Principles of Cryptocurrency Design

Esercitazione

Francesco Pasquale

8 maggio 2025

1 Consenso e Proof-of-Work

Per gli esercizi seguenti si vedano i file in pcd250407.

Esercizio 1. Il programma `block.py` genera una sequenza di blocchi a partire dal genesis block di Bitcoin e li scrive su un file `blockchain.dat`. Scrivere un programma che legga il file `blockchain.dat` e verifichi che

1. Ogni blocco contiene nel campo `prev_hash` l'id del blocco precedente;
2. L'id di ogni blocco è inferiore al `target` contenuto nel blocco.

Esercizio 2. Nel codice scritto a lezione abbiamo usato un un `target` costante. Definire un tempo medio `DELTA` che vogliamo imporre fra la creazione di due blocchi consecutivi (diciamo `DELTA = 120` secondi) e un numero `EPOCH_LEN` che corrisponde al numero di blocchi prima di riaggiornare il target (diciamo `EPOCH_LEN` $= 60$).

Modificare il codice in `block.py` in modo che, `target`(0) sia quello del *genesis block* e viene usato per i primi `EPOCH_LEN` blocchi, ma ogni volta che il numero di blocchi creati è $k *$ `EPOCH_LEN` per qualche $k \geqslant 1$, per i successivi `EPOCH_LEN` blocchi il target sia dato dalla formula

$$\texttt{target}(k) = \texttt{target(k-1)} \cdot \frac{\delta}{\texttt{DELTA}}$$

dove con δ abbiamo indicato la differenza fra il `timestamp` del blocco $k *$ `EPOCH_LEN` e il `timestamp` del blocco $(k-1) *$ `EPOCH_LEN`.

Esercizio 3. Scrivere un programma che legga un file `blockchain.dat` e, se contiene una catena che soddisfa i due punti dell'Esercizio 1, restituisca la *proof-of-work* della catena, ossia la somma, per ogni blocco, di 2^{256} diviso `target` $+ 1$.

Esercizio 4. Scrivere un programma che legga un file `blockchain.dat` e, se contiene una catena che soddisfa i due punti dell'Esercizio 1, inizi ad aggiungere blocchi alla catena.

Esercizio 5. Il `target` attuale (blocco n. `891953`) della blockchain di Bitcoin espresso in decimale è $151813 \cdot 2^{160}$.

1. Sapendo che il `target` è impostato in modo che, in media, viene creato un nuovo blocco ogni dieci minuti, stimare l'*hash rate* h (ossia il numero di hash per secondo) che i miner stanno complessivamente eseguendo attualmente.

2. Supponete che un extraterrestre arrivi sulla terra con un *hardware* che riesce a fare k volte più hash al secondo di quanti ne fanno tutti i miner attuali messi insieme, per qualche $k > 1$, e decida di voler usare il suo hardware per riscrivere l'intera blockchain di Bitcoin a partire dal *genesis block*. Assumendo che l'hash rate degli altri miner rimanga costante, calcolare approssimativamente quanto tempo passerebbe prima che, in accordo al protocollo Bitcoin, tutti riconoscano la blockchain dell'extraterreste come quella valida.

Esercizio 6. Assumendo che la funzione crittografica `sha256` sia un *random oracle*, qual è la probabilità che il prossimo blocco impieghi più di k minuti per essere creato, per $k > 10$?

2 Transazioni

Per gli esercizi seguenti si vedano i file in pcd250414.

Esercizio 7. Nel file `helpers.py` abbiamo implementato la funzione `varint2int` che prende in input uno *stream* di byte e restituisce il numero intero corrispondente, secondo le specifiche definite qui:
`https://developer.bitcoin.org/reference/transactions.html#compactsize-unsigned-integers`

Scrivere una funzione `int2varint` che esegua l'operazione inversa: prenda in input un intero non negativo minore di 2^{64} e restituisca una sequenza di byte che codifichi l'intero secondo le specifiche.

Esercizio 8. Nel file `transaction.py`, per ognuna delle classi `Tx`, `TxIn` e `TxOut` abbiamo implementato un metodo di classe `parse` che prende in input una sequenza di byte opportuna e restituisce l'oggetto codificato nella sequenza.

Per ognuna delle classi, scrivere un metodo `serialize` che restituisca una sequenza di byte contenente la serializzazione dell'oggetto della classe.

Esercizio 9. Scrivere un metodo `tot_out` per la classe `Tx` che restituisca la somma degli `amount` contenuti negli output della transazione.

Esercizio 10. Scrivere un metodo `fee` per la classe `Tx` che restituisca la *transaction fee* della transazione. La transaction fee è la differenza fra la somma degli `amount` contenuti negli output puntati dagli input della transazione e la somma degli `amount` contenuti negli output della transazione. Per recuperare gli output puntati dagli input di una transazione potete usare, per esempio, le api di un qualche sito che consente di recuperare dati dalla Blockchain, come mostrato per esempio nel file `get_tx.py`.

Esercizio 11. Progettare e implementare un metodo della classe `Tx` che restituisca in output un albero di cui l'istanza dell'oggetto è il nodo radice. I figli di un nodo/transazione u sono le transazioni i cui output sono puntati dagli input di u. Le foglie di un tale albero perciò saranno tutte transazioni *coinbase*.

3 Script

Tutte le prime transazioni di Bitcoin avevano dei `locking_script` del tipo *p2pk*. Successivamente, diversi nuovi tipi di locking script sono stati introdotti: il secondo, dopo p2pk, è stato quello chiamato *Pay-to-Public-Key-Hash (p2pkh)*. In questo tipo di `locking_script` non compare direttamente la chiave pubblica, ma l'hash della chiave pubblica `<pkhash>`. Complessivamente `locking_script` è formato da cinque comandi:

DUP HASH160 `<pkhash>` EQUALVERIFY CHECKSIG (1)

dove

- DUP è un'istruzione che inserisce nello stack una copia dell'elemento sottostante. Per esempio, lo script `<1>` DUP verrebbe eseguito in questo modo

- HASH160 è un'istruzione che estrae dallo stack l'elemento sottostante e inserisce nello stack il suo hash (opportunamente calcolato);
- `<pkhash>`è l'hash di una chiave pubblica;
- EQUALVERIFY è un'istruzione che estrae due elementi sottostanti dallo stack e se sono uguali non fa nulla, altrimenti interrompe l'esecuzione dello script dichiarando la transazione non valida;
- CHECKSIG è il comando che abbiamo visto in precedenza in *p2pk*.

Esercizio 12. Qual è lo `unlocking_script` che consente di sbloccare un `locking_script` del tipo *p2pkh*?

Esercizio 13. L'istruzione EQUAL estrae due elementi sottostanti dallo stack e inserisce nello stack TRUE se i due elementi sono uguali e FALSE altrimenti. Scrivere un `locking_script` che può essere sbloccato solo da chi conosce un messaggio `msg` il cui hash è `<msg_hash>`.

Esercizio 14. L'istruzione NOT estrae un elemento dallo stack e, se è TRUE o FALSE ne inserisce la negazione nello stack, altrimenti inserisce nello stack FALSE. L'istruzione VERIFY estrae un elemento dallo stack e, se è TRUE non fa nulla altrimenti interrompe l'esecuzione e la transazione è non valida. L'istruzione SWAP estrae due elementi dallo stack e li reinserisce nello stack in ordine inverso. Usando queste e le altre istruzioni viste in precedenza, progettare un `locking_script` che può essere sbloccato solo con un `unlocking_script` che contenga un messaggio `<msg2>` diverso da un dato messaggio `<msg>` che però abbia lo stesso valore di hash (calcolato con HASH160).

Per gli esercizi seguenti si vedano i file in pcd250424.

Esercizio 15. Nel file `script.py` abbiamo implementato un metodo di classe `parse` che prende in input una sequenza di byte opportuna e restituisce l'oggetto codificato nella sequenza. Scrivere un metodo `serialize` che restituisca una sequenza di byte contenente la serializzazione dell'oggetto della classe.

Esercizio 16. Fare il parsing del seguente *locking script* `6e879169a87ca887`. Usando le descrizioni degli OP_CODES[1], determinare cosa dovrebbe contenere un *unlocking script* per ottenere uno script che restituisca TRUE.

[1] Le trovate, per esempio, qui https://wiki.bitcoinsv.io/index.php/Opcodes_used_in_Bitcoin_Script

4 Curve ellittiche e indirizzi Bitcoin

Possiamo descrivere una curva ellittica come l'insieme dei punti (x, y), a coordinate in un opportuno campo, che soddisfano l'equazione

$$y^2 = x^3 + ax + b \tag{2}$$

dove a e b sono parametri che definiscono la curva, con l'aggiunta di un punto che chiamiamo *punto all'infinito* $\{\infty\}$.

Esercizio 17. Scrivere un programma che prenda in input i parametri a e b e gli estremi di un intervallo di numeri reali, $[x_0, x_1] \subseteq \mathbb{R}$ e disegni il grafico della curva in (2) al variare di $x \in [x_0, x_1]$.

Per le applicazioni in crittografia non si considera il campo dei numeri reali, ma i campi finiti $\mathbb{F}_p$, dove p è un numero primo, ossia $\mathbb{F}_p$ è l'insieme dei numeri interi $\{0, 1, \ldots, p-1\}$ con le usuali operazioni di somma e prodotto, modulo p.

La curva ellittica che viene usata in Bitcoin si chiama secp256k1 ed è definita dai parametri $a = 0$, $b = 7$, $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$.

Esercizio 18. Scrivere un programma per verificare che il numero p definito qui sopra è un numero primo.

Su una curva ellittica

$$\mathcal{C} = \{(x, y) \in \mathbb{F}_p^2 \,:\, y^2 = x^3 + ax + b\} \cup \{\infty\}$$

si può definire un'operazione, che chiamiamo *somma*, che a due punti sulla curva $P, Q \in \mathcal{C}$ associa un terzo punto sulla curva $P + Q \in \mathcal{C}$, in modo tale che $(\mathcal{C}, +)$ sia un gruppo.

È facile vedere che, siccome $(\mathcal{C}, +)$ è un gruppo, c'è un algoritmo polinomiale per il seguente problema computazionale
INPUT: Un punto sulla curva $G \in \mathcal{C}$ e un intero $k \in \mathbb{N}$
OUTPUT: Il punto $P \in \mathcal{C}$ tale che $P = kG$
Dove con kP intendiamo $P + P + \cdots + P$, k volte.

Esercizio 19. Descrivere un algoritmo che, assumendo che $P + P$ si possa calcolare in tempo $\mathcal{O}(1)$, calcoli kP in tempo $\mathcal{O}(\log k)$.

D'altra parte, nessuno conosce un algoritmo polinomiale per il problema computazionale inverso (Discrete Logarithm Problem)
INPUT: Due punti sulla curva $G, P \in \mathcal{C}$
OUTPUT: Un intero $k \in \mathbb{N}$ tale che $P = kG$, oppure `None` se un tale intero non esiste.

Perciò, dato un *punto base* $G \in \mathcal{C}$ si può definire una coppia di chiavi $(\texttt{sk}, \texttt{pk})$ dove la chiave segreta `sk` è un intero positivo minore di p, e la chiave pubblica `pk` è il punto sulla curva $\texttt{sk} \cdot G$

Il punto base della curva SECP256K1 è $G = (x, y)$, dove

$x =$ `0x 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798`
$y =$ `0x 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8`

Esercizio 20. Verificare che le coordinate (x, y) qui sopra soddisfano l'equazione $y^2 = x^3 + 7 \mod p$.

Esercizio 21. L'ordine del gruppo generato dal punto base della curva è

$$n = \texttt{0x FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141}$$

Verificare che n è un numero primo.

Per gli esercizi seguenti si vedano i file in pcd250505.

Esercizio 22. A lezione abbiamo generato un indirizzo bitcoin-testnet a partire dal numero

$$r = \texttt{1032026271963554642597181681368518600454288997291561949319905424087874760107 2}$$

e abbiamo inviato a quell'indirizzo 0.001 bitcoin-testnet. Tovare il modo di spostare quei 0.001 bitcoin-testnet inviandoli a un altro indirizzo sotto il vostro controllo.

Esercizio 23. Il Wallet Import Format (WIF) è un formato standard con cui è possibile esportare e importare chiavi private fra wallet bitcoin. Per ottenerlo, dalla sequenza di byte `sk` che costituiscono la chiave privata si procede i questo modo

1. Si aggiunge come prefisso il byte `0xEF` per la *testnet* e il byte `0x80` per la *mainnet*;
2. Se si vuole considerare una public key in formato compresso si aggiunge come suffisso il byte `0x01`
3. Della sequenza di byte così ottenuta si calcola l'`hash256` e i primi quattro byte di questo hash vengono aggiunti come suffisso;
4. Si codifica la sequenza di byte così ottenuta in `base58`

Scrivere un metodo `wif` che restituisca la chiave privata in quel formato. Testare il vostro metodo, generando chiavi private e provando a importarle in un wallet standard, per esempio Electrum.

Esercizio 24. Scrivere un metodo di classe PARSE_WIF che legga una stringa di byte in formato WIF e restituisca l'oggetto corrispondente della classe ADDRESS.

Esercizio 25. La libreria *random* che abbiamo usato a lezione per fare i nostri test non è sicura per generare numeri casuali da usare come chiavi private. Il seguente codice genera una sequenza di 32 byte (256 bit) in modo più sicuro

```
import secrets
sk_string = secrets.token_bytes(32)
print(sk_string.hex())
```

Scrivere un programma che trovi una chiave privata `sk` "sicura" e tale che l'indirizzo Bitcoin ottenuto a partire da quella chiave inizi con `1PCD`.

Esercizio 26. Verificare che uno degli indirizzi contenuti negli output script P2PKH della transazione
b35d91a71f226ba961162ca18f321b4d9aada8a0e722430ad3d2d2e4dda9a2c0,
l'indirizzo `1CLJKodMLHhAwiDYFWDiwmz31cMfhqKnEc`, è l'hash di una chiave pubblica non compressa che ha come come chiave privata lo `sha256` della stringa `Francesco`. Quella transazione ha 500 output del tipo P2PKH tutti con lo stesso ammontare. Scrivere un programma *brute-force* che cerchi di individuare se in quella transazione ci sono altri indirizzi che hanno come chiave privata l'hash `sha256` di altri nomi.

Esercizio 27. Ottenere dei bitcoin testnet tramite qualche faucet, provare a eseguire delle transazioni verso indirizzi con chiavi private facilmente individuabili tramite brute-force e vedere quanto tempo "resistono" prima che qualche bot ne individui la chiave privata e li spenda.