# Principles of Cryptocurrency Design

Esercitazione

Francesco Pasquale

31 marzo 2025

## 1 Il protocollo di Dolev-Strong

Consideriamo il protocollo di Dolev-Strong per il problema Byzantine Broadcast.

**Algorithm 1** Dolev-Strong Protocol for Byzantine Broadcast

1: ROUND 0. Ogni nodo $i$ inizializza un insieme vuoto $\mathcal{E}_i = \emptyset$.
2: La sorgente (il nodo 1) riceve in input $b$ e invia $\langle b \rangle_1$ a tutti i nodi.
3: PER OGNI ROUND $r = 1, \ldots, f$. Ogni nodo $i$:
4: Per ogni messaggio $\langle \hat{b} \rangle_{1,j_1,j_2,\ldots,j_{r-1}}$ con $r$ firme distinte (inclusa quella
5: della sorgente) ricevuto nel Round $r-1$:
6: SE $\hat{b} \notin \mathcal{E}_i$:
7: Aggiungi $\hat{b}$ a $\mathcal{E}_i$;
8: Firma il messaggio $\langle \hat{b} \rangle_{1,j_1,j_2,\ldots,j_{r-1}}$;
9: Invia il messaggio firmato $\langle \hat{b} \rangle_{1,j_1,j_2,\ldots,j_{r-1},i}$ a tutti i nodi;
10: ROUND $f+1$. Ogni nodo $i$:
11: Per ogni messaggio $\langle \hat{b} \rangle_{1,j_1,j_2,\ldots,j_f}$ con $f+1$ firme distinte (inclusa quella
12: della sorgente) ricevuto nel Round $f$:
13: SE $\hat{b} \notin \mathcal{E}_i$, aggiungi $\hat{b}$ a $\mathcal{E}_i$;
14: SE $\mathcal{E}_i$ contiene un solo elemento, allora OUTPUT l'elemento in $\mathcal{E}_i$,
15: Altrimenti OUTPUT 0



**Esercizio 1.** Alle linee 4 e 11 del protocollo ogni nodo onesto verifica, oltre al *numero* di firme distinte presenti ($r$ firme per messaggi arrivati nel round $r-1$), che fra le firme ci sia anche la firma della sorgente. Descrivere un attacco al protocollo che i nodi corrotti potrebbero mettere in atto se i nodi onesti non verificassero la presenza della firma della sorgente.

**Esercizio 2.** Descrivere un attacco al protocollo che i nodi corrotti potrebbero mettere in atto se fossero in numero maggiore di $f$.

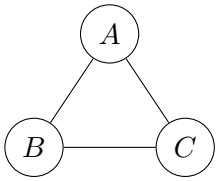
**Esercizio 3.** Supponete che c'è un *bug* nell'implementazione del sistema di firme digitali usato nel protocollo che consente a un nodo corrotto di falsificare le firme dei nodi onesti. Descrivere un attacco al protocollo che i nodi corrotti possono mettere in atto in questo caso.

**Esercizio 4.** Se l'insieme $F \subseteq [n]$ dei nodi corrotti è noto a priori, il problema Byzantine Broadcast (BB) diventa banale. Progettare un protocollo che prende in input l'insieme $F \subseteq [n]$ dei nodi corrotti, esegue un solo round di comunicazione, e soddisfa *validity* e *consistency* se tutti i nodi in $[n] \setminus F$ sono onesti.

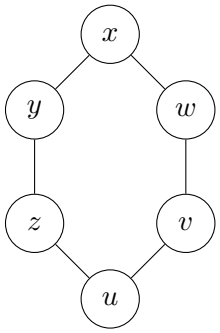
## 2 Lower bound per Byzantine Broadcast senza PKI

**Esercizio 5.** Considerate i sistemi $G$ e $H$ qui di seguito.

Sistema $G$




Sistema $H$




1. Eseguire il protocollo di Dolev-Strong con $f = 1$ nel sistema $G$, quando tutti i nodi sono onesti, $A$ è il nodo sorgente e riceve in input 1.

2. Eseguire il protocollo di Dolev-Strong con $f = 1$ nel sistema $H$ in cui $x$ e $u$ sono copie esatte[1] di $A$ e ricevono in input 1, $y$ e $v$ sono copie esatte di $B$ e $z$ e $w$ sono copie esatte di $C$ (tutti i nodi sono onesti e seguono il protocollo).

3. Osservare che le esecuzioni dei due punti precedenti sono equivalenti ($A$, $B$ e $C$ non possono distinguere se sono nel sistema $G$ o nel sistema $H$).

4. Eseguire il protocollo di Dolev-Strong con $f = 1$ nel sistema $H$ in cui $x$ e $u$ sono copie esatte di $A$, ma alla copia di $A$ in $x$ viene dato in input 1 mentre alla copia di $A$ in $u$ viene dato in input 0; $y$ e $v$ sono copie esatte di $B$ e $z$ e $w$ sono copie esatte di $C$ (tutti i nodi sono onesti e seguono il protocollo). Quali sono gli output dei sei nodi?

5. È possibile simulare l'esecuzione in $H$ del punto precedente con una esecuzione equivalente nel sistema $G$ in cui

   (a) $B$ e $C$ sono nodi onesti e $A$ è un nodo corrotto che simula il comportamento dei nodi $x, u, v$ e $w$?

   (b) $A$ e $B$ sono nodi onesti e $C$ è un nodo corrotto che simula il comportamento dei nodi $z, u, v$ e $w$?



**Esercizio 6.** A lezione abbiamo dimostrato che, in assenza di PKI non esistono protocolli per il problema Byzantine Broadcast che soddisfano *validity* e *consistency* per tre nodi, se almeno uno dei tre è corrotto.

Generalizzare la dimostrazione al caso di un numero di nodi arbitrario: dato un $n$ qualunque, in assenza di PKI nessun protocollo per Byzantine Broadcast può soddisfare *validity* e *consistency*, se almeno $n/3$ dei nodi sono corrotti.

## 3 Randomized Byzantine Broadcast senza PKI

Sia $H : \mathbb{N} \cup \{0\} \to [n]$ una funzione tale che $H(0) = 1$ e per ogni $r \in \mathbb{N}$ si comporta come un *random oracle*. A lezione abbiamo dimostrato che il protocollo seguente per Byzantine Broadcast soddisfa *Validity* con probabilità 1 e soddisfa *Consistency* con probabilità almeno $1 - (2/3)^{k-1}$.

[1] Ossia entrambe hanno la stessa chiave privata

**Algorithm 2** Randomized Byzantine Broadcast

1: ROUND 0. La sorgente (il nodo 1) riceve in input $b$ e inizializza $\mathtt{sb}_1 = b$.
2: Ogni nodo $i$ inizializza $\mathtt{sb}_i = \perp$.
3: PER OGNI ITERAZIONE $r = 0, \ldots, k-1$:
4: ROUND $3r$. Il *leader* $\ell_r = H(r)$ dell'iterazione:
5: Se $\mathtt{sb}_{\ell_r} \neq \perp$ allora invia a tutti $\mathtt{sb}_{\ell_r}$;
6: Altrimenti sceglie un bit $\{0, 1\}$ u.a.r. e lo invia a tutti.
7: ROUND $3r + 1$. Ogni nodo $i$:
8: Se $\mathtt{sb}_i \neq \perp$ allora invia a tutti $\mathtt{sb}_i$;
9: Altrimenti invia a tutti il bit ricevuto nel round $3r$ dal leader $\ell_r$ (se non ha ricevuto nulla da $\ell_r$ o ha ricevuto entrambi i bit, invia 0 o 1 arbitrariamente)
10: ROUND $3r + 2$. Ogni nodo $i$:
11: Se c'è un bit $\hat{b}$ che ha ricevuto nel round $3r + 1$ da almeno $2n/3$ nodi distinti allora imposta $\mathtt{sb}_i = \hat{b}$;
12: Altrimenti imposta $\mathtt{sb}_i = \perp$.
13: ROUND $3k$. Ogni nodo $i$:
14: OUTPUT $\mathtt{sb}_i$

**Esercizio 7.** Si consideri il protocollo in Algorithm 2. Dato $\delta > 0$, quanto deve essere grande il numero di iterazioni $k = k(\delta)$ affinché la probabilità che l'algoritmo soddifi la condizione di *consistency* sia almeno $1 - \delta$? Quanto deve essere grande $k$ se scegliamo $\delta = 1/n$? Confrontare il numero di round di questo protocollo con quello del protocollo di Dolev-Strong.

**Esercizio 8.** Alla linea 11 il protocollo stabilisce che ogni nodo $i$ deve decidere come impostare il bit $\mathtt{sb}_i$ a seconda che abbia ricevuto o no almeno $2n/3$ "voti" per uno specifico bit $\hat{b}$ nel round precedente.

1. Mostrare un attacco che i nodi corrotti potrebbero mettere in atto se la decisione fosse presa in funzione di un numero di voti $h < 2n/3$;
2. Mostrare un attacco che i nodi corrotti potrebbero mettere in atto se la decisione fosse presa in funzione di un numero di voti $h > 2n/3$.

**Esercizio 9.** Cosa succederebbe se non imponessimo che $H(0) = 1$ (ossia che il leader $\ell_0$ dell'iterazione $r = 0$ è il nodo sorgente)?

# 4 Modello asincrono

Si ricordi che un *protocollo* nel modello asincrono è *event driven*, cioè specifica cosa fa un nodo nel momento in cui riceve un messaggio: il nodo può eseguire un numero arbitrario di operazioni locali, cambiare *stato* (lo *stato* di un nodo è costituito dai valori delle sue variabili locali, che possono dipendere dall'input e da tutti i messaggi ricevuti). Una configurazione $C$ è una coppia $C = (S, M)$ dove $S$ rappresenta lo stato di ogni nodo e $M$ è l'insieme di messaggi non ancora consegnati, la *message pool*. Un messaggio $m$ è una coppia $m = (p, x)$ dove $p$ è il nodo destinatario del messaggio e $x$ è il contenuto del messaggio.

Se $\Pi$ è un protocollo nel modello asincrono, data una configurazione $C = (S, M)$ e un messaggio $m = (p, x)$, indichiamo con $m(C)$ la configurazione $C' = (S', M')$ che si ottiene in base al protocollo quando il messaggio $m$ viene consegnato, ossia l'insieme $S'$ è quello che si ottiene da $S$ aggiornando lo stato del nodo $p$ dopo l'elaborazione delle informazioni contenute in $x$ e $M'$ è la *message pool*

che si ottiene da $M$ togliendo il messaggio appena consegnato $m$ e aggiungendo tutti i messaggi inviati da $p$ a seguito della ricezione del messaggio $m$.

**Esercizio 10.** Sia $\Pi$ un protocollo deterministico nel modello asincrono, sia $C = (S, M)$ una configurazione e siano $m_1 = (p_1, x_1)$ e $m_2 = (p_1, x_2)$ due messaggi in $M$. Osservare che se $p_1 \neq p_2$ allora

- $m_2$ sta nella *message pool* della configurazione $m_1(C)$ e $m_1$ sta nella *message pool* della configurazione $m_2(C)$;
- $m_2(m_1(C)) = m_1(m_2(C))$.

---

Uno *schedule* $\sigma$ è una sequenza ordinata di messaggi $\sigma = \big((p_1, x_1), \ldots, (p_k, x_k)\big)$. Diciamo che uno schedul $\sigma$ è *applicabile* a una configurazione $C$ se $m_1 \in M$ e se per ogni $i = 2, \ldots, k$, il messaggio $m_i$ appartiene alla *message pool* della configurazione $\sigma_{[1:i-1]}(C)$, dove con $\sigma_{[1:i-1]}$ intendiamo lo *schedule* formato dai primi $i-1$ messaggi dello *schedule* $\sigma$. Dato un protocollo $\Pi$, una configurazione $C$ e uno schedule $\sigma$, indichiamo con $\sigma(C)$ la configurazione che si ottiene partendo da $C$ in base al protocollo $\Pi$ dopo la consegna di tutti i messaggi in $\sigma$ (nell'ordine stabilito dalla sequenza).

**Esercizio 11.** Sia $\Pi$ un protocollo deterministico nel modello asincrono, sia $C = (S, M)$ una configurazione e siano $\sigma_1 = \big((p_1, x_1), \ldots, (p_k, x_k)\big)$ e $\sigma_2 = \big((q_1, y_1), \ldots, (q_h, y_h)\big)$ due *schedule* applicabili a $C$. Osservare che se gli insiemi di nodi destinatari nei due schedule sono disgiunti, $\{p_1, \ldots, p_k\} \cap \{q_1, \ldots, q_h\} = \emptyset$, allora

- $\sigma_2$ è applicabile a $\sigma_1(C)$ e $\sigma_1$ è applicabile a $\sigma_2(C)$;
- $\sigma_2(\sigma_1(C)) = \sigma_1(\sigma_2(C))$.

# 5 Funzioni Hash Crittografiche e Firme Digitali

Con il programma seguente (o con qualunque altra implementazione della funzione hash crittografica sha256) potete verificare che l'hash espresso in esadecimale della stringa *Francesco 16114492071* inizia con una sequenza di 9 zeri.

```
from hashlib import sha256

nonce = '16114492071'
text = 'Francesco ' + nonce

print(sha256(text.encode('utf8')).hexdigest())
```

**Esercizio 12.** Scrivere un programma che trovi una stringa `<nonce>` tale che l'hash della stringa `<nome> <nonce>`, per il vostro `<nome>` inizi con una sequenza di 9 zeri. Stimare il *running time* del programma ed eseguirlo.

Le due linee seguenti sono un estratto di un file `/etc/shadow` contenente gli hash delle password degli utenti di un sistema GNU/Linux

```
satoshi:$y$j9T$2wTxPAjXwsoqXJc2eiYbb1$iERNSCCzND12k9zqBms.CqAC7CtzaGQhJnao/53Jjh2:
nakamoto:$y$j9T$RrPRoX82jNdhTLDHuVmhY1$UIzs5jXCaEiB8lgXyBbQ8uwiOYoulfguzeTl2mPUx41:
```

Dalla pagina di manuale shadow vediamo che in ogni riga i campi sono separati da ":" (nel caso in questione sono omessi tutti i campi esclusi i primi due). Il primo campo è lo *username* dell'utente. Il secondo campo è a sua volta diviso in tre parti separate dal simbolo $: la prima indica lo schema utilizzato per ottenere l'hash della password, in questo caso `y` indica che lo schema utilizzato è yescrypt. Le altre tre parti indicano rispettivamente i parametri passati allo schema, il *salt* e l'hash ottenuto.

Nonostante questo sistema di memorizzazione degli hash delle password sia molto sicuro, gli utenti possono sempre scegliere password "deboli". Nell'esempio in questione infatti uno dei due utenti ha scelto una password adeguatamente complessa, l'altro ha scelto una password molto debole.

**Esercizio 13.** Usando un opportuno software per il *password cracking* (per esempio, John the Ripper) determinare quale dei due utenti ha una password debole.

**Esercizio 14.** Il codice Python seguente[2]

**Algorithm 3** Hashed RSA

```
 1. from Crypto.PublicKey import RSA
 2. from hashlib import sha256
 3.
 4. keyPair = RSA.generate(bits = 1024)
 5.
 6. msg = b'Benvenuti al corso di Principles of Cryptocurrency Design - AA 24/25!'
 7. msg_hash = int.from_bytes(sha256(msg).digest(), byteorder = 'big')
 8. msg_sig = pow(msg_hash, keyPair.d, keyPair.n)
 9.
10. print("Firma: ", hex(msg_sig))
```

ha scritto a video questa stringa:

```
Firma: 0x1c46bba62ba574a4b048adc57226ae9f0538a2da7202100d93ff0c41cd3767d8074690b
e996e8f0e9c38f75f222bb15c7dbd48db5211d08f1d072f196357fd047169530d0393f4c6f569eddf
8286f7742a756708615cd995a10f9ee3335db9c79f14daaec77fb12e94bd07435df767d618825591c
7413f29d572e7a320514ba0
```

Scrivere un programma per verificare che si tratta di una firma valida del messaggio *Benvenuti al corso di Principles of Cryptocurrency Design - AA 24/25!* generata con la chiave privata associata alla chiave pubblica

```
n = 0xaded882f418c88459440a37076cc42a9b2f40c56d6308ebd32f1724631f5035cea206d17200
8884c3993670e44362f2624ff6ea3f508f9d085f05d1a2487ea4949578588ff2c551a05a0f98369e2
5dc7441237c785e8ea58379ebe54cf82e9067c02457d3ebde78e7daeaabe4ede6cd561820ed136ed6
89b7e70b79a412ad509

e = 0x10001
```

**Esercizio 15.** Supponiamo che, invece di firmare l'*hash* del messaggio (linee 7 e 8 in Algorithm 3), avessimo firmato direttamente il messaggio:

1. Scrivere un programma per verificare che la seguente

   ```
   Firma: 0x87106345c3d2bbde2f7a778fc3551c1559761c15f032efbc4351c0f14e453a95e4
   2e419461adf461a76883680b9bfd73f3d032cb1b7996ab371039303a5ef0e9088bfa12c996ae
   ```

[2]La libreria `hashlib` è built-in Python. Per `Crypto` usare la libreria `pycryptodome` `https://pypi.org/project/pycryptodome/`

**Algorithm 4** Textbook RSA

```
1. from Crypto.PublicKey import RSA
2.
3. keyPair = RSA.generate(bits = 1024)
4.
5. msg = b'Benvenuti al corso di Principles of Cryptocurrency Design - AA 24/25!'
6. msg_sig = pow(int.from_bytes(msg, byteorder='big'), keyPair.d, keyPair.n)
7.
8. print("Firma: ", hex(msg_sig))
```

`deda21cfde926eee625de2ae1a3319ce594b23264da46b7912e3659087ebe58cd890636ad3a7cfdf71a3e6f574d7a1d0452260d2a3b1a22549`

è una firma valida del messaggio *Benvenuti al corso di Principles of Cryptocurrency Design - AA 24/25!* generata con lo schema in Algorithm 4 con la chiave privata associata alla stessa chiave pubblica dell'esercizio precedente.

2. Mostrare che lo schema di firma digitale in Algorithm 4 non è sicuro trovando una coppia di numeri interi (`fake_msg`, `fake_sig`) tale che `fake_sig` risulti una firma valida per `fake_msg` relativamente alla chiave pubblica del punto precedente.[3]

3. Notare il motivo per cui l'attacco del punto precedente invece non è attuabile sullo schema di firma digitale in Algorithm 3.



[3] Hint: Partire da una `fake_sig` arbitraria e trovare un `fake_msg` che ha `fake_sig` come firma.