato di questa funzione è indicato come $\mathbb{K}^{agg,\text{MxN}}$.

I [N*(N-1)]/2 componenti chiave in $\mathbb{K}^{agg,\text{(N-1)xN}}$ possono essere dati in pasto alla funzione `merge`, producendo la chiave di gruppo $K^{grp,\text{(N-1)xN}}$. È importante notare che tutti le [N*(N-1)]/2 componenti di chiave privata possono essere assemblate con soli (N-1) partecipanti poiché ogni membro di questo sottogruppo condivide un segreto Diffie-Hellman con l'N$^{\text{esimo}}$ partecipante.

Esempio 2-su-3

Si supponga che tre persone dotate di chiavi pubbliche $\{K_1^{base}, K_2^{base}, K_3^{base}\}$, di cui ciascuna conosce la rispettiva chiave privata, vogliano creare una chiave multisig 2-su-3. Dopo uno scambio Diffie-Hellman e l'invio reciproco delle chiavi pubbliche, ognuno di loro conosce quanto segue:

1. Persona 1: $k_{1,2}^{sh,2\text{x}3}$, $k_{1,3}^{sh,2\text{x}3}$, $K_{2,3}^{sh,2\text{x}3}$
2. Persona 2: $k_{2,1}^{sh,2\text{x}3}$, $k_{2,3}^{sh,2\text{x}3}$, $K_{1,3}^{sh,2\text{x}3}$
3. Persona 3: $k_{3,1}^{sh,2\text{x}3}$, $k_{3,2}^{sh,2\text{x}3}$, $K_{1,2}^{sh,2\text{x}3}$

Dove $k_{1,2}^{sh,2\text{x}3} = k_{2,1}^{sh,2\text{x}3}$, e così via. Inoltre, l'insieme $\mathbb{S}^{2\text{x}3}$ è costituito dagli elementi $\{K_{1,2}^{sh,2\text{x}3}, K_{1,3}^{sh,2\text{x}3}, K_{2,3}^{sh,2\text{x}3}\}$.

L'esecuzione di `premerge` e `merge` crea la seguente chiave di gruppo[29]:

$$\begin{aligned} K^{grp,2\text{x}3} =& \mathcal{H}_n(T_{agg}, \mathbb{S}^{2\text{x}3}, K_{1,2}^{sh,2\text{x}3}) K_{1,2}^{sh,2\text{x}3} + \\ & \mathcal{H}_n(T_{agg}, \mathbb{S}^{2\text{x}3}, K_{1,3}^{sh,2\text{x}3}) K_{1,3}^{sh,2\text{x}3} + \\ & \mathcal{H}_n(T_{agg}, \mathbb{S}^{2\text{x}3}, K_{2,3}^{sh,2\text{x}3}) K_{2,3}^{sh,2\text{x}3} \end{aligned}$$

Si supponga adesso che le persone 1 e 2 vogliono firmare un messaggio $\mathfrak{m}$. In questo esempio verrà utilizzata una firma in stile Schnorr di base a fine dimostrativo:

[29] Poiché la chiave unita è composta da segreti condivisi, un osservatore che conosce solo le chiavi pubbliche base originali non sarebbe in grado di aggregarle (Sezione 9.2.2) e identificare i membri della chiave unita.

1. Ogni partecipante $e \in \{1, 2\}$ effettua i seguenti passaggi:
   (a) sceglie un componente casuale $\alpha_e \in_R \mathbb{Z}_l$,
   (b) calcola $\alpha_e G$,
   (c) si impegna sul valore $C^{\alpha}_e = \mathcal{H}_n(T_{com}, \alpha_e G)$,
   (d) ed invia C^{α}_e agli altri partecipanti in modo sicuro.
2. Alla ricezione di tutti i C^{α}_e, ogni partecipante invia $\alpha_e G$ e verifica la legittimità degli altri impegni.
3. Ogni partecipante calcola
$$\alpha G = \sum_e \alpha_e G$$
$$c = \mathcal{H}_n(\mathfrak{m}, [\alpha G])$$
4. Il partecipante 1 fa quanto segue:
   (a) calcola $r_1 = \alpha_1 - c * [k^{agg,2x3}_{1,3} + k^{agg,2x3}_{1,2}]$,
   (b) e invia r_1 al partecipante 2 in modo sicuro.
5. Il partecipante 2 fa quanto segue:
   (a) calcola $r_2 = \alpha_2 - c * k^{agg,2x3}_{2,3}$,
   (b) e invia r_2 al partecipante 1 in modo sicuro.
6. Ogni partecipante calcola
$$r = \sum_e r_e$$
7. Infine, qualsiasi partecipante può pubblicare la firma $\sigma(\mathfrak{m}) = (c, r)$.

L'unica differenza con le firme con soglia minore di N è come 'viene chiuso il cerchio' attraverso la definizione di $r_{\pi,e}$ (nel caso delle firme ad anello, con indice segreto π). Ogni partecipante deve includere il proprio segreto condiviso corrispondente al 'partecipante mancante', ma poiché tutti gli altri segreti condivisi sono raddoppiati c'è un trucco. Dato l'insieme $\mathbb{S}^{base}$ di tutte le chiavi originali dei partecipanti, solo la *prima persona* - ordinata per indice in $\mathbb{S}^{base}$ - con la copia di un segreto condiviso lo usa per calcolare il suo $r_{\pi,e}$.[30,31]

[30] In pratica questo significa che un iniziatore dovrebbe determinare quale sottoinsieme di M firmatari firmerà un dato messaggio. Se scopre che O firmatari sono disposti a firmare, con (O ≥ M), può orchestrare più tentativi di firma concorrenti per ogni sottoinsieme di dimensione M all'interno di O per aumentare le possibilità di successo. Sembra che Monero utilizzi questo approccio. Si scopre anche (un punto esoterico) che l'elenco *originale* delle destinazioni di output creato dall'iniziatore viene mescolato casualmente, e quell'elenco casuale viene quindi utilizzato da tutti i tentativi di firma concorrenti e da tutti gli altri co-firmatari (questo è correlato al flag oscuro `shuffle_outs`). src/wallet/wallet2.cpp `transfer_selected_rct()`

[31] Attualmente Monero sembra utilizzare un metodo di firma round-robin, dove l'iniziatore firma con tutte le sue chiavi private, passa la transazione parzialmente firmata a un altro firmatario che firma con tutte le *sue* chiavi private (che non sono ancora state usate per firmare), che passa a un altro firmatario, e così via, fino all'ultimo firmatario che può pubblicare la transazione o inviarla ad altri firmatari in modo che possano pubblicarla. src/wallet/wallet2.cpp `sign_multisig_tx()`

Nell'esempio precedente, il partecipante 1 calcola:

$$r_1 = \alpha_1 - c * [k_{1,3}^{agg,2\text{x}3} + k_{1,2}^{agg,2\text{x}3}]$$

mentre il partecipante 2 calcola solo:

$$r_2 = \alpha_2 - c * k_{2,3}^{agg,2\text{x}3}$$

Lo stesso principio si applica al calcolo dell'immagine chiave di gruppo nelle transazioni multisig Monero con soglia minore di N.

9.6.3 Aggregazione delle Chiavi M-su-N

Trattiamo il caso (N-1)-su-N in modo tale da poter generalizzare in maniera più chiara il caso M-su-N. Nel caso di multisig (N-1)-su-N ogni segreto condiviso tra due chiavi pubbliche, come K_1^{base} e K_2^{base}, contiene due chiavi private, $k_1^{base} k_2^{base} G$. È un segreto perché si suppone che solo la persona 1 può calcolare $k_1^{base} K_2^{base}$ e solo la persona 2 può calcolare $k_2^{base} K_1^{base}$.

Si supponga la presenza di una terza persona con K_3^{base}. I segreti condivisi in questo caso sono $k_1^{base} k_2^{base} G$, $k_1^{base} k_3^{base} G$, e $k_2^{base} k_3^{base} G$, e i partecipanti si scambiano le rispettive chiavi pubbliche (rendendole non più segrete). Ognuno di loro, dunque, contribuisce con una chiave privata a due chiavi pubbliche. Si supponga adesso la creazione di un nuovo segreto condiviso attraverso la terza chiave pubblica.

La persona 1 calcola il segreto condiviso $k_1^{base} * (k_2^{base} k_3^{base} G)$, la persona 2 calcola $k_2^{base} * (k_1^{base} k_3^{base} G)$, e la persona 3 calcola $k_3^{base} * (k_1^{base} k_2^{base} G)$. Ora tutti conoscono $k_1^{base} k_2^{base} k_3^{base} G$, creando un segreto (purché nessuno lo pubblichi) condiviso a tre vie.

src/multisig/multisig.cpp `generate_multisig_derivations()`

Il gruppo può usare $k^{sh,1\text{x}3} = \mathcal{H}_n(k_1^{base} k_2^{base} k_3^{base} G)$ come chiave privata condivisa, e pubblicare

$$K^{grp,1\text{x}3} = \mathcal{H}_n(T_{agg}, \mathbb{S}^{1\text{x}3}, K^{sh,1\text{x}3}) K^{sh,1\text{x}3}$$

come indirizzo multisig 1-su-3.

In una multisig 3-su-3 ogni singola persona possiede un segreto, in una multisig 2-su-3 ogni gruppo di 2 persone possiede un segreto condiviso, e nel caso 1-su-3 ogni gruppo di 3 persone possiede un segreto condiviso.

È possibile adesso proseguire con la generalizzazione al caso M-su-N: ogni possibile gruppo di (N-M+1) persone possiede un segreto condiviso [108]. Se (N-M) persone mancano, tutti i loro segreti condivisi sono posseduti da almeno una delle M persone rimanenti, che possono collaborare per produrre la firma con la chiave di gruppo.

Algoritmo M-su-N

Dati i partecipanti $e \in \{1, ..., N\}$ con chiavi private iniziali $k_1^{base}, ..., k_N^{base}$ che desiderano creare una chiave congiunta M-su-N (M $\leq$ N; M $\geq$ 1 e N $\geq$ 2), è possibile seguire un algoritmo interattivo al fine di produrla.

src/wallet/wallet2.cpp `exchange_multisig_keys()`

$\mathbb{S}_s$ sarà usato per denotare tutte le chiavi pubbliche *univoche* allo stadio $s \in \{0, ..., (N-M)\}$. L'insieme finale $\mathbb{S}_{N-M}$ è ordinato secondo un criterio di ordinamento (ad esempio, dal più piccolo al più grande numericamente, cioè lessicograficamente). Questa notazione offre una certa comodità, e $\mathbb{S}_s$ equivale a $\mathbb{S}^{\text{(N-s)xN}}$ delle sezioni precedenti.

Sarà usato $\mathbb{S}^K_{s,e}$ per denotare l'insieme di chiavi pubbliche che ogni partecipante ha creato allo stadio s dell'algoritmo. All'inizio $\mathbb{S}^K_{0,e} = \{K^{base}_e\}$.

Alla fine dell'esecuzione dell'algoritmo, l'insieme $\mathbb{S}^k_e$ conterrà le chiavi private di aggregazione di ogni partecipante.

1. Ogni partecipante e invia il proprio insieme di chiavi pubbliche originali $\mathbb{S}^K_{0,e} = \{K^{base}_e\}$ agli altri partecipanti in modo sicuro.

2. Ogni partecipante costruisce $\mathbb{S}_0$ raccogliendo tutti gli $\mathbb{S}^K_{0,e}$ e rimuovendo i duplicati.

3. Per lo stadio $s \in \{1, ..., (N-M)\}$ (salta se M = N)

   (a) Ogni partecipante e effettua i seguenti passaggi:

      i. Per ogni elemento g_{s-1} di $\mathbb{S}_{s-1} \notin \mathbb{S}^K_{s-1,e}$, calcola un nuovo segreto condiviso
      $$k^{base}_e * \mathbb{S}_{s-1}[g_{s-1}]$$
      ii. Inserisce tutti i nuovi segreti condivisi in $\mathbb{S}^K_{s,e}$.
      iii. Se $s = (N-M)$, calcola la chiave privata condivisa per ogni elemento x in $\mathbb{S}^K_{N-M,e}$
      $$\mathbb{S}^k_e[x] = \mathcal{H}_n(\mathbb{S}^K_{N-M,e}[x])$$
      e sovrascrive la chiave pubblica impostando $\mathbb{S}^K_{N-M,e}[x] = \mathbb{S}^k_e[x] * G$.
      iv. Invia agli altri partecipanti $\mathbb{S}^K_{s,e}$.

   (b) Ogni partecipante costruisce $\mathbb{S}_s$ raccogliendo tutti gli $\mathbb{S}^K_{s,e}$ e rimuovendo i duplicati.[32]

4. Ogni partecipante ordina $\mathbb{S}_{N-M}$ secondo il criterio stabilito.

5. La funzione `premerge` prende $\mathbb{S}_{(N-M)}$ come input, e per ogni $g \in \{1, ..., (\text{dimensione di } \mathbb{S}_{N-M})\}$ la corrispondente chiave di aggregazione è:
$$\mathbb{K}^{agg,\text{MxN}}[g] = \mathcal{H}_n(T_{agg}, \mathbb{S}_{(N-M)}, \mathbb{S}_{(N-M)}[g]) * \mathbb{S}_{(N-M)}[g]$$

6. La funzione `merge` prende $\mathbb{K}^{agg,\text{MxN}}$ come input, e la chiave di gruppo restituita è:
$$K^{grp,\text{MxN}} = \sum_{g=1}^{\text{dimensione di } \mathbb{S}_{N-M}} \mathbb{K}^{agg,\text{MxN}}[g]$$

7. Ogni partecipante e sovrascrive ogni elemento x in $\mathbb{S}^k_e$ con la propria chiave privata di aggregazione:
$$\mathbb{S}^k_e[x] = \mathcal{H}_n(T_{agg}, \mathbb{S}_{(N-M)}, \mathbb{S}^k_e[x]G) * \mathbb{S}^k_e[x]$$

[32] I partecipanti dovrebbero tenere traccia di chi ha quali chiavi all'ultimo stadio ($s = N - M$), per facilitare la firma collaborativa, dove solo la prima persona in $\mathbb{S}_0$ con una certa chiave privata la usa per firmare. Vedi Sezione 9.6.2.

Nota: Se si desidera conferire a determinati utenti un potere di firma maggiore in una multisig, come 2 quote in una 3-su-4, essi dovrebbero usare più componenti chiave di partenza invece di riutilizzare le stesse.

9.7 Famiglie di Chiavi

Fino ad ora è stata considerata l'aggregazione delle chiavi solo tra un semplice gruppo di firmatari. Ad esempio, Alice, Bob e Carol che contribuiscono ciascuno con componenti chiave a un indirizzo multisig 2-su-3.

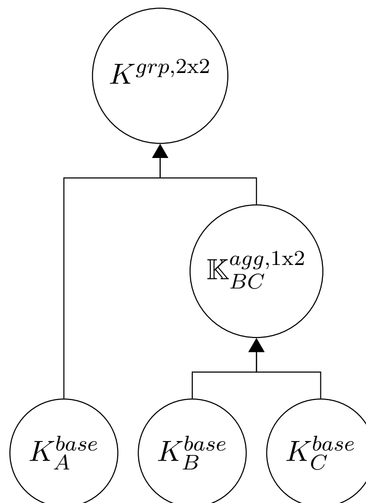
Si supponga che Alice voglia firmare tutte le transazioni da quell'indirizzo, ma non voglia che Bob o Carol firmino senza di lei. In altre parole, (Alice + Bob) o (Alice + Carol) sono accettabili, ma non (Bob + Carol).

È possibile ottenere ciò attraverso una suddivisione in due livelli di aggregazione delle chiavi. Prima un'aggregazione multisig 1-su-2 $\mathbb{K}^{agg,1\text{x}2}_{BC}$ tra Bob e Carol, poi una chiave di gruppo 2-su-2 $K^{grp,2\text{x}2}$ tra Alice e $\mathbb{K}^{agg,1\text{x}2}_{BC}$. In pratica, un indirizzo multisig (2-su-([1-su-1] e [1-su-2])).

Ciò implica che le strutture di accesso ai diritti di firma possano essere piuttosto flessibili.

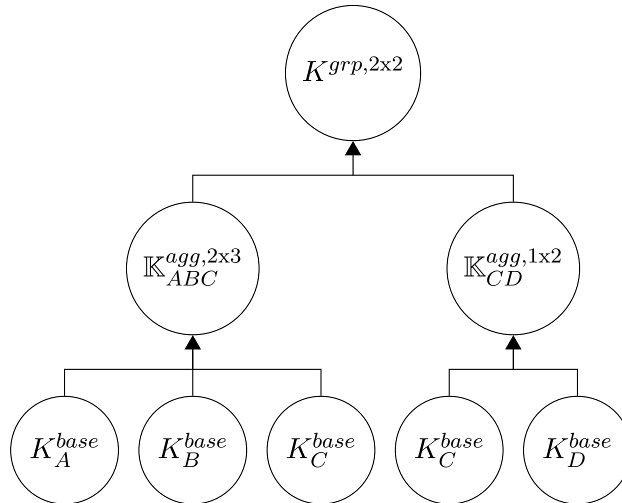
9.7.1 Alberi Genealogici

È possibile illustrare attraverso un diagramma l'indirizzo multisig (2-su-([1-su-1] e [1-su-2])) come segue:



Le chiavi $K^{base}_A, K^{base}_B, K^{base}_C$ sono considerate *antenati radice*, mentre $\mathbb{K}^{agg,1\text{x}2}_{BC}$ è il *figlio* dei *genitori* K^{base}_B e K^{base}_C. I genitori possono avere più di un figlio, anche se per chiarezza concettuale consideriamo ogni copia di un genitore come distinta. Ciò significa che possono esserci più antenati radice che coincidono con la stessa chiave.

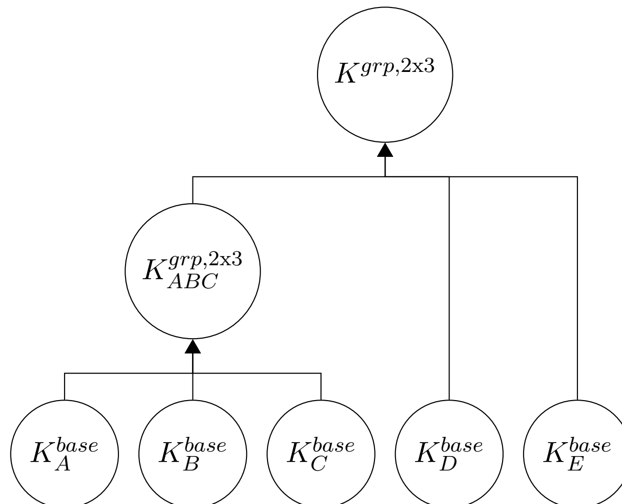
Ad esempio, in questo 2-su-3 e 1-su-2 uniti in un 2-su-2, la chiave di Carol K_C^{base} è usata due volte e dunque inserita nell'albero due volte:



Insiemi separati $\mathbb{S}$ sono definiti per ogni sotto-coalizione multisig. Ci sono tre insiemi *premerge* nell'esempio di sopra: $\mathbb{S}_{ABC}^{2x3} = \{K_{AB}^{sh,2x3}, K_{BC}^{sh,2x3}, K_{AC}^{sh,2x3}\}$, $\mathbb{S}_{CD}^{1x2} = \{K_{CD}^{sh,1x2}\}$, e $\mathbb{S}_{final}^{2x3} = \{\mathbb{K}_{ABC}^{agg,2x3}, \mathbb{K}_{CD}^{agg,1x2}\}$.

9.7.2 Annidamento delle Chiavi Multisig

Supponiamo di disporre della seguente famiglia di chiavi:



Se uniamo le chiavi in $\mathbb{S}_{ABC}^{2x3}$ corrispondenti al primo 2-su-3, incontriamo un problema al livello successivo. Prendiamo un solo segreto condiviso, tra $K_{ABC}^{grp,2x3}$ e K_D^{base}, per illustrare:

$$k_{ABC,D} = \mathcal{H}_n(k_{ABC}^{grp,2x3} K_D^{base})$$

Ora, due persone del gruppo ABC potrebbero facilmente contribuire con componenti della chiave di aggregazione in modo che la sotto-coalizione possa calcolare:

$$k^{grp,2\text{x}3}_{ABC} K^{base}_{D} = \sum k^{agg,2\text{x}3}_{ABC} K^{base}_{D}$$

Il problema è che tutti i partecipanti del gruppo ABC possono calcolare $k^{sh,2\text{x}3}_{ABC,D} = \mathcal{H}_n(k^{grp,2\text{x}3}_{ABC} K^{base}_{D})$! Se i partecipanti di una multisig di livello inferiore conoscono tutte le chiavi private per una multisig di livello superiore, allora la multisig di livello inferiore potrebbe anche essere 1-su-N.

Aggiriamo questo problema non unendo completamente le chiavi fino alla chiave figlia finale. Invece, eseguiamo semplicemente `premerge` per tutte le chiavi prodotte dalle multisig di livello più basso.

Soluzione per l'Annidamento

Per usare $\mathbb{K}^{agg,\text{MxN}}$ in una nuova multisig, la passiamo in giro proprio come una chiave normale, ma con una differenza. Le operazioni che coinvolgono $\mathbb{K}^{agg,\text{MxN}}$ usano ciascuna delle sue chiavi membro, invece della chiave di gruppo congiunta. Ad esempio, la 'chiave' pubblica di un segreto condiviso tra $\mathbb{K}^{agg,2\text{x}3}_{x}$ e K^{base}_{A} potrebbe apparire come segue:

$$\mathbb{K}^{sh,1\text{x}2}_{x,A} = \{[\mathcal{H}_n(k^{base}_{A} \mathbb{K}^{agg,2\text{x}3}_{x}[1]) * G], [\mathcal{H}_n(k^{base}_{A} \mathbb{K}^{agg,2\text{x}3}_{x}[2]) * G], ...\}$$

In questo modo tutti i membri di $\mathbb{K}^{agg,2\text{x}3}_{x}$ conoscono solo i segreti condivisi corrispondenti alle loro chiavi private dalla multisig 2-su-3 di livello inferiore. Un'operazione tra un insieme di chiavi di dimensione due ${}^{2}\mathbb{K}_{A}$ e un insieme di chiavi di dimensione tre ${}^{3}\mathbb{K}_{B}$ produrrebbe un insieme di chiavi di dimensione sei ${}^{6}\mathbb{K}_{AB}$. Possiamo generalizzare tutte le chiavi in una famiglia di chiavi come insiemi di chiavi, dove le singole chiavi sono denotate ${}^{1}\mathbb{K}$. Gli elementi di un insieme di chiavi sono ordinati secondo un criterio prestabilito (cioè dal più piccolo al più grande numericamente), e gli insiemi contenenti a loro volta insiemi di chiavi (ad es. insiemi $\mathbb{S}$) sono ordinati in base al primo elemento in ogni insieme di chiavi, sempre secondo un criterio prestabilito.

Facciamo in modo che gli insiemi di chiavi si propaghino attraverso la struttura familiare, con ogni gruppo multisig annidato che invia il proprio insieme di aggregazione `premerge` come 'chiave base' per il livello successivo,[33] fino a quando non compare l'insieme di aggregazione dell'ultimo figlio, a quel punto viene finalmente usato `merge`.

Gli utenti dovrebbero memorizzare le loro chiavi private base, le chiavi private di aggregazione per tutti i livelli di una struttura familiare multisig, e le chiavi pubbliche per tutti i livelli. Ciò facilita la creazione di nuove strutture, l'`unione` di multisig annidate, e la collaborazione con altri firmatari per ricostruire una struttura in caso di problemi.

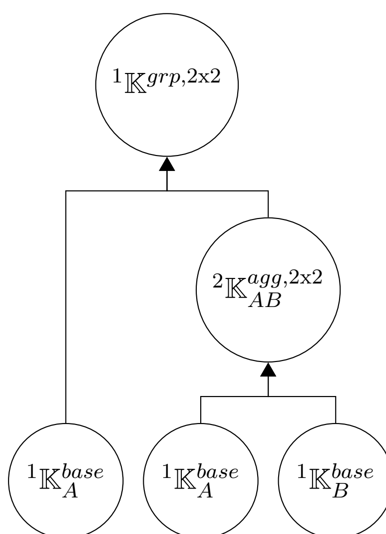
[33] Si noti che `premerge` deve essere eseguito sugli output di *tutte* le multisig annidate, anche quando una multisig N'-su-N' è annidata in una N-su-N, perché l'insieme $\mathbb{S}$ cambierà.

9.7.3 Implicazioni per Monero

Ogni sotto-coalizione che contribuisce alla chiave finale deve contribuire con delle componenti alle transazioni Monero (come i valori di apertura αG), e quindi ogni sotto-sotto-coalizione deve contribuire alla sua sotto-coalizione figlia.

Ciò significa che ogni antenato radice, anche quando ci sono più copie della stessa chiave nella struttura familiare, deve contribuire con un componente radice al proprio figlio, e ogni figlio con un componente al proprio figlio e così via. Usiamo semplici somme ad ogni livello.

Prendiamo come esempio la seguente famiglia di chiavi:



In questo caso i partecipanti devono calcolare un valore di gruppo x per una produrre una firma. Gli antenati radice contribuiscono: $x_{A,1}$, $x_{A,2}$, x_B. Il totale è $x = x_{A,1} + x_{A,2} + x_B$.

Attualmente non esistono implementazioni di chiavi multisig annidate in Monero.

CAPITOLO 10

Marketplace Monero con Deposito a Garanzia (Escrow)

La maggior parte delle volte, acquistare da un negozio online è un'operazione semplice che procede senza intoppi. In genere, l'acquirente invia del denaro al venditore e il prodotto atteso arriva alla porta di casa. Se il prodotto presenta un difetto, o in alcuni casi se l'acquirente cambia idea, può restituirlo e ottenere un rimborso.

È difficile fidarsi di una persona o di una organizzazione che non si è mai incontrata prima, e molti acquirenti si sentono al sicuro sapendo che la propria compagnia di carte di credito può annullare un pagamento su richiesta [27].

Le transazioni in criptovalute non sono reversibili, ed il supporto legale a disposizione di acquirenti o venditori è limitato quando qualcosa va storto, specialmente per Monero, che non permette un'analisi semplice della blockchain [45]. Il fondamento dello shopping online sicuro con criptovalute è rappresentato dagli scambi con garanzia (escrow) basati sulle multifirme 2-su-3 (2-of-3 multisignature), che consentono a terze parti di mediare le controversie. Affidandosi a queste terze parti, anche venditori e acquirenti completamente anonimi possono interagire senza formalità.

Poiché le interazioni multisig in Monero possono risultare piuttosto complesse (si veda la Sezione 9.4.2), dedichiamo questo capitolo alla descrizione di un ambiente di acquisto con garanzia il più efficiente possibile.[1,2]

[1] René "rbrunner7" Brunner, un contributore di Monero che ha creato il MMS [115, 114], ha studiato l'integrazione del multisig di Monero nel marketplace digitale decentralizzato basato su criptovalute OpenBazaar `https://openbazaar.org/`. I concetti presentati qui sono ispirati agli ostacoli incontrati da René [116] (la sua 'analisi preliminare').

[2] L'impressione iniziale degli autori è che l'attuale implementazione multisig di Monero abbia già un flusso di

10.1 Caratteristiche Essenziali

Esistono diversi requisiti e funzionalità di base per rendere più fluide le interazioni tra acquirenti e venditori online. Prendiamo questi punti dall'indagine di René su OpenBazaar [116], in quanto sono ragionevoli ed estendibili.

- *Vendita offline*: Un acquirente dovrebbe poter accedere al negozio online di un venditore e effettuare un ordine anche quando il venditore è offline. Ovviamente, il venditore dovrà poi andare online per confermare l'ordine ed evaderlo.[3]

- *Pagamenti basati su ordini di acquisto*: Gli indirizzi per la ricezione dei fondi del venditore sono univoci per ciascun ordine di acquisto, in modo da poter abbinare univocamente ordini e pagamenti.

- *Acquisto ad alta fiducia*: Un acquirente può, se si fida del venditore, pagare un prodotto in anticipo, dunque prima che questo venga evaso e consegnato.
  - *Pagamento diretto online*: Dopo aver verificato che il venditore sia online e che la sua inserzione sia disponibile, l'acquirente invia il denaro in una singola transazione all'indirizzo fornito dal venditore, il quale segnala quindi che l'ordine è in fase di evasione.
  - *Pagamento offline*: Se un venditore è offline, l'acquirente crea e finanzia un indirizzo multisig 1-su-2 con abbastanza fondi per coprire l'acquisto desiderato. Quando il venditore torna online, può prelevare i fondi dall'indirizzo multisig (restituendo l'eventuale resto all'acquirente) ed evadere l'ordine. Se il venditore non torna mai online (o ad esempio dopo un periodo di attesa ragionevole), oppure se l'acquirente cambia idea prima che ciò accada, l'acquirente può svuotare l'indirizzo multisig 1-su-2 ritrasferendo i fondi nel proprio portafoglio personale.

- *Acquisto moderato*: Viene costruito un indirizzo multisig 2-su-3 tra acquirente, venditore e un moderatore scelto di comune accordo. L'acquirente finanzia questo indirizzo prima che il venditore evada l'ordine, e una volta che il prodotto è stato consegnato, due delle tre parti collaborano per sbloccare i fondi. Se venditore e acquirente non raggiungono un accordo, possono affidarsi al giudizio del moderatore.

Non tratteremo l'acquisto ad alta fiducia, in quanto, non essendo necessarie comunicazioni complesse, le funzionalità risultano piuttosto banali.[4]

---

creazione delle transazioni simile a quello necessario per un ambiente di acquisto con garanzia, il che è una buona notizia per eventuali sforzi di implementazione. I lettori dovrebbero notare che il multisig di Monero necessita di alcuni aggiornamenti di sicurezza prima di poter essere adottato su larga scala [101].

[3] Evadere un ordine di acquisto significa spedire il prodotto affinché venga consegnato all'acquirente.

[4] Il multisig 1-su-2 può sfruttare alcuni concetti utili anche per il multisig 2-su-3, in particolare per quanto riguarda la costruzione iniziale dell'indirizzo.

10.1.1 Flusso di Lavoro per l'Acquisto

Tutti gli acquisti dovrebbero seguire la stessa sequenza di passaggi, supponendo che tutte le parti agiscano con la dovuta diligenza. Alcuni passaggi riguardano ciò che un moderatore dovrebbe aspettarsi quando interviene, ad esempio chiedere all'acquirente se ha richiesto un rimborso prima di richiedere il suo intervento.

1. Un acquirente accede al negozio online del venditore, individua un prodotto da acquistare, seleziona 'Acquista', rende disponibili i fondi per tale acquisto, ed infine invia l'ordine al venditore.

2. Il venditore riceve l'ordine, verifica che il prodotto sia disponibile e che i fondi siano sufficienti, dunque restituisce eventualmente il denaro all'acquirente oppure evade l'ordine inviando il prodotto e notificando l'acquirente con una ricevuta.
   - Per un multisig 2-su-3, l'acquirente può facoltativamente autorizzare il pagamento al momento della notifica di evasione.

3. L'acquirente riceve il prodotto come previsto, oppure non lo riceve in tempo o riceve un prodotto difettoso.
   - *Prodotto conforme*: L'acquirente può lasciare un feedback facoltativo al venditore.
     - *L'acquirente lascia un feedback*: Il feedback può essere positivo o negativo.
       - *Feedback positivo*: Se si tratta di un pagamento multisig 2-su-3 non ancora finalizzato, questo è il momento in cui l'acquirente conferma il pagamento al venditore. Altrimenti, è semplicemente una recensione positiva. [FINE DEL FLUSSO]
       - *Feedback negativo*: Se si tratta di un pagamento multisig 2-su-3, si passa al flusso di 'prodotto non conforme'. Altrimenti è solo una recensione negativa.
     - *L'acquirente non fa nulla*: O il venditore è già stato pagato, oppure si tratta di un multisig 2-su-3 e ha bisogno della cooperazione di qualcuno per sbloccare i fondi.
       - *Il venditore è stato pagato*: [FINE DEL FLUSSO]
       - *Il venditore non è stato pagato*: Il venditore tenta di ottenere il pagamento.
         (a) Il venditore contatta l'acquirente chiedendo il pagamento (o inviando un promemoria).
         (b) L'acquirente può rispondere o meno.
           - *L'acquirente risponde*: Può effettuare il pagamento, oppure richiedere un rimborso.
             > *L'acquirente effettua il pagamento*: [FINE DEL FLUSSO]
             > *L'acquirente richiede un rimborso*: Passare al flusso di 'prodotto non conforme'.
           - *L'acquirente non risponde*: Il venditore coinvolge il moderatore per sbloccare i fondi. [FINE DEL FLUSSO]
   - *Nessun prodotto o prodotto difettoso*: L'acquirente richiede un rimborso.

(a) L'acquirente contatta il venditore chiedendo un rimborso, eventualmente fornendo una spiegazione.

(b) Il venditore accetta la richiesta di rimborso, la contesta o la ignora.
   - *Il venditore accetta*: Il denaro viene rimborsato all'acquirente. [FINE DEL FLUSSO]
   - *Il venditore contesta*: Può trattarsi di un pagamento multisig 2-su-3 o meno.
     - ∗ *Non è un multisig 2-su-3*: [FINE DEL FLUSSO]
     - ∗ *È un multisig 2-su-3*: L'acquirente può rinunciare alla richiesta di rimborso (esplicitamente o non rispondendo in tempo), oppure insistere.
       - · *L'acquirente rinuncia*: Può aver autorizzato il pagamento al venditore o meno.
         > *Ha autorizzato il pagamento*: [FINE DEL FLUSSO]
         > *Non ha autorizzato il pagamento*: Il venditore contatta il moderatore, che autorizza il pagamento. [FINE DEL FLUSSO]
       - · *L'acquirente insiste*: Il venditore o l'acquirente contatta il moderatore, che coopera con le parti per giudicare la situazione. [FINE DEL FLUSSO]

10.2 Multisig Monero Trasparente

Possiamo sfruttare il naturale flusso di lavoro degli ordini di acquisto per integrare quasi tutte le parti di un'interazione multisig Monero 2-su-3 senza che i partecipanti se ne accorgano. C'è solo un piccolo passaggio aggiuntivo per il venditore alla fine, in cui deve firmare e inviare la transazione finale per ricevere il pagamento, in modo analogo a "svuotare la cassa".[5]

10.2.1 Basi dell'Interazione Multisig

Tutte le interazioni multisig 2-su-3 contengono lo stesso insieme di round di comunicazione, che coinvolgono la configurazione dell'indirizzo e la costruzione della transazione. Per rispettare il normale flusso di lavoro, utilizziamo un processo riorganizzato di costruzione della transazione rispetto al capitolo dedicato al multisig (si vedano le Sezioni 9.4.2 e 9.6.2).[6]

1. *Configurazione dell'indirizzo*

   (a) Tutti gli utenti devono innanzitutto conoscere le chiavi di base degli altri partecipanti, che useranno per costruire segreti condivisi. Trasmettono le chiavi pubbliche di questi segreti condivisi (ad es. $K^{sh} = \mathcal{H}_n(k_A^{base} * k_B^{base} G)G$) agli altri utenti.

[5] Estendere questo schema oltre (N-1)-su-N è probabilmente irrealizzabile senza ulteriori passaggi, a causa dei round aggiuntivi necessari per configurare un indirizzo multisig con soglia inferiore.

[6] Questa procedura è in realtà abbastanza simile a come Monero organizza attualmente le transazioni multisig.

(b) Dopo aver appreso tutte le chiavi pubbliche dei segreti condivisi, ciascun utente può eseguire le operazioni `premerge` e poi `merge` per ottenere la chiave pubblica di spesa dell'indirizzo. Le chiavi private aggregate di spesa saranno utilizzate per firmare le transazioni. Un hash della chiave privata del segreto condiviso tra i firmatari principali (acquirente e venditore), ad es. $k^{sh} = \mathcal{H}_n(k_A^{base} * k_B^{base} G)$, sarà utilizzato come chiave di visualizzazione (ad es. $k^v = \mathcal{H}_n(k^{sh})$). Saranno approfonditi questi dettagli in seguito (Sezione 10.2.2).

2. *Costruzione della transazione*: Si assume che l'indirizzo possieda almeno un output e che le immagini delle chiavi siano sconosciute. Ci sono due firmatari: l'iniziatore e il co-firmatario.

   (a) *Inizio della transazione*: L'iniziatore decide di avviare una transazione. Genera valori di apertura (ad es. αG) per tutti gli output posseduti (non è ancora sicuro di quali verranno usati), e li comunica. Crea anche immagini parziali delle chiavi per quei determinati output, e le firma con una prova di legittimità (vedi Sezione 9.5). Invia queste informazioni, insieme al proprio indirizzo personale per la ricezione di fondi (ad es. per il resto, se appropriato), al co-firmatario.

   (b) *Creazione di una transazione parziale*: Il co-firmatario verifica che tutte le informazioni fornite siano valide. Decide i destinatari degli output e gli importi (che possono essere parzialmente basati sulle raccomandazioni dell'iniziatore), gli input da usare e i relativi decoy dei ring member, e la fee della transazione. Genera una chiave privata per la transazione (o più chiavi se sono coinvolti sottoindirizzo), crea indirizzi una tantum, commitment sugli output, importi codificati, maschere di commitment sugli pseudo output, e valori di apertura per i commitment a zero. Per dimostrare che gli importi sono in range, costruisce la Bulletproof per tutti gli output. Genera anche valori di apertura per la propria firma (ma non li comunica), scalari casuali per le firme MLSAG, immagini parziali delle chiavi per gli output posseduti, e prove di legittimità per tali immagini. Tutto questo viene inviato all'iniziatore.

   (c) *Firma parziale dell'iniziatore*: L'iniziatore verifica che le informazioni della transazione parziale siano valide e conformi alle sue aspettative (ad es. importi e destinatari sono corretti). Completa le firme MLSAG e le firma con le sue chiavi private, quindi invia la transazione parzialmente firmata al co-firmatario insieme ai propri valori di apertura rivelati.

   (d) *Completamento della transazione*: Il co-firmatario completa la firma della transazione e la invia alla rete.

Firma a Impegno Singolo (Single-commitment signing)

A differenza di quanto raccomandato nel capitolo sul multisig, viene fornito un solo impegno per transazione parziale (da parte dell'iniziatore della transazione), e viene rivelato solo dopo che il cofirmatario ha esplicitamente inviato il proprio valore di apertura. Lo scopo dell'impegno ai valori di apertura (es. αG) è impedire a un cofirmatario malevolo di utilizzare il proprio valore

di apertura per influenzare la sfida che verrà prodotta, il che potrebbe permettergli di scoprire le chiavi di aggregazione degli altri cofirmatari (si veda la Sezione 9.3). Se anche solo un valore di apertura parziale non è disponibile quando l'attore malevolo genera il proprio, allora è impossibile (o almeno trascurabilmente probabile) per lui avere un controllo sulla generazione della sfida.[7,8,9]

Semplificare in questo modo rimuove un round di comunicazione, il che ha conseguenze importanti per l'esperienza d'interazione tra acquirente e venditore.

10.2.2 Esperienza Utente con Escrow

Segue una descrizione dettagliata delle interazioni tra acquirente, venditore e moderatore in un acquisto online basato su multisig 2-di-3 utilizzando Monero.

1. *L'acquirente effettua un acquisto*
   (a) *Nuova sessione di acquisto dell'acquirente*: Un'acquirente entra in un marketplace online, e il suo client genera un nuovo sottoindirizzo da utilizzare se inizia un nuovo ordine di acquisto.[10] In quel marketplace trova venditori, ciascuno dei quali offre una selezione di prodotti e prezzi. Invisibili all'acquirente ma visibili al suo client (cioè al software che sta usando per acquistare), ogni prodotto ha una chiave base per l'acquisto multisig. Accanto a essa c'è un elenco di moderatori preselezionati, ognuno dei quali possiede una chiave base e una chiave pubblica di un segreto condiviso precalcolato tra venditore e moderatore.[11]

[7] Questo è anche il motivo per cui l'iniziatore rivela i propri valori di apertura solo dopo che tutte le informazioni sulla transazione sono state determinate, affinché nessuno dei firmatari possa alterare il messaggio MLSAG e influenzare la sfida.

[8] La firma con singolo impegno potrebbe essere generalizzata come firma con (M-1) impegni, in cui solo l'autore della transazione parziale non si impegna e rivela, mentre gli altri cofirmatari rivelano solo dopo che la transazione è stata completamente determinata. Per esempio, supponiamo che ci sia un indirizzo 3-di-3 con cofirmatari (A, B, C), che tentano la firma con singolo impegno. I firmatari B e C sono in una coalizione malevola contro A, mentre C è l'iniziatore e B è l'autore della transazione parziale. C inizia con un impegno, poi A fornisce il proprio valore di apertura (senza impegno). Quando B costruisce la transazione parziale, può cospirare con C per controllare la sfida della firma al fine di esporre la chiave privata di A. Si noti anche che la firma con (M-1) impegni è un concetto originale presentato qui per la prima volta, e non è supportato da alcuna ricerca avanzata o implementazione di codice. Potrebbe risultare completamente errato.

[9] Un modo per pensare a questo è considerare il significato e lo scopo di un "impegno" (si veda la Sezione 5.1). Una volta che Alice si impegna al valore A, ne è vincolata, e non può trarre vantaggio da nuove informazioni derivanti dall'evento B (causato da Bob) che avviene successivamente. Inoltre, se A non è stato rivelato, allora B non può esserne influenzato. Alice e Bob possono essere certi che A e B siano indipendenti. Sosteniamo che la firma con singolo impegno, come descritta, soddisfi questo standard ed è equivalente alla firma con impegno completo. Se l'impegno c è una funzione unidirezionale dei valori di apertura $\alpha_A G$ e $\alpha_B G$ (es. $c = \mathcal{H}_n(\alpha_A G, \alpha_B G)$), allora se $\alpha_A G$ viene inizialmente impegnato, $\alpha_B G$ viene rivelato dopo che $C(\alpha_A G)$ appare, e $\alpha_A G$ viene rivelato dopo $\alpha_B G$, allora $\alpha_B G$ e $\alpha_A G$ sono indipendenti, e c sarà casuale sia dal punto di vista di Alice che di Bob (a meno che non collaborino, ed eccetto con probabilità trascurabile).

[10] Usare un nuovo sottoindirizzo per ogni ordine di acquisto, o addirittura per ogni venditore o prodotto del venditore, rende più difficile per i venditori tracciare il comportamento dei clienti. Aiuta anche a garantire l'unicità di ogni ordine, ad esempio nel caso in cui si acquisti due volte lo stesso oggetto.

[11] Sarebbe semplice per i venditori includere, invisibilmente per gli acquirenti, commitment ai valori di apertura delle transazioni. Tuttavia, per gestire più ordini per lo stesso prodotto, dovrebbero fornire molti commitment in anticipo per ogni potenziale acquirente. Questo potrebbe diventare molto disordinato. È qui che entra in gioco la nostra semplificazione della firma con commitment singolo.

(b) *L'acquirente aggiunge un prodotto al carrello*: L'acquirente decide di acquistare qualcosa, seleziona l'opzione di pagamento (cioè pagamento diretto, multisig 1-di-2, o multisig 2-di-3), e se seleziona multisig 2-di-3 viene presentata una lista di moderatori disponibili tra cui scegliere. Quando aggiunge il prodotto al carrello, il suo client, in modo trasparente (e supponendo abbia selezionato multisig 2-di-3), utilizza la chiave base del prodotto, quella del moderatore, e la chiave pubblica del segreto condiviso tra venditore e moderatore, combinandole con la chiave di spesa del sottoindirizzo di sessione dell'acquirente (come chiave base) per costruire un indirizzo multisig 2-di-3 acquirente-venditore-moderatore.[12]

La chiave di visualizzazione è l'hash del segreto condiviso privato tra acquirente e venditore (non della chiave privata aggregata, cioè prima di `premerge`), mentre la chiave di cifratura per le comunicazioni tra acquirente e venditore è un hash della chiave di visualizzazione.[13]

(c) *L'acquirente procede al checkout*: L'acquirente visualizza il carrello con tutti i prodotti e decide di procedere al checkout. A questo punto rende i fondi disponibili prima di finalizzare l'ordine. Il client costruisce una transazione (ma non la firma ancora) che pagherà direttamente il venditore, oppure finanzierà un indirizzo multisig (aggiungendo una piccola somma per le fee future). Se si finanzia un indirizzo multisig 2-di-3, il client inizializza anche due transazioni per prelevare fondi da quell'indirizzo. Una potrà essere usata per pagare il venditore, l'altra per rimborsare l'acquirente. Le partial key images degli input sono basate sulla transazione di finanziamento non ancora firmata.

In realtà, servono solo i valori di apertura impegnati per le due transazioni, e separatamente una copia delle partial key images (con prova di legittimità) e una copia del sottoindirizzo di sessione dell'acquirente. Quel sottoindirizzo ha un doppio scopo: è l'indirizzo dell'acquirente per rimborsi o resti, e la sua chiave di spesa è la chiave base multisig dell'acquirente.[14]

(d) *L'acquirente autorizza il pagamento*: Dopo aver esaminato tutti i dettagli dell'ordine, l'acquirente lo autorizza.[15] Il client completa la firma della transazione di finanziamento e la invia alla rete.[16] Invia l'ordine di acquisto, insieme all'hash della transazione di finanziamento, le transazioni multisig inizializzate, e la chiave pubblica del segreto condiviso acquirente-moderatore, al venditore.[17]

[12] Il modo in cui un marketplace dovrebbe essere implementato è aperto a interpretazioni; ad esempio la scelta del tipo di pagamento potrebbe essere mostrata all'utente durante il checkout anziché durante l'aggiunta al carrello.

[13] Lo stesso processo avverrebbe per il multisig 1-di-2, escludendo il moderatore.

[14] È importante inizializzare transazioni separate, poiché i valori di apertura impegnati possono essere usati una sola volta.

[15] Se l'acquirente annulla l'ordine, la sua transazione di finanziamento e le transazioni multisig parziali vengono eliminate.

[16] Se il carrello contiene prodotti di più venditori, il client può creare più ordini separati. I venditori possono essere tutti pagati dalla stessa transazione di finanziamento.

[17] Il client dell'acquirente dovrebbe tenere traccia dei dettagli dell'ordine come il prezzo totale, per verificare in seguito il contenuto delle transazioni multisig prima di firmarle.

2. *Il venditore evade l'ordine*

   (a) *Il venditore valuta l'ordine*: Il venditore esamina l'ordine e lo approva per la spedizione. Se ha ricevuto un pagamento diretto non deve fare altro. Se il pagamento è multisig 1-di-2 può creare una transazione per prelevare da quell'indirizzo. Per multisig 2-di-3 il client genera un sottoindirizzo per ricevere il pagamento, e costruisce due transazioni parziali a partire da quelle inizializzate dall'acquirente. La transazione di pagamento invia un importo pari al prezzo del prodotto al venditore e il resto all'acquirente, mentre quella di rimborso restituisce tutto all'acquirente.[18] Nota che l'indirizzo multisig viene ricostruito usando le chiavi base acquirente-venditore-moderatore e la chiave pubblica del segreto condiviso acquirente-moderatore.

   (b) *Il venditore spedisce il prodotto*: Il venditore spedisce il prodotto e invia una notifica di completamento all'acquirente. Questa notifica include una ricevuta dell'acquisto, e una richiesta per completare il pagamento (da qui in avanti si assume multisig 2-di-3). Invisibili all'utente, ci sono il sottoindirizzo di ordine del venditore, utile per una disputa, e le due transazioni parziali.

3. *L'acquirente completa il pagamento o richiede un rimborso*

   (a) *L'acquirente invia la transazione firmata parzialmente*: L'acquirente decide se completare il pagamento o richiedere un rimborso. Il suo client crea una firma parziale sulla transazione appropriata e la invia al venditore. Un eventuale rimborso potrebbe essere accompagnato da una giustificazione.

   (b) *Il venditore completa la transazione*: Il venditore riceve la transazione parzialmente firmata, la completa, e la invia alla rete. Se necessario, invia una notifica di rimborso con prova all'acquirente.

4. *Disputa con moderatore*: In qualsiasi momento dopo che l'acquirente ha inviato un ordine e prima che l'indirizzo multisig venga svuotato, il venditore o l'acquirente possono decidere di coinvolgere il moderatore. Party_A è chi solleva la disputa, Party_B è il convenuto.[19]

   (a) *Party_A contatta il moderatore*: Party_A inizializza due transazioni (per pagamento o rimborso), questa volta pensate per la firma Party_A-moderatore, e le invia al moderatore con tutte le informazioni necessarie per ricostruire l'indirizzo multisig (chiavi base, chiave pubblica del segreto condiviso Party_A-Party_B, chiave privata di visualizzazione) e leggere il saldo (partial key images e prove).

   (b) *Il moderatore gestisce la disputa*

      i. *Il moderatore prende in carico la disputa*: Riconosce di aver ricevuto la disputa, crea le transazioni parziali dai dati ricevuti, e le invia a Party_A. Notifica anche Party_B e inizia due transazioni con lui nel caso Party_A non collabori con il verdetto finale.

[18] Le transazioni parziali potrebbero condividere molti valori poiché usano gli stessi input, ma solo una di esse verrà firmata. Per modularità e robustezza progettuale è preferibile gestirle separatamente.

[19] Il nostro design per la risoluzione delle dispute presume buona fede. Attori non collaborativi renderanno il processo più complicato.

ii. *Il moderatore valuta il caso*: Analizza le prove disponibili e può interagire con le parti per ottenere ulteriori informazioni. Potrebbe tentare una mediazione.

iii. *La disputa si conclude*: Le parti possono risolvere autonomamente oppure il moderatore emette un verdetto che comunica a entrambi.

- Nota: Se la parte convenuta deve ricevere fondi ma non ha fornito un indirizzo, il moderatore può contattarla anche dopo la chiusura della disputa per completare il trasferimento.

(c) *Party_A o B accetta il verdetto*: Se nessuna transazione Party_A-Party_B è stata finalizzata, la disputa si conclude con la decisione del moderatore.

i. *Party_A accetta*

A. Party_A completa la propria firma parziale sulla transazione del verdetto e la invia al moderatore.

B. Il moderatore completa la firma e invia la transazione alla rete.

ii. *Party_B accetta*

A. Party_B crea una transazione parziale basata sul verdetto inizializzato dal moderatore, e la invia al moderatore. Questo può avvenire anche prima che il verdetto sia definito, preparando entrambe le possibilità.

B. Il moderatore firma parzialmente la transazione e la rimanda a Party_B.

C. Party_B completa la firma e invia la transazione alla rete. Invia l'hash della transazione al moderatore.

(d) *Il moderatore chiude la disputa*: Riassume la disputa e la sua risoluzione, inviando il report a venditore e acquirente.

In seguito sono trattate quattro principali ottimizzazioni progettuali.

Moderatori Preselezionati

Selezionando i moderatori in anticipo, i venditori possono creare un segreto condiviso con ciascuno di essi per ogni loro prodotto e pubblicarne la chiave pubblica insieme alle informazioni del prodotto.[20] In questo modo gli acquirenti possono costruire l'indirizzo multisig completo in un solo passaggio, non appena decidono di acquistare qualcosa, in linea con il requisito della 'vendita offline'. Preselezionare più moderatori consente agli acquirenti di scegliere quello di cui si fidano di più.

Gli acquirenti possono anche, se non si fidano dei moderatori accettati da un venditore, cooperare con un moderatore online di loro scelta per creare un indirizzo multisig utilizzando la chiave base

[20] È importante notare che questi indirizzi multisig sono comunque resistenti ai test di aggregazione delle chiavi, poiché i segreti condivisi con l'acquirente sono sconosciuti agli osservatori.

del prodotto del venditore. Dopo aver ricevuto un ordine d'acquisto, il venditore può accettare quel nuovo moderatore oppure rifiutare la vendita.[21]

Si suppone che il desiderio reciproco, da parte di acquirenti e venditori, di buoni moderatori porti nel tempo alla creazione di una gerarchia di moderatori organizzata in base alla qualità e all'equità del servizio fornito. I moderatori di qualità inferiore o con minore reputazione probabilmente guadagnerebbero meno o gestirebbero transazioni meno significative.[22]

Sottoindirizzi e ID dei Prodotti

I venditori creano una nuova chiave base per ogni linea di prodotti o ID, e tali chiavi vengono usate per costruire indirizzi multisig 2-di-3.[23] Quando i venditori evadono un ordine d'acquisto, creano un sottoindirizzo unico per la ricezione dei fondi, che può essere usato per abbinare gli ordini d'acquisto ai pagamenti ricevuti.

Il requisito dei 'pagamenti basati su ordine d'acquisto' è soddisfatto in modo efficiente, soprattutto perché i fondi inviati a sottoindirizzi differenti sono accessibili in modo immediato dallo stesso portafoglio (vedi Sezione 4.3).

Transazioni Parziali Anticipate

Le transazioni multisig richiedono più passaggi rispetto a quelle tradizionali, quindi è consigliato inizializzarle il prima possibile. Per comodità dell'utente, le transazioni parziali raramente usate (ad es. rimborsi) possono essere create in anticipo, in modo da essere immediatamente disponibili per la firma qualora si presenti la necessità.

Accesso Condizionale del Moderatore

Per motivi di efficienza e privacy, i moderatori hanno bisogno di accedere ai dettagli di una transazione solo in caso di disputa. Per ottenere ciò, rendiamo la chiave di visualizzazione privata del multisig un hash del segreto condiviso privato tra acquirente e venditore: $k^{v,grp}_{purchase-order} =$

[21] Per praticità, un servizio di escrow potrebbe essere 'sempre online', e invece di usare moderatori preselezionati, tutti gli indirizzi multisig 2-di-3 vengono creati attivamente con tale servizio al momento dell'ordine. Un'altra possibilità è usare multisig annidato (Sezione 9.7), dove il moderatore preselezionato è in realtà un gruppo multisig 1-di-N. In questo modo, ogni volta che sorge una disputa, un qualsiasi moderatore disponibile di quel gruppo può intervenire. La realizzazione di questa opzione richiederebbe probabilmente uno sforzo di sviluppo considerevole.

[22] Non ci è chiaro quale sia il metodo di finanziamento migliore, o più probabile, per i moderatori. Forse riceveranno un compenso fisso o una percentuale per ogni transazione moderata o per ogni transazione in cui vengono aggiunti come moderatori (e poi, se il compenso non è previsto nelle transazioni parziali originali, potrebbero rifiutarsi di collaborare in caso di disputa), oppure utenti e/o venditori e/o piattaforme di marketplace potrebbero stipulare contratti con loro.

[23] Questa chiave base è utilizzata anche per gli acquisti 1-di-2 multisig. Riteniamo importante non esporre la chiave di spesa privata nel canale di comunicazione, quindi usare un segreto condiviso tra acquirente e venditore ha molto senso.

$\mathcal{H}_n(T_{mv}, k_{AB}^{sh,2x3})$, dove T_{mv} è il separatore di dominio per la chiave di visualizzazione del marketplace, e A e B corrispondono rispettivamente a venditore e acquirente, e $k_{AB}^{sh,2x3} = \mathcal{H}_n(k_A^{base} * k_B^{base} G)$. Estendiamo ciò che questa chiave può 'visualizzare' includendo anche il transcript della comunicazione tra acquirente e venditore. In altre parole, la chiave di cifratura della comunicazione è $k_{purchase-order}^{ce} = \mathcal{H}_n(T_{me}, k_{purchase-order}^{v,grp})$ (T_{me} è il separatore di dominio per la chiave di cifratura del marketplace).[24,25]

I moderatori ottengono accesso alle comunicazioni tra acquirente e venditore, e la possibilità di autorizzare pagamenti, solo quando una delle parti rilascia loro la chiave di visualizzazione.[26]

Inoltre, i venditori possono verificare che l'host del marketplace (che potrebbe anche essere l'unico moderatore disponibile, a seconda di come viene implementato questo concetto) non stia effettuando un attacco MITM ('man in the middle') nelle loro conversazioni con i clienti (cioè fingendo di essere l'acquirente o il venditore) controllando che le chiavi base pubblicate per ciascun prodotto corrispondano a quelle effettivamente visualizzate. Poiché la chiave base dell'acquirente, che viene usata per creare l'indirizzo multisig, fa anche parte della chiave di cifratura, un host malevolo avrebbe notevoli difficoltà a orchestrare un attacco MITM.

[24] Separare la chiave di visualizzazione dalla chiave di cifratura consente di concedere solo i diritti di visualizzazione del registro delle comunicazioni, senza dare accesso alla cronologia delle transazioni dell'indirizzo multisig.

[25] Questo metodo è usato anche per indirizzi multisig 1-di-2.

[26] È importante che i moderatori verifichino che il registro delle comunicazioni ricevuto non sia stato alterato. Un modo possibile è far includere a ciascun cofirmatario un hash firmato del registro dei messaggi ogni volta che inviano un nuovo messaggio. I moderatori possono esaminare il botta e risposta, e la serie di hash registrati, per identificare eventuali discrepanze. Questo aiuterebbe anche i cofirmatari a individuare messaggi che non sono stati trasmessi correttamente, o a creare prove che certi messaggi sono stati effettivamente ricevuti da determinati firmatari.

CAPITOLO 11

Transazioni Monero Congiunte (TxTangle)

Esistono diverse e inevitabili euristiche del grafo delle transazioni dovute alla natura di diverse entità e scenari. In particolare, il comportamento di miner, pool (Sezione 5.1 di [90]), marketplace in escrow e exchange presenta schemi chiari e aperti all'analisi anche all'interno del protocollo di firme ad anello di Monero.

In questo capitolo verrà trattato *TxTangle*, analogo a CoinJoin di Bitcoin [2], un metodo per confondere tali euristiche.[1] In sostanza, diverse transazioni vengono compresse in un'unica transazione, facendo sì che i modelli di comportamento di ciascun partecipante si mescolino.

Per raggiungere tale offuscamento, deve essere irragionevolmente difficile per gli osservatori utilizzare le informazioni contenute in una transazione congiunta per raggruppare input e output, associarli ai singoli partecipanti o sapere quanti partecipanti ci fossero effettivamente.[2] Inoltre, anche i partecipanti stessi non dovrebbero essere a conoscenza del numero di partecipanti, o essere in grado di raggruppare gli input e gli output degli altri partecipanti a meno che non controllino tutti i raggruppamenti tranne uno.[3] Infine, dovrebbe essere possibile costruire transazioni congiunte senza fare affidamento su un'autorità centrale [55]. Fortunatamente, tutti questi requisiti possono essere soddisfatti in Monero.

[1] Questo capitolo costituisce la proposta per un protocollo di transazioni congiunte. Al momento della stesura, nessun protocollo di questo tipo è stato implementato. Una proposta precedente, denominata MoJoin e creata da un ricercatore del Monero Research Lab (MRL) sotto lo pseudonimo di Sarang Noether, richiedeva un dealer affidabile per funzionare. Tale dealer sembra essere in conflitto con l'impegno fondamentale del progetto Monero per la privacy e la fungibilità, e quindi MoJoin non è stato ulteriormente sviluppato.

[2] Poiché in Bitcoin gli importi sono chiaramente visibili, è spesso possibile raggruppare gli input e gli output di CoinJoin in base alle somme degli importi. [31]

[3] L'inquinamento delle transazioni congiunte da parte di un attore malevolo è un potenziale attacco a questo metodo, identificato per la prima volta per CoinJoin. [88]

11.1 Costruzione di Transazioni Congiunte

In una transazione normale, input e output sono collegati utilizzando la dimostrazione sul bilanciamento degli importi. Dalla Sezione 6.2.1, la somma degli impegni pseudo-output è uguale alla somma degli impegni di output (più l'impegno della commissione):

$$\sum_j C'^a_j - (\sum_t C^b_t + fH) = 0$$

Una transazione congiunta banale potrebbe prendere tutto il contenuto di più transazioni e raggrupparlo in una sola. I messaggi MLSAG firmerebbero tutti i dati delle sotto-transazioni, e il bilanciamento degli importi verrebbe dimostrato in modo piuttosto ovvio $(0 + 0 + 0 = 0)$.[4] Tuttavia, i raggruppamenti di input e output potrebbero essere identificati altrettanto facilmente testando se i sottoinsiemi di input/output hanno importi bilanciati.[5]

Possiamo facilmente aggirare questo problema calcolando segreti condivisi tra ogni coppia di partecipanti, quindi aggiungendo questi offset alle maschere dei loro pseudo impegni di output (Sezione 5.4). In ogni coppia, un partecipante aggiunge il segreto condiviso a uno dei suoi pseudo impegni di output, e l'altro partecipante lo sottrae da uno dei *suoi* pseudo impegni. Quando sommati insieme, i segreti si annullano, e poiché ogni coppia di partecipanti ha un segreto condiviso, il bilanciamento degli importi appare solo dopo che tutti gli impegni sono stati combinati.[6]

I segreti condivisi possono nascondere i raggruppamenti di input/output nel senso immediato, ma i partecipanti devono in qualche modo conoscere tutti gli input e gli output, e il modo più semplice è se ognuno comunica i propri raggruppamenti individuali di input/output. Chiaramente questo viola la premessa iniziale, e in ogni caso implica che i partecipanti conoscano il numero totale di partecipanti.

11.1.1 Canale di Comunicazione a n Vie

Il numero massimo di partecipanti ad una transazione congiunta è pari al numero di output o input (a seconda di quale sia inferiore). Il modello prevede che ogni partecipante reale finge di essere una persona diversa per ogni output che sta inviando. Questo serve allo scopo di impostare un canale di comunicazione di gruppo con altri potenziali partecipanti, senza rivelare quanti partecipanti ci sono.

Si supponga n ($2 \leq n \leq 16$, anche se almeno 3 è raccomandato)[7] persone apparentemente non correlate si riuniscano a intervalli apparentemente casuali in una stanza, programmata per aprirsi

[4] Poiché le prove di intervallo Bulletproof sono effettivamente aggregate in una sola (Sezione 5.5), i partecipanti dovrebbero collaborare in una certa misura anche nel caso banale.

[5] I partecipanti potrebbero provare a dividere la commissione in modo fantasioso per confondere gli osservatori, ma questo approccio fallirebbe di fronte alla forza bruta poiché le commissioni non sono così grandi (circa 32 bit o meno).

[6] Compensiamo i pseudo impegni di output invece degli impegni di output poiché le maschere degli impegni di output sono costruite dall'indirizzo del destinatario (Sezione 5.3).

[7] Attualmente, una transazione può avere al massimo 16 output.

all'ora t_0 e chiudersi all'ora t_1 (solo 16 persone possono essere in una stanza alla volta, e la stanza ha condizioni come la priorità della commissione, la commissione base per byte [per semplificare il consenso attorno alla media attuale e alla ricompensa del blocco], e l'intervallo di tipi di transazione accettabili poiché, ad esempio, le monete di vecchi portafogli Monero non possono essere spese direttamente attraverso RingCT [130]). A t_1 tutti i membri fittizi segnalano il desiderio di procedere pubblicando una chiave pubblica, e la stanza viene convertita in un canale di comunicazione a n vie costruendo un segreto condiviso tra tutti i membri fittizi.[8] Questo segreto condiviso viene utilizzato per crittografare il contenuto dei messaggi, mentre i membri fittizi firmano i messaggi relativi all'input utilizzando firme SAG (Sezione 3.3) in modo che non sia mai chiaro chi ha inviato un determinato messaggio, e i messaggi relativi all'output con un bLSAG (Sezione 3.4) sull'insieme delle chiavi pubbliche dei membri fittizi in modo che gli output effettivi siano dissociati dai membri fittizi.[9]

11.1.2 Turni di Messaggi per Costruire una Transazione Congiunta

Dopo aver impostato il canale, le transazioni TxTangle possono essere costruite in cinque turni di comunicazione, dove il turno successivo può iniziare solo dopo che il precedente è terminato, e a ogni turno viene assegnato un intervallo di tempo entro il quale i messaggi devono essere pubblicati in modo casuale. Questi intervalli hanno lo scopo di prevenire cluster di messaggi che rivelerebbero i raggruppamenti di input/output.

1. Ogni membro fittizio genera privatamente uno scalare casuale per ogni output previsto e li firma con bLSAG. Un elenco ordinato di questi scalari viene utilizzato per determinare gli indici di output (si ricordi la Sezione 4.2.1; lo scalare più piccolo ottiene l'indice $t = 0$).[10] Pubblicano questi bLSAG, e anche SAG che firmano la versione della transazione degli input previsti. Dopo questo turno i partecipanti possono calcolare il peso della transazione in base al numero di input e output, e stimare accuratamente la commissione richiesta.[11,12,13]

[8] Il metodo multisig della Sezione 9.6.3 è un modo, estendendo M-di-N fino a 1-di-N.

[9] Ogni set separato di bLSAG di TxTangle dovrebbe utilizzare le stesse immagini chiave, poiché tutto ciò che è correlato a un dato output è collegato insieme.

[10] La selezione dell'indice di output dovrebbe corrispondere ad altre implementazioni di costruzione di transazioni per evitare il fingerprinting di diversi software. Utilizziamo questo approccio effettivamente casuale per allinearci con l'implementazione principale, che randomizza anche gli output.

[11] Se si scopre che ci devono essere solo due partecipanti reali, in base al confronto del numero di input e output con il proprio conteggio di input/output, il TxTangle può essere abbandonato. Si raccomanda che ogni partecipante abbia almeno due input e due output, in caso di attori malevoli che non abbandonano i TxTangle anche quando si rendono conto che ci sono solo due partecipanti. Questa raccomandazione è aperta al dibattito, poiché l'utilizzo di più input e output non è euristicamente neutro.

[12] Le transazioni TxTangle non dovrebbero avere informazioni superflue memorizzate nel campo extra (ad esempio, nessun ID di pagamento crittografato a meno che non sia solo un TxTangle a 2 output che dovrebbe avere almeno un ID di pagamento crittografato fittizio).

[13] La stima della commissione dovrebbe basarsi su un approccio standardizzato, in modo che ogni partecipante calcoli la stessa cosa. Altrimenti gli output potrebbero essere raggruppati in base al metodo di calcolo della commissione. Questo stesso standard di commissione dovrebbe essere implementato al di fuori del TxTangle, per promuovere che le transazioni TxTangle appaiano uguali alle transazioni normali.

2. Ogni membro fittizio utilizza l'elenco delle chiavi pubbliche per costruire un segreto condiviso con ogni altro membro per bilanciare le maschere dei loro pseudo impegni di output, e decide chi aggiungere o rimuovere in base alla chiave pubblica più piccola di ogni coppia.[14] Ogni membro fittizio deve pagare $1/n$ della stima della commissione (utilizzando la divisione intera). Al membro fittizio con l'indice di output più basso viene data la responsabilità di pagare il resto della divisione (importo infinitesimale, ma che deve essere preso in considerazione per prevenire il fingerprinting delle transazioni TxTangle). Ogni membro genera privatamente chiavi pubbliche di transazione per ciascuno dei loro output (da non inviare ancora agli altri membri), e costruiscono i loro impegni di output, gli importi cifrati e le prove parziali della Parte A da utilizzare per la prova di intervallo aggregata Bulletproof, firmando tutto con bLSAG (un impegno, un importo codificato e una prova parziale per messaggio bLSAG, e l'immagine chiave li collega all'elenco originale di scalari casuali che è stato utilizzato per specificare gli indici di output). I pseudo impegni di output vengono generati normalmente (Sezione 5.4), quindi bilanciati con i segreti condivisi e firmati con SAG. Dopo che i bLSAG e i SAG sono pubblicati, e supponendo che i partecipanti abbiano stimato la commissione totale nello stesso modo, possono ora verificare il bilancio complessivo degli importi.[15]

3. Se gli importi si bilanciano correttamente, inizia un breve turno aggiuntivo per costruire il Bulletproof aggregato che dimostra che tutti gli importi di output rientrano nell'intervallo. Ogni membro fittizio utilizza le prove parziali della Parte A e gli impegni di output del turno precedente, e calcola privatamente la sfida aggregata A. La utilizzano per costruire la loro prova parziale della Parte B, che inviano al canale con un bLSAG.

4. I partecipanti iniziano a compilare il messaggio da firmare con MLSAG (si ricordi la nota a piè di pagina nella Sezione 6.2.2). Gli offset dei membri dell'anello di ogni input e le immagini chiave sono firmati con un SAG e associati al corretto pseudo impegno di output. L'indirizzo monouso di ogni output, la chiave pubblica della transazione e la prova parziale della Parte C (calcolata in base alle prove parziali della Parte B e a una sfida aggregata B) sono firmati con un bLSAG (questi possono anche includere un componente casuale della chiave pubblica di transazione base, che come vedremo può essere utilizzato per una mitigazione fittizia di Janus). Questi due tipologie di messaggio sono pubblicate e trasmesse in ordine casuale durante l'intervallo di comunicazione.

5. I partecipanti utilizzano tutte le prove parziali per completare il Bulletproof aggregato e applicano privatamente una tecnica di prodotto interno logaritmico per comprimerlo per la prova finale da includere nei dati della transazione. Una volta raccolte tutte le informazioni da firmare con MLSAG, ogni partecipante completa i MLSAG dei propri input e li invia casualmente (con un SAG per ciascuno) al canale durante l'intervallo di comunicazione. Qualsiasi partecipante può inviare la transazione non appena ha tutti i pezzi.

[14] Poiché i punti sono compressi (Sezione 2.4.2), basta interpretare le chiavi come interi a 32 byte. Il proprietario della chiave più piccola aggiunge, e il proprietario della chiave più grande rimuove, per convenzione.

[15] Non vengono pubblicati gli importi separati della commissione pagati nel caso in cui un partecipante l'abbia calcolata male, il che potrebbe rivelare un cluster di output a causa di una collezione di importi di commissione non standard. Se gli importi non si bilanciano correttamente, la transazione TxTangle può essere abbandonata.

Chiavi Pubbliche della Transazione e Mitigazione di Janus

Se ogni partecipante a un TxTangle conosce la chiave privata della transazione r (Sezione 4.2), allora chiunque di essi può testare gli indirizzi di output monouso degli altri rispetto a un elenco di indirizzi noti. Per questo motivo è necessario costruire le transazioni TxTangle come se fossero destinate ad un sottoindirizzo (Sezione 4.3), includendo diverse chiavi pubbliche di transazione per ogni output.

Per supportare una possibile implementazione di una mitigazione per l'attacco di Janus relativa ai sottoindirizzi, in cui una chiave pubblica di transazione *base* aggiuntiva è inclusa nel campo extra [100], i TxTangle dovrebbero anche avere una chiave *base* fittizia composta da una somma di chiavi casuali generate da ogni membro fittizio.[16]

Molti partecipanti ad una TxTangle che inviano denaro a un sottoindirizzo avranno probabilmente almeno due output, uno dei quali reindirizza il resto al partecipante. Ciò significa che qualsiasi partecipante ad una TxTangle può ancora sfruttare la mitigazione di Janus rendendo la chiave pubblica della transazione del proprio resto anche la chiave *base* per il destinatario del sottoindirizzo.[17] Il destinatario del sottoindirizzo potrebbe rendersi conto che la transazione è un TxTangle e che la chiave *base* probabilmente corrisponde all'output di resto del mittente.[18]

11.1.3 Debolezze

Gli attori malevoli hanno due modi principali per vanificare gli effetti di una TxTangle, che cerca di nascondere i raggruppamenti di input/output da potenziali avversari/analisti. Degli attaccanti potrebbero inquinare le transazioni, in modo che il sottoinsieme di partecipanti onesti sia più piccolo (o addirittura inesistente) [88]. Potrebbero anche causare il fallimento dei tentativi di effettuare una TxTangle e utilizzare i tentativi successivi degli stessi partecipanti per stimare i raggruppamenti di input/output.

Il primo attacco non è facile da mitigare, specialmente nel caso di una decentralizzazione tale per cui nessun partecipante necessita di una reputazione minima per avviare una TxTangle. Una possibile applicazione di TxTangle può essere attuata con i pool collaborativi, che possono nascondere a quale pool appartengono i loro miner tra una collezione di pool. Tali pool conoscerebbero

[16] L'annullamento della chiave (Sezione 9.2.2) non dovrebbe essere un problema, poiché è solo una chiave fittizia e dovrebbe idealmente essere indicizzata casualmente all'interno dell'elenco delle chiavi pubbliche di transazione.

[17] Se si invia al proprio sottoindirizzo, non è necessaria la mitigazione di Janus. I portafogli abilitati per la mitigazione di Janus dovrebbero riconoscere che l'importo speso in un TxTangle è uguale all'importo ricevuto dal proprio sottoindirizzo, in modo da non notificare erroneamente all'utente un problema.

[18] Ciò presuppone che le chiavi pubbliche di transazione hanno una corrispendenza 1:1 con gli output, come apparentemente accade oggi. Se fosse prassi che le chiavi pubbliche di transazione siano in ordine casuale o ordinato all'interno del campo extra, allora le transazioni TxTangle e non-TxTangle sarebbero in gran parte indistinguibili per i destinatari dei sottoindirizzi. Ci sono casi particolari in cui i partecipanti ad una TxTangle non sono in grado di includere una chiave *base* (ad esempio, quando tutti i loro output sono a sottoindirizzi), o dove è chiaramente non-TxTangle poiché il destinatario del sottoindirizzo riceve la maggior parte o tutti gli output. Si noti che poiché le transazioni TxTangle avrebbero generalmente molti più output di una transazione tipica, questa euristica può essere utilizzata per differenziare i TxTangle dalle normali transazioni con sottoindirizzo.

i raggruppamenti di input/output, ma poiché lo scopo è aiutare i loro miner connessi, sarebbe opportuno per loro mantenere segrete le informazioni. Inoltre, tali TxTangle non consentirebbero la partecipazione di attori malevoli, supponendo che i pool siano onesti.

Il secondo attacco può essere eluso limitando i tentativi di effettuare una TxTangle, inserendo una pausa prima di eseguirne di nuovi, e rigenerando sempre la maggior parte degli elementi casuali di una transazione per nuovi tentativi. Questi elementi includono le chiavi pubbliche della transazione, le maschere dei pseudo impegni, gli scalari delle prove di intervallo e gli scalari MLSAG. In particolare, l'insieme delle esche (decoy) dell'anello per ogni input dovrebbe rimanere lo stesso per prevenire confronti incrociati che rivelino l'input reale. Se possibile, dovrebbero essere utilizzati input reali diversi per ogni tentativo di TxTangle. Poiché questa debolezza è inevitabile rende più accettabile l'argomento trattato nella prossima sezione.

11.2 TxTangle Ospitato

Il TxTangle totalmente decentralizzato presenta alcune questioni aperte da risolvere. Come vengono avviati e fatti rispettare i turni temporali? Come vengono create le stanze, affinché i partecipanti possano incontrarsi? Il modo più diretto è attraverso un host TxTangle, che crea e gestisce queste stanze.

Tale host sembrerebbe andare contro l'obiettivo della partecipazione offuscata, poiché ogni individuo deve connettersi e inviargli messaggi che potrebbero essere utilizzati per correlare i raggruppamenti di input/output (specialmente se l'host partecipa e conosce il contenuto dei messaggi). Possiamo usare una rete come I2P[19] per far sì che ogni messaggio ricevuto dall'host appaia come proveniente da un individuo unico.

11.2.1 Comunicazione di Base con un Host su I2P e Altre Funzionalità

Con I2P, gli utenti creano i cosiddetti *tunnel* che fanno passare messaggi fortemente crittografati attraverso i client di altri utenti prima di raggiungere la loro destinazione. Da quanto comprendiamo, questi tunnel possono trasportare più messaggi prima di essere distrutti e ricreati (ad esempio, sembra esserci un timer di 10 minuti sui tunnel). È essenziale per il nostro caso d'uso controllare attentamente quando vengono creati nuovi tunnel e quali messaggi possono uscire dallo stesso tunnel.[20]

1. *Richiesta di TxTangle*: nella nostra proposta originale ad n vie (Sezione 11.1.1) i partecipanti aggiungono gradualmente i loro membri fittizi alle stanze TxTangle disponibili prima

[19] The Invisible Internet Project (`https://geti2p.net/en/`).

[20] In I2P ci sono "tunnel in uscita" e "tunnel in entrata" (vedi `https://geti2p.net/en/docs/how/tunnel-routing`). Tutto ciò che viene ricevuto tramite un tunnel in entrata sembra provenire dalla stessa fonte anche se da più fonti, quindi in superficie sembrerebbe che gli utenti TxTangle non abbiano bisogno di creare tunnel diversi per tutti i loro casi d'uso. Tuttavia, se l'host TxTangle si rende il punto di ingresso per il proprio tunnel in entrata, allora ottiene l'accesso diretto ai tunnel in uscita dei partecipanti TxTangle.

che siano programmate per chiudere. Tuttavia, se un volume sufficientemente elevato di utenti tenta di effettuare TxTangle contemporaneamente, è probabile che si verifichi un'alta percentuale di fallimenti poiché gli utenti tentano di inserire casualmente tutti i loro output previsti nella stessa "stanza" TxTangle, ma poi le stanze si riempirebbero troppo presto e i partecipanti sarebbero costretti a ritirarsi. Un bel caos.

È possibile apportare un'ottimizzazione significativa comunicando all'host quanti output si posseggono (ad esempio, fornendogli un elenco delle nostre chiavi pubbliche di membri fittizi) e lasciandogli assemblare i partecipanti di ogni TxTangle. Poiché manteniamo ancora il protocollo di messaggistica bLSAG e SAG, l'host non sarà in grado di identificare i raggruppamenti di output nella transazione finale. Tutto ciò che sa è il numero di partecipanti e quanti output aveva ciascuno. Inoltre, in questo scenario gli osservatori non possono monitorare le stanze TxTangle aperte per dedurre informazioni sui partecipanti, un importante miglioramento della privacy. Si noti che la capacità dell'host di inquinare i TxTangle non è significativamente diversa dal design senza host, dunque questa modifica è neutrale rispetto a tale vettore di attacco.

2. *Metodo di comunicazione*: poiché l'host funge da centro di trasporto dei messaggi, è più semplice per lui gestire la comunicazione TxTangle. Durante ogni turno l'host raccoglie messaggi dai membri fittizi (ancora casualmente durante un intervallo di comunicazione), e alla fine di un turno c'è una breve fase di distribuzione dei dati in cui invia tutti i dati raccolti a ciascun partecipante, attendendo periodo di *buffer* prima del turno successivo per garantire che i messaggi vengano ricevuti e abbiano il tempo di essere elaborati.

3. *Tunnel e raggruppamenti input/output*: Una volta avviato un TxTangle, gli utenti devono dissociare le proprie identità fittizie dagli output effettivi che vengono creati. Ciò significa creare nuovi tunnel per i messaggi firmati bLSAG, dove ciascun tunnel può trasmettere solo messaggi relativi ad un output specifico (è accettabile trasmettere più messaggi simili attraverso lo stesso tunnel, poiché ovviamente le informazioni sullo stesso output provengono dalla stessa fonte). Devono anche creare nuovi tunnel per i messaggi firmati SAG relativi a input specifici.

4. *Minaccia di attacco MITM dell'host*: L'host potrebbe ingannare un partecipante fingendo di essere altri partecipanti, poiché controlla l'invio dell'elenco dei membri fittizi per la costruzione dei messaggi bLSAG e SAG. Ad esempio, l'elenco che l'host invia al partecipante A potrebbe contenere esclusivamente i membri fittizi del partecipante A e dell'host stesso. I messaggi ricevuti dal partecipante B potrebbero essere rifirmati utilizzando l'elenco di A prima di essere ritrasmessi ad A. Poiché tutti i messaggi firmati dall'elenco di A appartengono ad A, l'host avrebbe una conoscenza diretta dei raggruppamenti input/output di A!

Possiamo impedire all'host di agire come MITM delle interazioni oneste dei partecipanti modificando il modo in cui vengono create le chiavi pubbliche della transazione. I partecipanti si inviano a vicenda le rispettive chiavi pubbliche di transazione normalmente previste (con un bLSAG), quindi, in modo molto simile all'aggregazione robusta delle chiavi affrontata nella Sezione 9.2.3, le chiavi effettive che vengono incluse nei dati della transazione (e utilizzate

per creare le maschere degli impegni di output, ecc.) sono precedute da un hash dell'elenco dei membri fittizi. Definiamo dunque $\mathcal{H}_n(T_{agg}, \mathbb{S}_{mock}, r_t G) * r_t G$ come la t-esima chiave pubblica di transazione. Integrare l'elenco dei membri fittizi nella transazione stessa rende molto difficile completare i TxTangle senza comunicazione diretta tra tutti i partecipanti effettivi.[21]

11.2.2 Host come Servizio

È importante per la sostenibilità e il continuo miglioramento che un servizio TxTangle operi a scopo di lucro[22]. Anziché compromettere le identità degli utenti con un modello basato su account, l'host può partecipare a ogni TxTangle con un singolo output, e richiedere ai partecipanti di finanziarlo. Quando si accede all'*eepsite* dell'host per richiedere i TxTangle, gli utenti ricevono in risposta la tariffa per il servizio, che dovrebbe essere pagata per output.

I partecipanti, inoltre, sono responsabili del pagamento delle frazioni della commissione *e* della tariffa dell'host. In questa tipologia di servizio, la chiave pubblica del membro fittizio più piccola (esclusa la chiave dell'host) si prende il resto sia della commissione che della tariffa dell'host.[23] Poiché l'host non ha input, non ha pseudo impegni di output per annullare la maschera del suo impegno di output. Invece, crea segreti condivisi con gli altri membri fittizi come al solito, quindi separa la sua maschera di impegno reale in blocchi di dimensioni casuali per ogni altro membro fittizio e li divide per i segreti condivisi. Pubblica un elenco di tali scalari (corrispondendoli a ogni altro membro fittizio in base alla loro chiave pubblica), firmando con la sua chiave membro fittizia in modo che i partecipanti possano constatare che proviene dall'host. La comparsa di questo elenco segnala l'inizio del turno 1 della Sezione 11.1.2 (ad esempio, la fine del turno di impostazione '0'). I membri fittizi moltiplicheranno il loro scalare fornitogli dall'host per il segreto condiviso appropriato, e lo aggiungeranno alla loro maschera di pseudo impegno. In questo modo, anche l'output dell'host non può essere identificato da alcun partecipante nella transazione finale senza una coalizione completa contro di lui.

Per semplificare i calcoli della commissione, l'host può distribuire la commissione totale da utilizzare nella transazione alla fine del turno 1, poiché apprenderà presto il peso della transazione. I partecipanti possono verificare che l'importo sia simile all'importo previsto e pagarne la frazione.

Se i partecipanti collaborano per imbrogliare e non pagare la tariffa di hosting, l'host può terminare il TxTangle al turno 3. Può anche terminare se nel canale compaiono messaggi che non dovrebbero esserci o che non sono validi.

Alla fine del turno 5 l'host completa la transazione e la invia alla rete per la verifica, come parte del suo servizio. Include l'hash della transazione nel messaggio finale da distribuire.

[21] Se la mitigazione di Janus è implementata, questa difesa MITM dovrebbe invece essere fatta con la chiave base falsa di Janus. Ogni membro fittizio fornisce una chiave casuale $r_{mock}G$, dunque la chiave base effettiva è $\sum_{mock} \mathcal{H}_n(T_{agg}, \mathbb{S}_{mock}, r_{mock}G) * r_{mock}G$.

[22] Mentre un servizio TxTangle implementato può essere a scopo di lucro, il codice stesso potrebbe essere open source. Questo sarebbe importante per l'audit del software wallet che interagisce con un servizio TxTangle.

[23] È necessario usare le chiavi dei membri fittizi qui poiché l'host non paga una commissione, e il suo indice di output è sconosciuto.

11.3 Dealer Fidato

Ci sono alcuni svantaggi nel TxTangle totalmente decentralizzato. Richiede che tutti i partecipanti comunichino e collaborino attivamente all'interno di tempistiche rigorose (e si trovino a vicenda per iniziare), e ciò è difficile da implementare.

L'aggiunta di un *dealer* centrale, che è responsabile della raccolta delle informazioni sulla transazione da ciascun partecipante e dell'offuscamento dei raggruppamenti input/output, può rendere il procedimento più semplice. Il costo da pagare è una soglia di fiducia più alta, poiché il *dealer* deve (come minimo) conoscere quei raggruppamenti.[24]

11.3.1 Procedura Basata su Dealer

Il *dealer* pubblicizzerà la sua disponibilità a gestire i TxTangle e raccoglierà le richieste da potenziali partecipanti (costituite dal numero di input previsti [con i loro tipi] e output). Il *dealer* può partecipare con il proprio set di input/output se lo desidera.

Dopo che è stato assemblato un gruppo di massimo 16 output (dovrebbero esserci due o più partecipanti, e nessun partecipante può avere tutti gli output o input tranne uno), il *dealer* avvierà il primo dei cinque turni di comunicazione. In ogni turno il *dealer* raccoglie informazioni da ciascun partecipante, prende delle decisioni e invia messaggi che indicano l'inizio di un nuovo turno.

1. Per iniziare, il dealer genera, per ogni coppia di partecipanti, uno scalare casuale, e decide quale partecipante in ogni coppia dovrebbe avere la versione positiva o negativa di tale scalare. Utilizza il numero e il tipo di input e output per stimare la commissione totale della richiesta. Somma gli scalari di ciascun partecipante, e invia privatamente a ciascuno la loro somma, insieme alla frazione della commissione che si presume paghino, e gli indici (scelti casualmente) dei loro output. Questi messaggi costituiscono un segnale ai partecipanti che un TxTangle sta iniziando.

2. Ogni partecipante costruisce la propria sotto-transazione come farebbe normalmente, generando chiavi pubbliche di transazione separate per i propri output (con mitigazione di Janus se necessario), calcolando indirizzi di output monouso e codificando gli importi di output, creando pseudo impegni di output che bilanciano gli impegni di output e la frazione della commissione, assemblare un elenco di offset dei membri dell'anello da utilizzare nelle firme MLSAG insieme alle immagini chiave appropriate, e aggiungere a uno dei propri pseudo impegni di output lo scalare inviato dal dealer (moltiplicato per G). In seguito, i partecipanti creano prove parziali della Parte A per i propri output, e inviano tutte queste informazioni al dealer. Il dealer verifica il bilanciamento degli importi di input e output, e invia l'elenco completo delle prove parziali della Parte A a ciascun partecipante.

[24] Questa sezione è ispirata al protocollo MoJoin.

3. Ogni partecipante calcola la sfida aggregata A, e genera prove parziali della Parte B che invia al dealer. Il dealer raccoglie le prove parziali e le distribuisce a tutti gli altri partecipanti.

4. Ogni partecipante calcola la sfida aggregata B, e genera prove parziali della Parte C che invia al dealer. Il dealer le raccoglie, e applica la tecnica del prodotto interno logaritmico per comprimerle nella prova finale. Supponendo che la prova sia verificata come dovrebbe, genera una chiave pubblica di transazione *base* falsa di Janus casuale, e invia il messaggio da firmare in MLSAG a ciascun partecipante.

5. Ogni partecipante completa i propri MLSAG e li invia al dealer. Una volta che ha tutto il necessario, il *dealer* può finire di costruire la transazione, e inviarla per essere inclusa nella blockchain. Può anche inviare l'ID della transazione a ciascun partecipante in modo che possano confermare che è stata pubblicata.

Bibliografia

[1] Add intervening v5 fork for increased min block size. https://github.com/monero-project/monero/pull/1869 [Online; accessed 05/24/2018].

[2] CoinJoin. https://en.bitcoin.it/wiki/CoinJoin [Online; accessed 01/26/2020].

[3] cryptography - What is a cryptographic oracle? https://security.stackexchange.com/questions/10617/what-is-a-cryptographic-oracle [Online; accessed 04/22/2018].

[4] Cryptography Tutorial. https://www.tutorialspoint.com/cryptography/index.htm [Online; accessed 05/19/2018].

[5] Cryptonote Address Tests. https://xmr.llcoins.net/addresstests.html [Online; accessed 04/19/2018].

[6] Directed acyclic graph. https://en.wikipedia.org/wiki/Directed_acyclic_graph [Online; accessed 05/27/2018].

[7] Extended Euclidean algorithm. https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm [Online; accessed 03/19/2020].

[8] hackerone #501585. https://hackerone.com/reports/501585 [Online; accessed 12/28/2019].

[9] How do payment ids work? https://monero.stackexchange.com/questions/1910/how-do-payment-ids-work [Online; accessed 04/21/2018].

[10] Modular arithmetic. https://en.wikipedia.org/wiki/Modular_arithmetic [Online; accessed 04/16/2018].

[11] Monero 0.13.0 "Beryllium Bullet" Release. https://www.getmonero.org/2018/10/11/monero-0.13.0-released.html [Online; accessed 10/12/2019].

[12] Monero 0.14.0 "Boron Butterfly" Release. https://web.getmonero.org/2019/02/25/monero-0.14.0-released.html [Online; accessed 10/12/2019].

[13] Monero v0.9.3 - Hydrogen Helix - released! https://www.reddit.com/r/Monero/comments/4bgw4z/monero_v093_hydrogen_helix_released_urgent_and/ [Online; accessed 05/24/2018].

[14] Randomx. https://www.monerooutreach.org/stories/RandomX.php [Online; accessed 10/12/2019].

[15] Ring CT; Moneropedia. https://getmonero.org/resources/moneropedia/ringCT.html [Online; accessed 06/05/2018].

[16] Tail emission. https://getmonero.org/resources/moneropedia/tail-emission.html [Online; accessed 05/24/2018].

[17] Trust the math? An Update. http://www.math.columbia.edu/~woit/wordpress/?p=6522 [Online; accessed 04/04/2018].

[18] Useful for learning about Monero coin emission. https://www.reddit.com/r/Monero/comments/512kwh/useful_for_learning_about_monero_coin_emission/d78tpgi/ [Online; accessed 05/25/2018].

[19] What is a premine? https://www.cryptocompare.com/coins/guides/what-is-a-premine/ [Online; accessed 06/11/2018].

[20] What is the block maturity value seen in many pool interfaces? https://monero.stackexchange.com/questions/2251/what-is-the-block-maturity-value-seen-in-many-pool-interfaces [Online; accessed 05/26/2018].

[21] XOR – from Wolfram Mathworld. http://mathworld.wolfram.com/XOR.html [Online; accessed 04/21/2018].

[22] Federal Information Processing Standards Publication (FIPS 197). Advanced Encryption Standard (AES), 2001. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf [Online; access 03/04/2020].

[23] The .xz File Format, August 2009. https://tukaani.org/xz/xz-file-format.txt section 1.2 [Online; accessed 04/02/2020].

[24] Analysis of Bitcoin Transaction Size Trends, October 2015. https://tradeblock.com/blog/analysis-of-bitcoin-transaction-size-trends [Online; accessed 01/17/2020].

[25] Base58Check encoding, November 2017. https://en.bitcoin.it/wiki/Base58Check_encoding [Online; accessed 02/20/2020].

[26] Monero cryptonight variants, and add one for v7, April 2018. https://github.com/monero-project/monero/pull/3253 [Online; accessed 05/23/2018].

[27] Payment Reversal Explained + 10 Ways to Avoid Them, October 2018. https://tidalcommerce.com/learn/payment-reversal [Online; accessed 02/11/2020].

[28] Diffie–Hellman problem, December 2019. https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_problem [Online; accessed 03/31/2020].

[29] Kurt M. Alonso and Jordi Herrera Joancomartí. Monero - Privacy in the Blockchain. Cryptology ePrint Archive, Report 2018/535, 2018. https://eprint.iacr.org/2018/535.

[30] Kurt M. Alonso and Koe. Zero to Monero - First Edition, June 2018. https://web.getmonero.org/library/Zero-to-Monero-1-0-0.pdf [Online; accessed 01/15/2020].

[31] Kristov Atlas. Kristov Atlas Security Advisory 20140609-0, June 2014. https://www.coinjoinsudoku.com/advisory/ [Online; accessed 01/26/2020].

[32] Adam Back. Ring signature efficiency. BitcoinTalk, 2015. https://bitcointalk.org/index.php?topic=972541.msg10619684#msg10619684 [Online; accessed 04/04/2018].

[33] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. https://cr.yp.to/ecdh/curve25519-20060209.pdf [Online; accessed 03/04/2020].

[34] Daniel J. Bernstein. ChaCha, a variant of Salsa20, January 2008. https://cr.yp.to/chacha/chacha-20080120.pdf [Online; accessed 03/04/2020].

[35] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. *Twisted Edwards Curves*, pages 389–405. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. https://eprint.iacr.org/2008/013.pdf [Online; accessed 02/13/2020].

[36] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, Sep 2012. https://ed25519.cr.yp.to/ed25519-20110705.pdf [Online; accessed 03/04/2020].

[37] Daniel J. Bernstein and Tanja Lange. *Faster Addition and Doubling on Elliptic Curves*, pages 29–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. https://eprint.iacr.org/2007/286.pdf [Online; accessed 03/04/2020].

[38] Karina Bjørnholdt. Dansk politi har knækket bitcoin-koden, May 2017. http://www.dansk-politi.dk/artikler/2017/maj/dansk-politi-har-knaekket-bitcoin-koden [Online; accessed 04/04/2018].

[39] Dan Boneh and Victor Shoup. A Graduate Course in Applied Cryptography. https://crypto.stanford.edu/~dabo/cryptobook/ [Online; accessed 12/30/2019].

[40] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short Proofs for Confidential Transactions and More. https://eprint.iacr.org/2017/1066.pdf [Online; accessed 10/28/2018].

[41] Francisco Cabañas. Francisco Cabanas - Critical Role of Min Block Reward Trail Emission - DEF CON 27 Monero Village, December 2019. https://www.youtube.com/watch?v=IlghysBBuyU [Online; accessed 01/15/2020].

[42] Francisco Cabañas. Lightning talk: An Overview of Monero's Adaptive Blockweight Approach to Scaling, December 2019. https://frab.riat.at/en/36C3/public/events/125.html [Online; accessed 01/12/2020].

[43] Francisco Cabañas. MoneroKon 2019 - Spam Mitigation and Size Control in Permissionless Blockchains, June 2019. https://www.youtube.com/watch?v=Hbm0ub3qWw4&list=LL2HXH-vq_sTPXMNP4fZNKRw&index=10&t=0s [Online; accessed 01/10/2020].

[44] Miles Carlsten, Harry Kalodner, Matthew Weinberg, and Arvind Narayanan. On the Instability of Bitcoin Without the Block Reward, 2016. http://randomwalker.info/publications/mining_CCS.pdf [Online; accessed 01/12/2020].

[45] Chainalysis. THE 2020 STATE OF CRYPTO CRIME, January 2020. https://go.chainalysis.com/rs/503-FAP-074/images/2020-Crypto-Crime-Report.pdf [Online; accessed 02/11/2020].

[46] David Chaum and Eugène Van Heyst. Group Signatures. In *Proceedings of the 10th Annual International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'91, pages 257–265, Berlin, Heidelberg, 1991. Springer-Verlag. https://chaum.com/publications/Group_Signatures.pdf [Online; accessed 03/04/2020].

[47] dalek cryptography. Bulletproofs. https://doc-internal.dalek.rs/bulletproofs/index.html [Online; accessed 03/02/2020].

[48] Michael Davidson and Tyler Diamond. On the Profitability of Selfish Mining Against Multiple Difficulty Adjustment Algorithms. Cryptology ePrint Archive, Report 2020/094, 2020. https://eprint.iacr.org/2020/094 [Online; accessed 03/26/2020].

[49] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 2006. https://ee.stanford.edu/~hellman/publications/24.pdf [Online; accessed 03/04/2020].

[50] Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. On the Security of Two-Round Multi-Signatures. Cryptology ePrint Archive, Report 2018/417, 2018. https://eprint.iacr.org/2018/417 [Online; accessed 02/07/2020].

[51] Justin Ehrenhofer. Justin Ehrenhofer - Improving Monero Release Schedule - DEF CON 27 Monero Village, December 2019. https://www.youtube.com/watch?v=MjNXmJUk2Jo timestamp 22:15 [Online; accessed 03/20/2020].

[52] Justin Ehrenhofer. Monero Adds Blockchain Pruning and Improves Transaction Efficiency, February 2019. https://web.getmonero.org/2019/02/01/pruning.html [Online; accessed 12/30/2019].

[53] Justin Ehrenhofer and knacc. Advisory note for users making use of subaddresses, October 2019. https://web.getmonero.org/2019/10/18/subaddress-janus.html [Online; accessed 01/02/2020].

[54] Serhack et al. Mastering Monero, December 2019. https://masteringmonero.com/ [Online; accessed 01/10/2020].

[55] Exantech. Methods of anonymous blockchain analysis: an overview, November 2019. https://medium.com/@exantech/methods-of-anonymous-blockchain-analysis-an-overview-d700e27ea98c [Online; accessed 01/26/2020].

[56] Ittay Eyal and Emin Gün Sirer. Majority is not Enough: Bitcoin Mining is Vulnerable. *CoRR*, abs/1311.0243, 2013. https://arxiv.org/pdf/1311.0243.pdf [Online; accessed 03/04/2020].

[57] Amos Fiat and Adi Shamir. How To Prove Yourself: Practical Solutions to Identification and Signature Problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 186–194, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg. https://link.springer.com/content/pdf/10.1007%2F3-540-47721-7_12.pdf [Online; accessed 03/04/2020].

[58] Ryo "fireice_uk" Cryptocurrency. On-chain tracking of Monero and other Cryptonotes, April 2019. https://medium.com/@crypto_ryo/on-chain-tracking-of-monero-and-other-cryptonotes-e0afc6752527 [Online; accessed 03/25/2020].

[59] Riccardo "fluffypony" Spagni. Monero 0.12.0.0 "Lithium Luna" Release, March 2018. https://web.getmonero.org/2018/03/29/monero-0.12.0.0-released.html [Online; accessed 02/18/2020].

[60] Riccardo "fluffypony" Spagni and luigi1111. Disclosure of a Major Bug in Cryptonote Based Currencies, May 2017. https://getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-cryptonote-based-currencies.html [Online; accessed 04/10/2018].

[61] David Friedman. A Positive Account of Property Rights, 1994. http://www.daviddfriedman.com/Academic/Property/Property.html [Online; accessed 03/18/2020].

[62] Eiichiro Fujisaki and Koutarou Suzuki. *Traceable Ring Signature*, pages 181–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. https://link.springer.com/content/pdf/10.1007%2F978-3-540-71677-8_13.pdf [Online; accessed 03/04/2020].

[63] Fungible, July 2014. https://wiki.mises.org/wiki/Fungible [Online; accessed 03/31/2020].

[64] glv2. Varint description; Issue #2340. https://github.com/monero-project/monero/issues/2340#issuecomment-324692291 [Online; accessed 06/14/2018].

[65] Brandon Goodell, Sarang Noether, and Arthur Blue. Concise linkable ring signatures and applications, MRL-0011, September 2019. https://web.getmonero.org/resources/research-lab/pubs/MRL-0011.pdf [Online; accessed 02/02/2020].

[66] Thomas C Hales. The NSA back door to NIST. *Notices of the AMS*, 61(2):190–192. https://www.ams.org/notices/201402/rnoti-p190.pdf [Online; accessed 03/04/2020].

[67] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[68] Howard "hyc" Chu. RandomX, Pull Request #5549, May 2019. https://github.com/monero-project/monero/pull/5549 [Online; accessed 03/03/2020].

[69] Aram Jivanyan. Lelantus: Towards Confidentiality and Anonymity of Blockchain Transactions from Standard Assumptions. Cryptology ePrint Archive, Report 2019/373, 2019. https://eprint.iacr.org/2019/373.pdf [Online; accessed 03/04/2020].

[70] Don Johnson and Alfred Menezes. The Elliptic Curve Digital Signature Algorithm (ECDSA). Technical Report CORR 99-34, Dept. of C&O, University of Waterloo, Canada, 1999. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.472.9475&rep=rep1&type=pdf [Online; accessed 04/04/2018].

[71] JollyMort. Monero Dynamic Block Size and Dynamic Minimum Fee, March 2017. https://github.com/JollyMort/monero-research/blob/master/Monero%20Dynamic%20Block%20Size%20and%20Dynamic%20Minimum%20Fee/Monero%20Dynamic%20Block%20Size%20and%20Dynamic%20Minimum%20Fee%20-%20DRAFT.md [Online; accessed 01/12/2020].

[72] S. Josefsson, SJD AB, and N. Moeller. EdDSA and Ed25519. Internet Research Task Force (IRTF), 2015. https://tools.ietf.org/html/draft-josefsson-eddsa-ed25519-03 [Online; accessed 05/11/2018].

[73] Simon Josefsson and Ilari Liusvaara. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, January 2017. https://rfc-editor.org/rfc/rfc8032.txt [Online; accessed 03/04/2020].

[74] kenshi84. Subaddresses, Pull Request #2056, May 2017. https://github.com/monero-project/monero/pull/2056 [Online; accessed 02/16/2020].

[75] Eike Kiltz, Daniel Masny, and Jiaxin Pan. Optimal Security Proofs for Signatures from Identification Schemes. In *Proceedings, Part II, of the 36th Annual International Cryptology Conference on Advances in Cryptology — CRYPTO 2016 - Volume 9815*, pages 33–61, Berlin, Heidelberg, 2016. Springer-Verlag. https://eprint.iacr.org/2016/191.pdf [Online; accessed 03/04/2020].

[76] Bradley Kjell. Big Endian and Little Endian. https://chortle.ccsu.edu/AssemblyTutorial/Chapter-15/ass15_3.html [Online; accessed 01/23/2020].

[77] Alexander Klimov. ECC Patents?, October 2005. http://article.gmane.org/gmane.comp.encryption.general/7522 [Online; accessed 04/04/2018].

[78] koe and jtgrassie. Historical significance of FEE_PER_KB_OLD, December 2019. https://monero.stackexchange.com/questions/11864/historical-significance-of-fee-per-kb-old [Online; accessed 01/02/2020].

[79] koe and jtgrassie. Complete extra field structure (standard interpretation), January 2020. https://monero.stackexchange.com/questions/11888/complete-extra-field-structure-standard-interpretation [Online; accessed 01/05/2020].

[80] Mitchell Krawiec-Thayer. MoneroKon 2019 - Visualizing Monero: A Figure is Worth a Thousand Logs, June 2019. https://www.youtube.com/watch?v=XIrqyxU3k5Q [Online; accessed 01/06/2020].

[81] Mitchell Krawiec-Thayer. Numerical simulation for upper bound on dynamic blocksize expansion, January 2019. https://github.com/noncesense-research-lab/Blockchain_big_bang/blob/master/models/Isthmus_Bx_big_bang_model.ipynb [Online; accessed 01/08/2020].

[82] Russell W. F. Lai, Viktoria Ronge, Tim Ruffing, Dominique Schröder, Sri Thyagarajan, and Jiafan Wang. Omniring: Scaling Private Payments Without Trusted Setup. pages 31–48, 11 2019. https://eprint.iacr.org/2019/580.pdf [Online; accessed 03/04/2020].

[83] Joseph K. Liu, Victor K. Wei, and Duncan S. Wong. *Linkable Spontaneous Anonymous Group Signature for Ad Hoc Groups*, pages 325–335. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. https://eprint.iacr.org/2004/027.pdf [Online; accessed 03/04/2020].

[84] Ueli Maurer. Unifying Zero-Knowledge Proofs of Knowledge. In Bart Preneel, editor, *Progress in Cryptology – AFRICACRYPT 2009*, pages 272–286, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. https://www.crypto.ethz.ch/publications/files/Maurer09.pdf [Online; accessed 03/04/2020].

[85] Greg Maxwell. Confidential Transactions. https://people.xiph.org/~greg/confidential_values.txt [Online; accessed 04/04/2018].

[86] Gregory Maxwell and Andrew Poelstra. Borromean Ring Signatures. 2015. https://pdfs.semanticscholar.org/4160/470c7f6cf05ffc81a98e8fd67fb0c84836ea.pdf [Online; accessed 04/04/2018].

[87] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple Schnorr Multi-Signatures with Applications to Bitcoin. May 2018. https://eprint.iacr.org/2018/068.pdf [Online; accessed 03/01/2020].

[88] Sarah Meiklejohn and Claudio Orlandi. Privacy-Enhancing Overlays in Bitcoin. https://fc15.ifca.ai/preproceedings/bitcoin/paper_5.pdf [Online; accessed 01/26/2020].

[89] R. C. Merkle. Protocols for Public Key Cryptosystems. In *1980 IEEE Symposium on Security and Privacy*, pages 122–122, April 1980. http://www.merkle.com/papers/Protocols.pdf [Online; accessed 03/04/2020].

[90] Andrew Miller, Malte Möser, Kevin Lee, and Arvind Narayanan. An Empirical Analysis of Linkability in the Monero Blockchain. *CoRR*, abs/1704.04299, 2017. https://arxiv.org/pdf/1704.04299.pdf [Online; accessed 03/04/2020.

[91] Victor S Miller. Use of Elliptic Curves in Cryptography. In *Lecture Notes in Computer Sciences; 218 on Advances in cryptology—CRYPTO 85*, pages 417–426, Berlin, Heidelberg, 1986. Springer-Verlag. https://link.springer.com/content/pdf/10.1007/3-540-39799-X_31.pdf [Online; accessed 03/04/2020].

[92] Nicola Minichiello. The Bitcoin Big Bang: Tracking Tainted Bitcoins, June 2015. https://bravenewcoin.com/insights/the-bitcoin-big-bang-tracking-tainted-bitcoins [Online; accessed 03/31/2020].

[93] Ludwig Von Mises. Human Action: A Treatise on Economics; The Scholar's Edition, 1949. https://cdn.mises.org/Human%20Action_3.pdf [Online; accessed 03/05/2020].

[94] Monero inception and history. https://monero.stackexchange.com/questions/475/monero-inception-and-history-how-did-monero-get-started-what-are-its-origins-a/476#476 [Online; accessed 05/23/2018].

[95] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008. http://bitcoin.org/bitcoin.pdf [Online; accessed 03/04/2020].

[96] Arvind Narayanan. Bitcoin is unstable without the block reward, October 2016. https://freedom-to-tinker.com/2016/10/21/bitcoin-is-unstable-without-the-block-reward/ [Online; accessed 01/12/2020].

[97] Arvind Narayanan and Malte Möser. Obfuscation in Bitcoin: Techniques and Politics. *CoRR*, abs/1706.05432, 2017. https://arxiv.org/ftp/arxiv/papers/1706/1706.05432.pdf [Online; accessed 03/04/2020].

[98] Y. Nir, Check Point, A. Langley, and Google Inc. ChaCha20 and Poly1305 for IETF Protocols. Internet Research Task Force (IRTF), May 2015. https://tools.ietf.org/html/rfc7539 [Online; accessed 05/11/2018].

[99] NIST Releases SHA-3 Cryptographic Hash Standard, August 2015. https://www.nist.gov/news-events/news/2015/08/nist-releases-sha-3-cryptographic-hash-standard [Online; accessed 06/02/2018].

[100] Sarang Noether. Janus mitigation, Issue #62, January 2020. https://github.com/monero-project/research-lab/issues/62 [Online; accessed 02/17/2020].

[101] Sarang Noether. Multisignature implementation, Issue #67, January 2020. https://github.com/monero-project/research-lab/issues/67 [Online; accessed 02/16/2020].

[102] Sarang Noether. Transaction proofs (InProofV1 and OutProofV1) have incomplete Schnorr challenges, Issue #60, January 2020. https://github.com/monero-project/research-lab/issues/60 [Online; accessed 02/20/2020].

[103] Sarang Noether. WIP: Updated transaction proofs and tests, Pull Request #6329, February 2020. https://github.com/monero-project/monero/pull/6329 [Online; accessed 02/20/2020].

[104] Sarang Noether and Brandon Goodell. An efficient implementation of Monero subaddresses, MRL-0006, October 2017. https://web.getmonero.org/resources/research-lab/pubs/MRL-0006.pdf [Online; accessed 04/04/2018].

[105] Sarang Noether and Brandon Goodell. Triptych: logarithmic-sized linkable ring signatures with applications. Cryptology ePrint Archive, Report 2020/018, 2020. https://eprint.iacr.org/2020/018.pdf [Online; accessed 03/04/2020.

[106] Shen Noether. Understanding ge_fromfe_frombytes_vartime. https://web.getmonero.org/resources/research-lab/pubs/ge_fromfe.pdf [Online; accessed 01/20/2020].

[107] Shen Noether, Adam Mackenzie, and Monero Core Team. Ring Confidential Transactions, MRL-0005, February 2016. https://web.getmonero.org/resources/research-lab/pubs/MRL-0005.pdf [Online; accessed 06/15/2018].

[108] Shen Noether, Adam Mackenzie, and Monero Core Team. Ring Multisignature, April 2016. https://web.archive.org/web/20161023010706/https://shnoe.files.wordpress.com/2016/03/mrl-0008_april28.pdf [Online; accessed via WayBack Machine 01/21/2020].

[109] Shen Noether and Sarang Noether. Monero is Not That Mysterious, MRL-0003, September 2014. https://web.getmonero.org/resources/research-lab/pubs/MRL-0003.pdf [Online; accessed 06/15/2018].

[110] Shen Noether and Sarang Noether. Thring Signatures and their Applications to Spender-Ambiguous Digital Currencies, MRL-0009, November 2018. https://web.getmonero.org/resources/research-lab/pubs/MRL-0009.pdf [Online; accessed 01/15/2020].

[111] Michael Padilla. Beating Bitcoin bad guys, August 2016. http://www.sandia.gov/news/publications/labnews/articles/2016/19-08/bitcoin.html [Online; accessed 04/04/2018].

[112] Torben Pryds Pedersen. *Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing*, pages 129–140. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992. https://www.cs.cornell.edu/courses/cs754/2001fa/129.PDF [Online; accessed 03/04/2020].

[113] rbrunner7. 2/2 Multisig in CLI Wallet, January 2018. https://taiga.getmonero.org/project/rbrunner7-really-simple-multisig-transactions/wiki/22-multisig-in-cli-wallet [Online; accessed 01/21/2020].

[114] René "rbrunner7" Brunner. Multisig transactions with MMS and CLI wallet. https://web.getmonero.org/resources/user-guides/multisig-messaging-system.html [Online; accessed 01/21/2020].

[115] René "rbrunner7" Brunner. Project Rationale From the Initiator, January 2018. https://taiga.getmonero.org/project/rbrunner7-really-simple-multisig-transactions/wiki/home [Online; accessed 01/21/2020].

[116] René "rbrunner7" Brunner. Basic Monero support ready for assessment, Issue #1638, June 2019. https://github.com/OpenBazaar/openbazaar-go/issues/1638 [Online; accessed 02/11/2020].

[117] Jamie Redman. Industry Execs Claim Freshly Minted 'Virgin Bitcoins' Fetch 20% Premium, March 2020. https://news.bitcoin.com/industry-execs-freshly-minted-virgin-bitcoins/ [Online; accessed 03/31/2020].

[118] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to Leak a Secret. C. Boyd (Ed.): ASIACRYPT 2001, LNCS 2248, pp. 552-565, 2001. https://people.csail.mit.edu/rivest/pubs/RST01.pdf [Online; accessed 04/04/2018].

[119] SafeCurves. SafeCurves: choosing safe curves for elliptic-curve cryptography, 2013. https://safecurves.cr.yp.to/rigid.html [Online; accessed 03/25/2020].

[120] C. P. Schnorr. Efficient Identification and Signatures for Smart Cards. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 239–252, New York, NY, 1990. Springer New York. https://link.springer.com/content/pdf/10.1007%2F0-387-34805-0_22.pdf [Online; accessed 03/04/2020].

[121] Paola Scozzafava. Uniform distribution and sum modulo m of independent random variables. *Statistics & Probability Letters*, 18(4):313 – 314, 1993. https://sci-hub.tw/https://www.sciencedirect.com/science/article/abs/pii/016771529390021A [Online; accessed 03/04/2020].

[122] Bassam El Khoury Seguias. Monero Building Blocks, 2018. https://delfr.com/category/monero/ [Online; accessed 10/28/2018].

[123] Seigen, Max Jameson, Tuomo Nieminen, Neocortex, and Antonio M. Juarez. CryptoNight Hash Function. CryptoNote, March 2013. https://cryptonote.org/cns/cns008.txt [Online; accessed 04/04/2018].

[124] QingChun ShenTu and Jianping Yu. Research on Anonymization and De-anonymization in the Bitcoin System. *CoRR*, abs/1510.07782, 2015. https://arxiv.org/ftp/arxiv/papers/1510/1510.07782.pdf [Online; accessed 03/04/2020].

[125] thankful_for_today. [ANN][BMR] Bitmonero - a new coin based on CryptoNote technology - LAUNCHED, April 2014. Monero's actual launch date was April 18$^{\text{th}}$, 2014. https://bitcointalk.org/index.php?topic=563821.0 [Online; accessed 05/24/2018].

[126] Manny Trillo. Visa Transactions Hit Peak on Dec. 23, January 2011. https://www.visa.com/blogarchives/us/2011/01/12/visa-transactions-hit-peak-on-dec-23/index.html [Online; accessed 01/10/2020].

[127] Alicia Tuovila. Audit, July 2019. https://www.investopedia.com/terms/a/audit.asp [Online; accessed 02/24/2020].

[128] UkoeHB. Proof an output has not been spent, Issue #68, January 2020. https://github.com/monero-project/research-lab/issues/68 [Online; accessed 02/19/2020].

[129] UkoeHB. Reduce minimum fee variability, Issue #70, February 2020. https://github.com/monero-project/research-lab/issues/70 [Online; accessed 03/20/2020].

[130] UkoeHB. Treat pre-RingCT outputs like coinbase outputs, Issue #59, January 2020. https://github.com/monero-project/research-lab/issues/59 [Online; accessed 03/22/2020].

[131] Maciej Ulas. Rational points on certain hyperelliptic curves over finite fields, 2007. https://arxiv.org/pdf/0706.1448.pdf [Online; accessed 03/03/2020].

[132] user36100 and cooldude45. Which entities are related to Bytecoin and Minergate?, December 2016. https://monero.stackexchange.com/questions/2930/which-entities-are-related-to-bytecoin-and-minergate [Online; accessed 01/17/2020].

[133] user36303 and koe. Duplicate ring members, January 2020. https://monero.stackexchange.com/questions/11882/duplicate-ring-members [Online; accessed 01/18/2020].

[134] Nicolas van Saberhagen. CryptoNote V2.0. https://cryptonote.org/whitepaper.pdf [Online; accessed 04/04/2018].

[135] David Wagner. A Generalized Birthday Problem. In Moti Yung, editor, *Advances in Cryptology — CRYPTO 2002*, pages 288–304, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. https://link.springer.com/content/pdf/10.1007%2F3-540-45708-9_19.pdf [Online; accessed 02/07/2020].

[136] Adam "waxwing" Gibson. From Zero (Knowledge) To Bulletproofs, March 2018. https://github.com/AdamISZ/from0k2bp/blob/master/from0k2bp.pdf [Online; accessed 03/01/2020].

[137] Adam "waxwing" Gibson. Avoiding Wagnerian Tragedies, December 2019. https://joinmarket.me/blog/blog/avoiding-wagnerian-tragedies/ [Online; accessed 03/01/2020].

[138] Albert Werner, Montag, Prometheus, and Tereno. CryptoNote Transaction Extra Field. CryptoNote, October 2012. https://cryptonote.org/cns/cns005.txt [Online; accessed 04/04/2018].

[139] Ursula Whitcher. The Birthday Problem. http://mathforum.org/dr.math/faq/faq.birthdayprob.html [Online; accessed 02/07/2020].

[140] Wikibooks. Cryptography/Prime Curve/Standard Projective Coordinates, March 2011. https://en.wikibooks.org/wiki/Cryptography/Prime_Curve/Standard_Projective_Coordinates [Online; accessed 03/03/2020].

[141] Tsz Hon Yuen, Shi-feng Sun, Joseph K. Liu, Man Ho Au, Muhammed F. Esgin, Qingzhao Zhang, and Dawu Gu. RingCT 3.0 for Blockchain Confidential Transaction: Shorter Size and Stronger Security. Cryptology ePrint Archive, Report 2019/508, 2019. https://eprint.iacr.org/2019/508.pdf [Online; accessed 03/04/2020].

[142] zawy12. Summary of Difficulty Algorithms, Issue #50, December 2019. https://github.com/zawy12/difficulty-algorithms/issues/50 [Online; accessed 02/11/2020].

Appendices

APPENDICE A

Struttura delle Transazioni RCTTypeBulletproof2

In questa appendice presentiamo la struttura, anche detto *dump*, di una transazione Monero reale di tipo RCTTypeBulletproof2, insieme a note esplicative per i campi rilevanti.

Questo dump è stato ottenuto dal block explorer https://xmrchain.net, ma può essere reperito anche eseguendo il comando print_tx <TransactionID> +hex +json attraverso l'eseguibile monerod avviato in modalità non-detached. <TransactionID> è l'hash della transazione (Sezione 7.4.1). La prima linea del dump indica il comando eseguito.

Per replicare gli stessi risultati, i lettori possono seguire i seguenti passaggi:

1. È necessario predisporre un ambiente dotato dello strumento Linea di Comando di Monero (command line tool CLI), che può essere scaricato dal sito ufficiale https://web.getmonero.org/downloads/ (oltre che da altri posti). È sufficiente scaricare la CLI per il proprio sistema operativo, spostare il file in una posizione opportuna nel file system ed infine estrarre l'archivio zip.

2. Aprire dunque un terminale/prompt dei comandi e navigare nella cartella creata in seguito all'estrazione.

3. Eseguire monerod tramite il comando ./monerod. In seguito, si avvierà il processo di sincronizzazione che scaricherà una copia della blockchain in locale. Purtroppo, non c'è un altro modo semplice per visualizzare le transazioni nel proprio ambiente (ad esempio senza usare un servizio blockchain explorer esterno) senza scaricare la blockchain.

4. Una volta che il processo di sincronizzazione si completa, sarà possibile invocare comandi come `print_tx` dal terminale. È disponibile il comando `help` per una panoramica sui comandi disponibili.

Il campo `rctsig_prunable`, come indica il nome, si può *potare* (rimuovere) dalla propria copia della blockchain. Ciò significa che una volta che è stato raggiunto il consenso per il blocco e l'attuale lunghezza della catena esclude tutte le possibilità di attacchi a doppia spesa, l'intero campo può essere potato e sostituito con il suo hash, in modo da poter essere utilizzato all'interno dell'albero di Merkle.

Le immagini chiave (key image) vengono archiviate separatamente, nell'area non potabile delle transazioni. Sono essenziali per rilevare gli attacchi di doppia spesa e non possono essere rimosse.

La nostra transazione di esempio ha 2 input e 2 output ed è stata aggiunta alla blockchain al timestamp 2020-03-02 19:01:10 UTC (come riportato dal miner del blocco).

```
1  print_tx 84799c2fc4c18188102041a74cef79486181df96478b717e8703512c7f7f3349
2  Found in blockchain at height 2045821
3  {
4    "version": 2,
5    "unlock_time": 0,
6    "vin": [ {
7        "key": {
8          "amount": 0,
9          "key_offsets": [ 14401866, 142824, 615514, 18703, 5949, 22840, 5572, 16439,
10         983, 4050, 320
11         ],
12         "k_image": "c439b9f0da76ca0bb17920ca1f1f3f1d216090751752b091bef9006918cb3db4"
13       }
14     }, {
15       "key": {
16         "amount": 0,
17         "key_offsets": [ 14515357, 640505, 8794, 1246, 20300, 18577, 17108, 9824, 581,
18         637, 1023
19         ],
20         "k_image": "03750c4b23e5be486e62608443151fa63992236910c41fa0c4a0a938bc6f5a37"
21       }
22     }
23   ],
24   "vout": [ {
25       "amount": 0,
26       "target": {
```

```
27          "key": "d890ba9ebfa1b44d0bd945126ad29a29d8975e7247189e5076c19fa7e3a8cb00"
28        }
29      }, {
30        "amount": 0,
31        "target": {
32          "key": "dbec330f8a67124860a9bfb86b66db18854986bd540e710365ad6079c8a1c7b0"
33        }
34      }
35    ],
36    "extra": [ 1, 3, 39, 58, 185, 169, 82, 229, 226, 22, 101, 230, 254, 20, 143,
37    37, 139, 28, 114, 77, 160, 229, 250, 107, 73, 105, 64, 208, 154, 182, 158, 200,
38    73, 2, 9, 1, 12, 76, 161, 40, 250, 50, 135, 231
39    ],
40    "rct_signatures": {
41      "type": 4,
42      "txnFee": 32460000,
43      "ecdhInfo": [ {
44          "amount": "171f967524e29632"
45        }, {
46          "amount": "5c2a1a9f54ccf40b"
47        }],
48      "outPk": [ "fed8aded6914f789b63c37f9d2eb5ee77149e1aa4700a482aea53f82177b3b41",
49      "670e086e40511a279e0e4be89c9417b4767251c5a68b4fc3deb80fdae7269c17"]
50    },
51    "rctsig_prunable": {
52      "nbp": 1,
53      "bp": [ {
54          "A": "98e5f23484e97bb5b2d453505db79caadf20dc2b69dd3f2b3dbf2a53ca280216",
55          "S": "b791d4bc6a4d71de5a79673ed4a5487a184122321ede0b7341bc3fdc0915a796",
56          "T1": "5d58cfa9b69ecdb2375647729e34e24ce5eb996b5275aa93f9871259f3a1aecd",
57          "T2": "1101994fea209b71a2aa25586e429c4c0f440067e2b197469aa1a9a1512f84b7",
58          "taux": "b0ad39da006404ccacee7f6d4658cf17e0f42419c284bdca03c0250303706c03",
59          "mu": "cacd7ca5404afa28e7c39918d9f80b7fe5e572a92a10696186d029b4923fa200",
60          "L": [ "d06404fc35a60c6c47a04e2e43435cb030267134847f7a49831a61f82307fc32",
61          "c9a5932468839ee0cda1aa2815f156746d4dce79dab3013f4c9946fce6b69eff",
62          "efdae043dcedb79512581480d80871c51e063fe9b7a5451829f7a7824bcc5a0b",
63          "56fd2e74ac6e1766cfd56c8303a90c68165a6b0855fae1d5b51a2e035f333a1d",
64          "81736ed768f57e7f8d440b4b18228d348dce1eca68f969e75fab458f44174c99",
65          "695711950e076f54cf24ad4408d309c1873d0f4bf40c449ef28d577ba74dd86d",
66          "4dc3147619a6c9401fec004652df290800069b776fe31b3c5cf98f64eb13ef2c"
67          ],
68          "R": [ "7650b8da45c705496c26136b4c1104a8da601ea761df8bba07f1249495d8f1ce",
```

```
69          "e87789fbe99a44554871fcf811723ee350cba40276ad5f1696a62d91a4363fa6",
70          "ebf663fe9bb580f0154d52ce2a6dae544e7f6fb2d3808531b0b0749f5152ddbf",
71          "5a4152682a1e812b196a265a6ba02e3647a6bd456b7987adff288c5b0b556ec6",
72          "dc0dcb2e696e11e4b62c20b6bfcb6182290748c5de254d64bf7f9e3c38fb46c9",
73          "101e2271ced03b229b88228d74b36088b40c88f26db8b1f9935b85fb3ab96043",
74          "b14aae1d35c9b176ac526c23f31b044559da75cf95bc640d1005bfcc6367040b"
75          ],
76          "a": "4809857de0bd6becdb64b85e9dfbf6085743a8496006b72ceb81e01080965003",
77          "b": "791d8dc3a4ddde5ba2416546127eb194918839ced3dea7399f9c36a17f36150e",
78          "t": "aace86a7a1cbdec3691859fa07fdc83eed9ca84b8a064ca3f0149e7d6b721c03"
79        }
80      ],
81      "MGs": [ {
82          "ss": [[ "d7a9b87cfa74ad5322eedd1bff4c4dca08bcff6f8578a29a8bc4ad6789dee106",
83          "f08e5dfade29d2e60e981cb561d749ea96ddc7e6855f76f9b807842d1a17fe00"],
84          ["de0a86d12be2426f605a5183446e3323275fe744f52fb439041ad2d59136ea0b",
85          "0028f97976630406e12c54094cbbe23a23fe5098f43bcae37339bfc0c4c74c07"],
86          ["f6eef1f99e605372cb1ec2b3dd4c6e56a550fec071c8b1d830b9fda34de5cc05",
87          "cd98fc987374a0ac993edf4c9af0a6f2d5b054f2af601b612ea118f405303306"],
88          ["5a8437575dae7e2183a1c620efbce655f3d6dc31e64c96276f04976243461e08",
89          "5090103f7f73a33024fbda999cd841b99b87c45fa32c4097cdc222fa3d7e9502"],
90          ["88d34246afbccbd24d2af2ba29d835813634e619912ea4ca194a32281ac14e0e",
91          "eacdf59478f132dd8dbb9580546f96de194092558ffceeff410ee9eb30ce570e"],
92          ["571dab8557921bbae30bda9b7e613c8a0cff378d1ec6413f59e4972f30f2470d",
93          "5ca78da9a129619299304d9b03186233370023debfdaddcd49c1a338c1f0c50d"],
94          ["ac8dbe6bb28839cf98f02908bd1451742a10c713fdd51319f2d42a58bf1d7507",
95          "7347bf16cba5ee6a6f2d4f6a59d1ed0c1a43060c3a235531e7f1a75cd8c8530d"],
96          ["b8876bd3a5766150f0fbc675ba9c774c2851c04afc4de0b17d3ac4b6de617402",
97          "e39f1d2452d76521cbf02b85a6b626eeb5994f6f28ce5cf81adc0ff2b8adb907"],
98          ["1309f8ead30b7be8d0c5932743b343ef6c0001cef3a4101eae98ffde53f46300",
99          "370693fa86838984e9a7232bca42fd3d6c0c2119d44471d61eee5233ba53c20f"],
100         ["80bc2da5fc5951f2c7406fce37a7aa72ffef9cfa21595b1b68dfab4b7b9f9f0c",
101         "c37137898234f00bce00746b131790f3223f97960eefe67231eb001092f5510c"],
102         ["01c89e07571fd365cac6744b34f1b44e06c1c31cbf3ee4156d08309345fdb20e",
103         "a35c8786695a86c0a4e677b102197a11dadc7171dd8c2e1de90d828f050ec00f"]],
104         "cc": "0d8b70c600c67714f3e9a0480f1ffc7477023c793752c1152d5df0813f75ff0f"
105       }, {
106         "ss": [[ "4536e585af58688b69d932ef3436947a13d2908755d1c644ca9d6a978f0f0206",
107         "9aab6509f4650482529219a805ee09cd96bb439ee1766ced5d3877bf1518370b"],
108         ["5849d6bf0f850fcee7acbef74bd7f02f77ecfaaa16a872f52479ebd27339760f",
109         "96a9ec61486b04201313ac8687eaf281af59af9fd10cf450cb26e9dc8f1ce804"],
110         ["7fe5dcc4d3eff02fca4fb4fa0a7299d212cd8cd43ec922d536f21f92c8f93f00",
```

```
111         "d306a62831b49700ae9daad44fcd00c541b959da32c4049d5bdd49be28d96701"],
112         ["2edb125a5670d30f6820c01b04b93dd8ff11f4d82d78e2379fe29d7a68d9c103",
113         "753ac25628c0dada7779c6f3f13980dfc5b7518fb5855fd0e7274e3075a3410c"],
114         ["264de632d9cb867e052f95007dfdf5a199975136c907f1d6ad73061938f49c01",
115         "dd7eb6028d0695411f647058f75c42c67660f10e265c83d024c4199bed073d01"],
116         ["b2ac07539336954f2e9b9cba298d4e1faa98e13e7039f7ae4234ac801641340f",
117         "69e130422516b82b456927b64fe02732a3f12b5ee00c7786fe2a381325bf3004"],
118         ["49ea699ca8cf2656d69020492cdfa69815fb69145e8f922bb32e358c23cebb0f",
119         "c5706f903c04c7bed9c74844f8e24521b01bc07b8dbf597621cceeeb3afc1d0c"],
120         ["a1faf85aa942ba30b9f0511141fcab3218c00953d046680d36e09c35c04be905",
121         "7b6b1b6fb23e0ee5ea43c2498ea60f4fcf62f70c7e0e905eb4d9afa1d0a18800"],
122         ["785d0993a70f1c2f0ac33c1f7632d64e34dd730d1d8a2fb0606f5770ed633506",
123         "e12777c49ffc3f6c35d27a9ccb3d9b8fed7f0864a880f7bae7399e334207280e"],
124         ["ab31972bf1d2f904d6b0bf18f4664fa2b16a1fb2644cd4e6278b63ade87b6d09",
125         "1efb04fe9a75c01a0fe291d0ae00c716e18c64199c1716a086dd6e32f63e0a07"],
126         ["a6f4e21a27bf8d28fc81c873f63f8d78e017666adbf038da0b83c2ad04ef6805",
127         "c02103455f93c2d7ec4b7152db7de00d1c9e806b1945426b6773026b4a85dd03"]],
128         "cc": "d5ac037bb78db41cf924af713b7379c39a4e13901d3eac017238550a1a3b910a"
129       }],
130     "pseudoOuts": [ "b313c1ae9ca06213684fbdefa9412f4966ad192bc0b2f74ed1731381adb7ab58",
131     "7148e7ef5cfd156c62a6e285e5712f8ef123575499ff9a11f838289870522423"]
132   }
133 }
```


Campi di una Transazione

- (linea 2) - il comando `print_tx` riporta l'altezza del blocco in cui è stata trovata la transazione, riportato qui a scopo dimostrativo.
- `version` (linea 4) - Formato/era della versione della transazione; '2' corrisponde a RingCT.
- `unlock_time` (linea 5) - Impedisce che gli output di una transazione vengano spesi fino al termine del tempo specificato. Può essere un'altezza di blocco o un timestamp UNIX, se il numero è maggiore dell'inizio del tempo UNIX. Il valore predefinito è zero quando non viene specificato alcun limite.
- `vin` (linea 6-23) - Lista di input (composta da due input in questo caso).
- `amount` (linea 8) - Campo deprecato (legacy), era dedicato all'importo della transazione (per transazioni di tipo 1).
- `key_offset` (linea 9) - Questo campo consente ai verificatori di trovare le chavi appartenenti all'anello e i commitment nella blockchain, al fine di constatare la legittimità di questi mem-

bri. Il primo offset è un indice assoluto rispetto alla cronologia della blockchain, mentre i successi sono indici relativi al primo offset. Ad esempio, con offset reali (assoluti) {7,11,15,20}, su blockchain verranno registrati come {7,4,4,5}. I verificatori possono calcolare l'ultimo offset sommando tutti gli indici (7+4+4+5 = 20) (Sezione 6.2.4).

- `k_image` (linea 12) - Immagine chiave $\tilde{K}_j$ trattata nella Sezione 3.5, dove $j = 1$ dato che è il primo input.
- `vout` (linee 24-35) - Lista di output (composta da due output in questo caso).
- `amount` (linea 25) - Campo deprecato dedicato all'importo delle transazioni di tipo 1.
- `key` (linea 27) - Chiave dell'indirizzo one-time di destinazione per l'output $t = 0$ come descritto nella Sezione 4.2
- `extra` (linee 36-39) - Dati aggiuntivi, inclusa la *chiave pubblica della transazione*, ovvero il segreto condiviso rG della Sezione 4.2, e il payment ID cifrato della Sezione 4.4. Tipicamente funziona come segue: Ogni numero rappresenta un byte (può assumere un valore compreso tra 0 e 255), e ogni tipo di elemento che può comparire nel campo ha un proprio *tag* (etichetta) e una *length* (lunghezza). Il *tag* indica quale tipo di informazione segue, mentre *length* specifica quanti byte occupa quell'informazione. Il primo numero è sempre un tag. In questo caso, '1' indica una 'chiave pubblica della transazione' (transaction public key). Le chiavi pubbliche delle transazioni sono sempre lunghe 32 byte, quindi non è necessario specificarne la lunghezza. Trenta-due byte dopo troviamo un nuovo tag '2', che indica una 'extra nonce', la cui lunghezza è '9'; il byte successivo è '1', che segnala un payment ID cifrato a 8 byte (l'extra nonce può contenere campi al suo interno, per qualche ragione). Seguono otto byte, e questo segna la fine del campo extra. Per ulteriori dettagli, si veda [79]. (Nota: nella specifica originale di Cryptonote, il primo byte indicava la dimensione del campo. Monero non utilizza questa convenzione). [138]

  src/cryptonote_basic/tx_extra.h
- `rct_signatures` (linee 40-50) - Prima parte della firma.
- `type` (linea 41) - Tipologia di firma; `RCTTypeBulletproof2` tipo 4. I tipi RingCT `RCTTypeFull` e `RCTTypeSimple` sono deprecati, ovvero tipo 1 e 2 rispettivamente. Le transazioni di mining usano `RCTTypeNull`, ovvero il tipo 0.
- `txnFee` (linea 42) - Commissioni della transazione in chiaro, in queso caso 0.00003246 XMR.
- `ecdhInfo` (linee 43-47) - 'Informazione sulla curva ellittica Diffie-Hellman': Importi offuscati per ogni output $t \in \{0, ..., p-1\}$; in questo caso $p = 2$.
- `amount` (linea 44) - Campo *ammontare* dell'output $t = 0$ come descritto nella Sezione 5.3.
- `outPk` (linee 48-49) - Commitment per ogni output, Sezione 5.4.
- `rctsig_prunable` (linee 51-132) - Seconda parte della firma.
- `nbp` (linea 52) - Numero di prove di intervallo (range proof) Bulletproof in questa transazione.

- `bp` (linee 53-80) - Prove Bulletproof (Si ricorda che i Bulletproof non sono stati trattati in questo documento)

$$\Pi_{BP} = (A, S, T_1, T_2, \tau_x, \mu, \mathbb{L}, \mathbb{R}, a, b, t)$$

- `MGs` (linee 81-129) - Firme MLSAG.
- `ss` (linee 82-103) - Componenti $r_{i,1}$ e $r_{i,2}$ derivanti dalla firma MLSAG del primo output

$$\sigma_j(\mathfrak{m}) = (c_1, r_{1,1}, r_{1,2}, ..., r_{v+1,1}, r_{v+1,2})$$

- `cc` (linea 104) - Componente c_1 della sopracitata firma MLSAG.
- `pseudoOuts` (linee 130-131) - Commitment pseudo-output C'^a_j, come descritto nella Sezione 5.3. Si ricorda che la somma di questi commitment è pari alla somma dei commitment dei due output di questa transazione (più i commitment delle commissioni della transazione fH).

APPENDICE B

Contenuto dei Blocchi

In questo appendice viene illustrata la struttura di un blocco della catena, in particolare il 1582196$^{\text{esimo}}$ dopo il blocco genesi. Il blocco in questione ha 5 transazioni che sono state registrate sulla blockchain al timestamp 2018-05-27 21:56:01 UTC (come riportato dal miner del blocco).

```
print_block 1582196
timestamp: 1527458161
previous hash: 30bb9b475a08f2ea6fe07a1fd591ea209a7f633a400b2673b8835a975348b0eb
nonce: 2147489363
is orphan: 0
height: 1582196
depth: 2
hash: 50c8e5e51453c2ab85ef99d817e166540b40ef5fd2ed15ebc863091ca2a04594
difficulty: 51333809600
reward: 4634817937431
{
  "major_version": 7,
  "minor_version": 7,
  "timestamp": 1527458161,
  "prev_id": "30bb9b475a08f2ea6fe07a1fd591ea209a7f633a400b2673b8835a975348b0eb",
  "nonce": 2147489363,
  "miner_tx": {
    "version": 2,
```

```
19      "unlock_time": 1582256,
20      "vin": [ {
21          "gen": {
22            "height": 1582196
23          }
24        }
25      ],
26      "vout": [ {
27          "amount": 4634817937431,
28          "target": {
29            "key": "39abd5f1c13dc6644d1c43db68691996bb3cd4a8619a37a227667cf2bf055401"
30          }
31        }
32      ],
33      "extra": [ 1, 89, 148, 148, 232, 110, 49, 77, 175, 158, 102, 45, 72, 201, 193,
34      18, 142, 202, 224, 47, 73, 31, 207, 236, 251, 94, 179, 190, 71, 72, 251, 110,
35      134, 2, 8, 1, 242, 62, 180, 82, 253, 252, 0
36      ],
37      "rct_signatures": {
38        "type": 0
39      }
40    },
41    "tx_hashes": [ "e9620db41b6b4e9ee675f7bfdeb9b9774b92aca0c53219247b8f8c7aecf773ae",
42                   "6d31593cd5680b849390f09d7ae70527653abb67d8e7fdca9e0154e5712591bf",
43                   "329e9c0caf6c32b0b7bf60d1c03655156bf33c0b09b6a39889c2ff9a24e94a54",
44                   "447c77a67adeb5dbf402183bc79201d6d6c2f65841ce95cf03621da5a6bffefc",
45                   "90a698b0db89bbb0704a4ffa4179dc149f8f8d01269a85f46ccd7f0007167ee4"
46    ]
47  }
```

Campi di un Blocco

- (linee 2-10) - Informazioni raccolte dal nodo (monerod) che non appartengono propriamente alla struttura del blocco.

- `is orphan` (linea 5) - Indica se il blocco è orfano. Di solito, i nodi della rete conservano tutti i rami durante una situazione di biforcazione della blockchain, e scartano i rami non necessari quando emerge il ramo con difficoltà cumulativa più alta, lasciando di conseguenza dei blocchi orfani.

- `depth` (linea 7) - In una copia di una blockchain, la profondità di un blocco è la distanza relativa al blocco più recente aggiunto alla catena.

- `hash` (line 8) - L'identificativo del blocco (block ID).
- `difficulty` (linea 9) - La difficoltà della rete non è registrata in un blocco, dato che gli utenti possono dedurre la difficoltà di un blocco dal relativo timestamp secondo le modalità descritte nella Sezione 7.2.
- `major_version` (linea 12) - Indica la versione del protocollo utilizzata per validare questo blocco.
- `minor_version` (linea 13) - In origine era usato per un meccanismo di voti dedicato ai miner, adesso segue lo stesso scopo di `major_version`. Dal momento che il campo non occupa molto spazio, probabilmente gli sviluppatori hanno pensato che lo sforzo di eliminare questo campo non valesse l'eventuale beneficio tratto in termini di risparmio di memoria.
- `timestamp` (linea 14) - La rappresentazione del timestamp UTC del blocco sotto forma di numero interno, come riportato dal miner del blocco.
- `prev_id` (linea 15) - L'identificativo del blocco precedente. Qui risiede l'essenza della blockchain di Monero.
- `nonce` (linea 16) - Il nonce utilizzato dal miner di questo blocco per raggiungere il suo obiettivo di difficoltà (difficulty target). Chiunque può ricalcolare la Proof of Work e verificare che il nonce sia valido.
- `miner_tx` (linee 17-40) - La transazione del miner del blocco.
- `version` (linea 18) - Formato/era della versione della transazione; '2' corrisponde a RingCT.
- `unlock_time` (linea 19) - Gli output della transazione del miner non possono essere spesi fino al blocco 1582256th, ovvero dopo l'estrazione di 59 ulteriori blocchi (corrisponde ad un tempo di blocco pari a 60, dato che non possono essere spesi prima del passare di un'intervallo di tempo pari a 60 blocchi, ovvero $2 * 60 = 120$ minuti).
- `vin` (linee 20-25) - Input della transazione del miner. Ovviamente non ci sono, dato che le transazioni dei miner sono usate per generare le ricompense dei blocchi e raccogliere le commissioni di rete.
- `gen` (linea 21) - Abbreviazione di 'generate'.
- `height` (linea 22) - L'altezza della blockchain da cui si è generata la ricompensa al miner di questo blocco. Ogni blocco può generare una sola ricompensa e una sola volta.
- `vout` (linee 26-32) - Output della transazione del miner.
- `amount` (linea 27) - Ammontare distribuito dal miner della transazione, contenente la ricompensa del blocco e le commissioni derivanti dalle transazioni registrate, in unità atomiche.
- `key` (linea 29) - Indirizzi one-time che conferiscono la proprietà della ricompensa generata al miner.

- `extra` (linee 33-36) - Informazioni extra sulla transazione del miner, inclusa la chiave pubblica della transazione.
- `type` (linea 38) - Tipologia della transazione, in questo caso '0' per `RCTTypeNull`, indicando una transazione miner.
- `tx_hashes` (linee 41-46) - Tutti gli ID delle transazioni incluse in questo blocco, eccetto l'ID della transazione del miner, ovvero:
  `06fb3e1cf889bb972774a8535208d98db164394ef2b14ecfe74814170557e6e9`

APPENDICE C

Blocco Genesi

In questo appendice viene illustrata la struttura del blocco genesi di Monero. Il blocco non ha alcuna transazione (ed invia semplicemente la ricompensa a thankful_for_today [125]). Il fondatore di Monero non ha aggiunto un timestamp, forse come retaggio di Bytecoin, la moneta da cui il codice di Monero deriva, i cui creatori avrebbero apparentemente cercato di nascondere un'ampia quantità di monete pre-estratte [94] e che potrebbero essere coinvolti nella gestione di una rete poco trasparente di software e servizi legati alle criptovalute [132].

Il blocco 1 è stato aggiunto alla blockchain al timestamp 2014-04-18 10:49:53 UTC (come riportato dal miner del blocco), di conseguenza possiamo assumere che il blocco genesi è stato creato nello stesso giorno, e ciò corrisponde con la data di lancio annunciata da thankful_for_today [125].

```
print_block 0
timestamp: 0
previous hash: 0000000000000000000000000000000000000000000000000000000000000000
nonce: 10000
is orphan: 0
height: 0
depth: 1580975
hash: 418015bb9ae982a1975da7d79277c2705727a56894ba0fb246adaabb1f4632e3
difficulty: 1
reward: 17592186044415
{
  "major_version": 1,
```

```
13    "minor_version": 0,
14    "timestamp": 0,
15    "prev_id": "0000000000000000000000000000000000000000000000000000000000000000",
16    "nonce": 10000,
17    "miner_tx": {
18      "version": 1,
19      "unlock_time": 60,
20      "vin": [ {
21          "gen": {
22            "height": 0
23          }
24        }
25      ],
26      "vout": [ {
27          "amount": 17592186044415,
28          "target": {
29            "key": "9b2e4c0281c0b02e7c53291a94d1d0cbff8883f8024f5142ee494ffbbd088071"
30          }
31        }
32      ],
33      "extra": [ 1, 119, 103, 170, 252, 222, 155, 224, 13, 207, 208, 152, 113, 94, 188,
34      247, 244, 16, 218, 235, 197, 130, 253, 166, 157, 36, 162, 142, 157, 11, 200, 144,
35      209
36      ],
37      "signatures": [ ]
38    },
39    "tx_hashes": [ ]
40  }
```

Campi del Blocco Genesi

Dal momento in cui il comando per stampare a schermo il blocco genesi e il blocco dell'Appendice B è lo stesso, la struttura apparirà molto simile. Teniamo dunque conto delle differenze:

- `difficulty` (linea 9) - La difficoltà del blocco genesi è indicata con un valore pari a 1, e ciò significa che qualsiasi `nonce` potrebbe funzionare.

- `timestamp` (linea 14) - Il blocco genesi non ha un timestamp significativo.

- `prev_id` (linea 15) - Per convenzione vengono utilizzati 32 bytes vuoti per l'ID del blocco precedente.

- `nonce` (linea 16) - $n = 10000$ per convenzione.

- `amount` (linea 27) - Corrisponde esattamente alla ricompensa del primo blocco (17.592186044415 XMR) calcolata come nella Sezione 7.3.1.
- `key` (linea 29) - I primissimi Moneroj estratti furono distribuiti al fondatore di Monero thankful_for_today.
- `extra` (linee 33-36) - Utilizzo della codifica affrontata nell'Appendice A, Il campo `extra` della transazione del miner del blocco genesi contiene soltanto la chiave pubblica della transazione.
- `signatures` (linea 37) - Non ci sono firme nel blocco genesi. Questo è soltanto un artefatto del comando `print_block`. Lo stesso vale per il campo `tx_hashes` linea 39.