# Principles of Cryptocurrency Design

Appunti ed Esercizi

Francesco Pasquale

5 marzo 2026

## 1 Il protocollo di Dolev-Strong

Consideriamo il protocollo di Dolev-Strong [1] per il problema Byzantine Broadcast.

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

1: ROUND 0. Ogni nodo $u$ inizializza un insieme vuoto $\mathcal{E}_u = \emptyset$.
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 $u$:
4: Per ogni messaggio $\langle \hat{b} \rangle_{1,\sigma_1,\sigma_2,\ldots,\sigma_{r-1}}$ con $r$ firme distinte (inclusa quella
5: della sorgente) ricevuto nel Round $r-1$:
6: SE $\hat{b} \notin \mathcal{E}_u$:
7: Aggiungi $\hat{b}$ a $\mathcal{E}_u$;
8: Firma il messaggio $\langle \hat{b} \rangle_{1,\sigma_1,\sigma_2,\ldots,\sigma_{r-1}}$;
9: Invia il messaggio firmato $\langle \hat{b} \rangle_{1,\sigma_1,\sigma_2,\ldots,\sigma_{r-1},\sigma_u}$ a tutti i nodi;
10: ROUND $f+1$. Ogni nodo $u$:
11: Per ogni messaggio $\langle \hat{b} \rangle_{1,\sigma_1,\sigma_2,\ldots,\sigma_f}$ con $f+1$ firme distinte (inclusa quella
12: della sorgente) ricevuto nel Round $f$:
13: SE $\hat{b} \notin \mathcal{E}_u$, aggiungi $\hat{b}$ a $\mathcal{E}_u$;
14: SE $\mathcal{E}_u$ contiene un solo elemento, allora OUTPUT l'elemento in $\mathcal{E}_u$,
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 quella 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 ci sia 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.** Ripercorrere la dimostrazione vista a lezione del fatto che il protocollo di Dolev-Strong soddisfa *validity* e *consistency* se il numero di nodi corrotti è minore o uguale a $f$. (Capitolo 3 in [4])

**Esercizio 5.** 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 Schemi di firme digitali

Uno schema di firma digitale è una tripla di algoritmi (probabilisici) polinomiali (KeyGen, Sign, Verify):

- $\textsc{KeyGen}(1^\lambda) = (\mathtt{sk}, \mathtt{pk})$
  Prende in input un *security parameter* $\lambda \in \mathbb{N}$ (in unario $1^\lambda$) e restituisce una coppia di chiavi $\mathtt{sk} \in \{0,1\}^{p(\lambda)}$ e $\mathtt{pk} \in \{0,1\}^{p(\lambda)}$, dove $p$ è un polinomio (in altri termini, la lunghezza delle chiavi è polinomiale in $\lambda$);

- $\textsc{Sign}(\mathtt{sk}, m) = \sigma$
  Prende in input una chiave segreta $\mathtt{sk}$ e un messaggio $m \in \{0,1\}^*$ e restituisce una *firma* $\sigma \in \{0,1\}^{p(\lambda)}$;

- $\textsc{Verify}(m, \mathtt{pk}, \sigma) = \textsc{True}/\textsc{False}$
  Prende in input un messaggio $m \in \{0,1\}^*$, una chiave pubblica $\mathtt{pk}$ e una firma $\sigma$ e restituisce True o False. Diciamo che $\sigma$ è una firma *valida* per $m$ rispetto alla chiave pubblica $\mathtt{pk}$ se $\textsc{Verify}(m, \mathtt{pk}, \sigma) = \textsc{True}$.

Lo schema è *corretto* se per ogni $\lambda \in \mathbb{N}$, per ogni coppia di chiavi $(\mathtt{sk}, \mathtt{pk})$ generata con $\textsc{KeyGen}(1^\lambda)$ e per ogni messaggio $m \in \{0,1\}^*$ risulta che

$$\textsc{Verify}\left(m, \mathtt{pk}, \textsc{Sign}(\mathtt{sk}, m)\right) = \textsc{True}$$

Diciamo che lo schema è *sicuro* se, data una coppia di chiavi $(\mathtt{sk}, \mathtt{pk})$ generata con $\textsc{KeyGen}(1^\lambda)$, dato un qualunque algoritmo (probabilistico) polinomiale $\mathcal{A}$ che

1. Prende in input $\mathtt{pk}$;
2. Può fare chiamate alla funzione $\textsc{Sign}(\mathtt{sk}, \cdot)$ (e quindi ottenere un numero polinomiale di coppie $(\hat{m}, \hat{\sigma})$, dove $\hat{\sigma}$ è una firma valida per $\hat{m}$ rispetto a $\mathtt{pk}$);
3. Restituisce in output una coppia $(m, \sigma)$;

allora per la coppia $(m, \sigma)$ data in output dall'algoritmo deve valere che, se $m$ non è uno dei messaggi su cui l'algoritmo ha chiamato la funzione $\textsc{Sign}(\mathtt{sk}, \cdot)$, la probabilità che $\sigma$ sia una firma valida per $m$ rispetto alla chiave pubblica $\mathtt{pk}$ è *negligible* in $\lambda$ (una funzione $f(\lambda)$ è *negligible* se va a zero più velocemente di qualunque polinomio al crescere di $\lambda$).

Per un background teorico sugli schemi di firme digitali si veda, per esempio, il Capitolo 12 in [2].

## 2.1 Esempio: RSA

Nello schema di firma digitale RSA [3]

- $\textsc{KeyGen}(1^\lambda)$
  - Genera due numeri primi $p$ e $q$ dell'ordine di $\lambda$ bit;
  - Calcola $n = pq$;
  - Genera un $e$ coprimo con $(p-1)(q-1)$;
  - Calcola l'inversa di $e$ modulo $(p-1)(q-1)$, ossia un $d$ tale che $de \equiv 1 \mod (p-1)(q-1)$;

  - Restituisce $\texttt{pk} = (n, e)$, $\texttt{sk} = (n, d)$.
- $\textsc{Sign}(\texttt{sk}, m)$
  - Calcola $h = \textsc{Hash}(m)$, dove $\textsc{Hash}$ è una funzione hash crittografica;
  - Restituisce $h^d \mod n$.
- $\textsc{Verify}(m, \texttt{pk}, \sigma)$
  - Calcola $h = \textsc{Hash}(m)$;
  - Se $\sigma^e \equiv h \mod n$ restituisce $\textsc{True}$ altrimenti restituisce $\textsc{False}$.

**Esercizio 6.** Dimostrare che lo schema RSA è *corretto*.

**Esercizio 7.** Il codice Python seguente[1]

**Algorithm 2** 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 a Principles of Cryptocurrency Design - AA 25/26!'
 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: 0x36ef68d9d5b6930e2cd9d2435ad171e4682828cb621b20daf836811bbca47f228539
db302a66bf567e12240bac78e0a6699c230854a83dddac392287212608c5597f04d05dce7c7a61
d7143bc96f644ecb9d732dbd01f86e0009f7a64946fe7e38634b0db41673e8354f5895b8b103d8
d5449155b1a6daef2bfc67f559e7315e
```

Scrivere un programma per verificare che si tratta di una firma valida del messaggio *Benvenuti a Principles of Cryptocurrency Design - AA 25/26!* rispetto alla chiave pubblica $\texttt{pk} = (n, e)$ dove

```
n = 0xb9922e17467d351c454bbbe1a83ee5e6da8c5e650857c1cf9b980bbfd295eadd9b2aa09a
97b88d409820b7a6380a4c952b617b755a2eaef6b2aced5f8914610baf343edb9e5be3cb15d11c
5f368427becd5969a8d1ab9d527ba5edad5d870600ea2b447cdfd45f227cba7076ede23087c4e7
c581e6693f0a67531250fa69d205
```

```
e = 0x10001
```

**Esercizio 8.** Supponiamo che, invece di firmare l'*hash* del messaggio (linee 7 e 8 in Algorithm 2), avessimo firmato direttamente il messaggio:

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

**Algorithm 3** Textbook RSA

```
1. from Crypto.PublicKey import RSA
2.
3. keyPair = RSA.generate(bits = 1024)
4.
5. msg = b'Benvenuti a Principles of Cryptocurrency Design - AA 25/26!'
6. msg_sig = pow(int.from_bytes(msg, byteorder='big'), keyPair.d, keyPair.n)
7.
8. print("Firma: ", hex(msg_sig))
```

1. Scrivere un programma per verificare che la firma seguente, generata con lo schema in Algorithm 3,

   `Firma: 0x96770d30c7f5370a6ffc1472e3638f8c58a86eae91aa675a17a9b2c924ebdaae fdd3ae56417b2ec0c88157427ed4dba55c3f926708c27d561ef4a06edd280061912d6f25b9 8e1582195da77547f5834b9b424541e6f6ca13313e0d1571cecb29e8d7518f9d3ce1586535 8b8c6b4b9947aa370f9140d25f3fd387457c3e5fc01a`

   è una firma valida del messaggio *Benvenuti a Principles of Cryptocurrency Design - AA 25/26!* rispetto alla stessa chiave pubblica dell'esercizio precedente.

2. Mostrare che lo schema di firma digitale in Algorithm 3 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.[2]

3. Notare il motivo per cui l'attacco del punto precedente invece non è attuabile sullo schema di firma digitale in Algorithm 2.

## Riferimenti bibliografici


[1] Danny Dolev and H. Raymond Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983. `https://www.cs.huji.ac.il/~dolev/pubs/authenticated.pdf`.

[2] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and hall/CRC, 2007.

[3] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. `https://dl.acm.org/doi/pdf/10.1145/359340.359342`.

[4] Elaine Shi. Foundations of distributed consensus and blockchains. Available at `https://www.distributedconsensus.net`, 2020.


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