UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'Automazione

Progettazione e implementazione di protocolli blockchain per l'identificazione biometrica

Design and implementation of blockchain protocols for biometric identification

Candidato:
Vito Scaraggi

Relatore:
Dott. Paolo Santini

Correlatori:
Prof. Marco Baldi
Dott. Massimo Battaglioni

Anno Accademico 2021-2022

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'Automazione
Via Brecce Bianche – 60131 Ancona (AN), Italy

Ringraziamenti

Inizierei col ringraziare chi ha contribuito materialmente allo sviluppo di questa tesi: il relatore Paolo Santini per l'intenso scambio via e-mail e la forte motivazione che mi ha trasmesso in questo lavoro; i correlatori Massimo Battaglioni e Marco Baldi per le preziose correzioni all'elaborato; mamma e papà per avermi aiutato nel concludere la raccolta dati, permettendomi di accedere da remoto al mio computer principale.
Mamma e papà, da voi prendo in prestito la perseveranza e l'umiltà, due valori fondamentali su cui posso dire di aver costruito, un mattone alla volta, gran parte di quello che posso raccontare. Grazie per il sostegno fatto di cibi pronti (e conseguenti mal di schiena), di parole di conforto, di calorosi abbracci ai miei ritorni e di consigli affettuosi.
Sorellina, ti ringrazio sia perchè me l'hai chiesto espressamente tu :) sia per i *watch party*, i frequenti "come stai?" e l'atmosfera scherzosa delle nostre conversazioni. Ti dedico questo traguardo perchè sia di ispirazione per i tuoi sogni e le tue ambizioni.
Ringrazio mio cugino Luca per le chiacchierate sui nostri interessi comuni; mia zia Giovanna che mi ha reso ricco con la paghetta post-esame; i nonni che mi dicono da sempre sorridenti "*Ad Maiora*"; i nonni premurosi che mi telefonano puntualmente dopo esami e trasferte; i miei parenti tutti, dai più vicini ai più lontani, per la costante generosità e le attenzioni nei miei confronti.
Ringrazio il gruppo di "amici dell'ombrellone" di Montesilvano, diventati poi "amici di tutti i giorni" per i bei momenti di festa e di sincera compagnia trascorsi insieme quando tornavo da Ancona.
Ringrazio i miei ex-coinquilini, Davide, Marco e Alessandro, per essere stati, nel modo meno convenzionale possibile, "casa" negli ultimi tre anni e aver portato tra le mura domestiche un che di vita universitaria quando fuori mancava.
Ringrazio i compagni di progetti e di corso, gli amici delle serate anconetane e i mille volti di questo tempo così veloce: magari avremmo potuto condividere qualcosa in più, stringere rapporti più autentici, ma devo dire che in questo la pandemia globale non ci ha reso le cose facili.
Non dimenticherò gli alti e i bassi che hanno caratterizzato questo percorso, le ansie, le gioie e quanto mi ha condotto fino a questo giorno: forte di queste esperienze, guardo con fiducia ai prossimi passi.

Ancona, Ottobre 2022

Vito Scaraggi

Sommario

Le tecnologie blockchain ricevono un successo sempre maggiore grazie a proprietà intrinseche che le rendono adatte a numerose applicazioni: dalle criptovalute alla tracciabilità dei prodotti, dal trattamento dei dati sanitari all'esercizio del voto elettorale. D'altra parte, progrediscono i metodi e le tecniche utilizzati per l'identificazione supportata da dati biometrici, ad esempio, impronte digitali e scansione del volto, che sono per natura *fuzzy*, ovvero caratterizzati da rumore di misura. La presente tesi tratta di due protocolli di firma digitale RSA e ECDSA integrati all'interno di una blockchain che adotta una *Proof of Work ad hoc* per il riconoscimento biometrico degli utenti. Lo studio prosegue con la progettazione di un'applicazione *standalone multiprocessing* in Python3 che simula il funzionamento della blockchain e implementa un'interfaccia grafica comprensiva di statistiche e grafici relativi alle performance.

Abstract

Blockchain technologies are more and more successful thanks to intrinsic properties that make them suitable for a lot of applications: from cryptocurrencies to product traceability, from healthcare data management to election vote. On the other hand, methods and tecnhiques used for identification with biometric data are progressing, such as digital fingerprints and face scan, which are fuzzy by nature, i.e., featured by measumerent noise. This thesis deals with two RSA- and ECDSA-based digital signature protocols inserted into a blockchain that adopts an ad-hoc Proof of Work for user biometric recognition. Study goes on with design of a standalone multiprocessing Python3 program which simulates blockchain functioning and implements a graphic interface including statistics and plots about performances.

Indice

Elenco delle figure

Capitolo 1

Introduzione

La quasi totalità delle applicazioni informatiche implementa un meccanismo di autenticazione dell'utenza. Attraverso il processo di autenticazione, l'utente è chiamato a dichiarare e confermare la propria identità per poter accedere ai servizi richiesti. Il sistema più comune utilizza due stringhe, l'username (pubblico) e la password (privata) per riconoscere gli utenti, e ha almeno due importanti requisiti: l'utente deve ricordare una password; l'ente verificatore deve memorizzarla in un luogo sicuro e mantenerla segreta. Le specifiche di sicurezza di questo protocollo spesso non sono ritenute sufficienti per applicazioni critiche (bancarie, sanitarie, ecc..).
In questi casi è sempre più largamente adottata la *Multi-factor Authentication* (MFA), approccio più robusto basato sull'utilizzo di almeno due dei seguenti tre fattori[1]:

- il *Knowledge factor* corrisponde a "un elemento che l'utente conosce": per esempio una password o un PIN;
- il *Possession factor* coincide con "un elemento che l'utente ha": per esempio una OneTimePassword (OTP) o una chiave hardware;
- l'*Inherence factor* è "un elemento che l'utente è", ovvero un dato che si considera inseparabile dall'identità del soggetto: di questa categoria fanno parte tipicamente i dati biometrici, che quantificano caratteristiche fisiche umane.

L'*Inherence factor* non è un dato immune a falsificazione e sottrazione ad opera di potenziali cyberattaccanti, tuttavia ha il vantaggio di non dover essere memorizzato da parte dell'utente, a cui basta scansionare un dito della mano o l'iride.
Un sistema di autenticazione basato sul fattore di inerenza deve tenere conto della *fuzzyness* dei dati biometrici, cioè della presenza di rumore nelle misurazioni. Infatti, sperimentalmente, due rilevamenti biometrici relativi alla stessa caratteristica fisica del medesimo utente sono sempre diversi.
Come è possibile autenticare un utente mediante un fattore di riconoscimento che può rivelarsi estremamente variabile? Una soluzione al problema è presentata in questo elaborato, in cui si tratta l'analisi e l'implementazione di due protocolli di autenticazione (RSA) e identificazione (ECDSA), basati su dati biometrici, che mirano a sfruttare la capacità di calcolo della blockchain.

1.1 Contributo

I contenuti e gli approfondimenti presentati in questa tesi si inseriscono nel contesto definito nel paper "*Working to Clear Fuzzy Signatures: Blockchain Protocols for Biometric Identification and Authentication*" [2] del Dipartimento di Ingegneria dell'Informazione UNIVPM. Nell'articolo vengono formalizzati due protocolli blockchain per l'identificazione biometrica, fondati sul concetto di *clearing*, ovvero "pulizia", delle firme digitali *fuzzy*. I protocolli utilizzano due noti algoritmi crittografici, RSA e ECDSA, e integrano al loro interno alcune misure che garantiscono la sicurezza della rete blockchain anche in presenza di *evil miners*, cioè nodi malevoli.
L'elaborato si pone in continuità con l'articolo, in primo luogo, ribadendo la struttura e le proprietà dei protocolli di consenso teorizzati, e in seguito, descrivendo un'implementazione software in linguaggio Python. La trattazione teorica segue, a grandi linee, quella presentata nel *paper*: dalla definizione di schema di firma *fuzzy* si giunge alle specializzazione nelle varianti RSA e ECDSA; l'individuazione di una vulnerabilità nella rete blockchain spinge a un'importante rettifica dei protocolli proposti; la valutazione dei tempi di *mining* di un blocco al variare del numero di transazioni porta all'introduzione di una *Proof of Work* aggiuntiva in stile Bitcoin.
Il software, *standalone* e dotato di interfaccia grafica, è stato progettato per fare uso del *multiprocessing*, in modo da riprodurre l'attività in parallelo dei *miners*, e per fornire statistiche e grafici relativi a funzionamento e prestazioni dei protocolli. I blocchi creati dalla simulazione possono essere consultati dall'*user* tramite un *explorer*. Le statistiche riguardano il tempo di creazione dei blocchi e i tentativi impiegati dai *miners* nella *Proof of Work* e sono distinte per numero di transazioni per blocco, numero di *fair* e *evil miners*, fase di *clearing* e *classical PoW*.
La codifica in linguaggio Python si rifà al paradigma della Programmazione Orientata agli Oggetti e ad alcuni noti *Design Pattern*.
I test possono essere eseguiti con varie configurazioni, modificando parametri come ad esempio il tipo di protocollo, il numero di blocchi e il numero di *miners*. Tra le impostazioni figura anche il tipo di distribuzione dei dati biometrici che può essere settata come "uniforme" o "binomiale".
L'analisi dei risultati ottenuti con il simulatore ha evidenziato alcune differenze rispetto al modello teorico. Questi scostamenti dipendono principalmente dalle assunzioni effettuate in fase di modellazione: i test hanno individuato quali ipotesi sono troppo stringenti e su quali aspetti porre maggiore attenzione.

1.2 Struttura della tesi

La tesi è organizzata come segue. Nel secondo capitolo, si analizza la natura *fuzzy* dei dati biometrici e ne viene descritta la distribuzione tramite alcune assunzioni; inoltre si presenta lo stato dell'arte nell'ambito dell'identificazione biometrica. Nel

terzo capitolo si definisce cos'è uno schema di firma digitale *non-fuzzy* e *fuzzy*, specializzando il concetto nelle varianti basate sugli algoritmi crittografici RSA e ECDSA. Nel quarto capitolo, si definisce cosa sono una blockchain e la *Byzantine Fault Tolerance*; si individua un problema di sicurezza presente nei protocolli discussi da risolvere con l'uso di uno *Pseudo-Number Generator*; si introduce una *Proof of Work* classica aggiuntiva per bilanciare il tempo di creazione dei blocchi. Il quinto capitolo verte sull'implementazione in Python del simulatore blockchain, a partire dalla formulazione dei requisiti progettuali fino alla spiegazione di alcuni *snippet* di codice. Il sesto capitolo include un esempio d'uso del simulatore e osservazioni sui dati ricavati da numerosi esperimenti.

Capitolo 2

Dati biometrici

Con il termine dato biometrico si intende qualsiasi caratteristica fisica o comportamentale misurabile che abbia le proprietà di essere universale, identificante e sufficientemente invariante nel tempo: i dati biometrici devono essere caratteristiche comuni a tutti gli individui e tali da poter essere associate in maniera univoca alla persona che le presenta.
Sono esempi di dato biometrico fisiologico le impronte digitali, il colore e la dimensione dell'iride, la fisionomia del volto e la sagoma della mano. Alcuni sistemi biometrici sono in grado di analizzare anche caratteristiche comportamentali quali l'impronta vocale, lo stile di scrittura grafica e i movimenti del corpo.
Tutti i dati biometrici sono caratterizzati da rumore di misura e per questo motivo sono chiamati *fuzzy*, letteralmente "sfocati". Nel caso specifico delle impronte digitali, le misurazioni possono variare notevolmente in base a posizionamento del dito e alla sensibilità dello scanner. Sebbene il risultato della scansione di un'impronta digitale sembri imprevedibile, è possibile fare delle assunzioni basate su osservazioni empiriche che evidenziano la somiglianza tra dati biometrici appartenenti alla stessa persona.
Un sistema biometrico ha la funzione di *pattern recognizer* [3] dei dati biometrici e, tipicamente, deve garantire l'implementazione di alcuni requisiti specifici: il riconoscimento degli utenti dev'essere accurato e veloce indipendentemente dal rumore biometrico e da altri fattori ambientali che possono influire sulla misura; le tecnologie utilizzate non devono essere ritenute invasive dai soggetti sottoposti a identificazione; il sistema dovrebbe essere particolarmente difficile da raggirare con azioni fraudolente.
Prima di ottenere l'accesso ad un servizio, agli utenti è richiesto tipicamente di registrarsi. Anche nei sistemi biometrici è prevista una fase di *enrollment*. A seguito della registrazione, l'utente prova ad accreditarsi nel sistema secondo una delle seguenti modalità:

- Autenticazione: la validazione dell'identità di un utente avviene comparando il dato biometrico in input con il dato presente nel database che corrisponde al medesimo utente. Questa modalità presuppone che ogni utente reclami la propria identità tramite un identificativo pubblico, per esempio un username o un indirizzo di posta elettronica.

- Identificazione: il dato biometrico in input viene comparato con tutti i dati presenti nel database finchè non viene trovato un *match*. In questa modalità non è necessario alcun dato aggiuntivo da collegare univocamente all'utente.

2.1 Distribuzione fuzzy

In questa sezione si effettuano delle ipotesi sulla distribuzione delle chiavi segrete fuzzy, cioè dei dati biometrici. Si supponga che ogni chiave segreta sia un intero positivo x e per ogni utente sia fissato un valore di riferimento $\overline{x}$. Si denoti con $P(\cdot)$ la probabilità di un evento e sia w un intero positivo sufficientemente piccolo uguale per tutti gli utenti. Si assume che:

- se x è una chiave segreta e $\overline{x}$ il corrispondente valore di riferimento allora:

$$P(|x - \overline{x}| \leq w) \approx 1 \tag{2.1}$$

- se x_1 e x_2 sono due chiavi segrete relative a utenti diversi allora:

$$P(|x_1 - x_2| \gg w) \approx 1 \tag{2.2}$$

La (2.1) esprime la "somiglianza" tra due chiavi segrete relative allo stesso utente; la (2.2) indica la "lontananza" tra due chiavi segrete relative a utenti diversi e stabilisce che le distribuzioni di chiavi segrete relative a utenti diversi, con ogni probabilità, non si sovrappongono.
Nella presente trattazione, misurare il dato biometrico corrisponde a estrarre una chiave segreta da una distribuzione di probabilità discreta D. Una distribuzione di probabilità discreta $\phi(x)$ è una distribuzione di probabilità definita su un insieme discreto K. Nei protocolli trattati, si precisa che:

$$K = \{x \,|\, x \equiv \overline{x} + e \mod n \,, e \in [-w, w]\} \tag{2.3}$$

con n parametro pubblico.
La distribuzione di e sull'insieme $[-w, w]$ non è nota a priori ed è particolarmente difficile stabilire se coincida con una distribuzione di probabilità classica. Infatti il rumore biometrico dipende fortemente dalla sensibilità del sensore di misura nonché dall'interazione utente - dispositivo. Per questo motivo, l'articolo [2] considera una distribuzione discreta uniforme, semplificando la trattazione senza perdere di generalità, in quanto le caratteristiche principali del protocollo non dipendono dalla distribuzione.
La figura 2.1 rappresenta una possibile distribuzione delle chiavi segrete *fuzzy* per due utenti distinti: le curve sono distribuzioni normali discrete centrate nel valore di riferimento $\overline{x_i}$ che assumono valore per $x \in [\overline{x_i} - w, \overline{x_i} + w]$. Per le assunzioni fatte, le due distribuzioni dovrebbero essere molto più distanti sull'asse delle ascisse

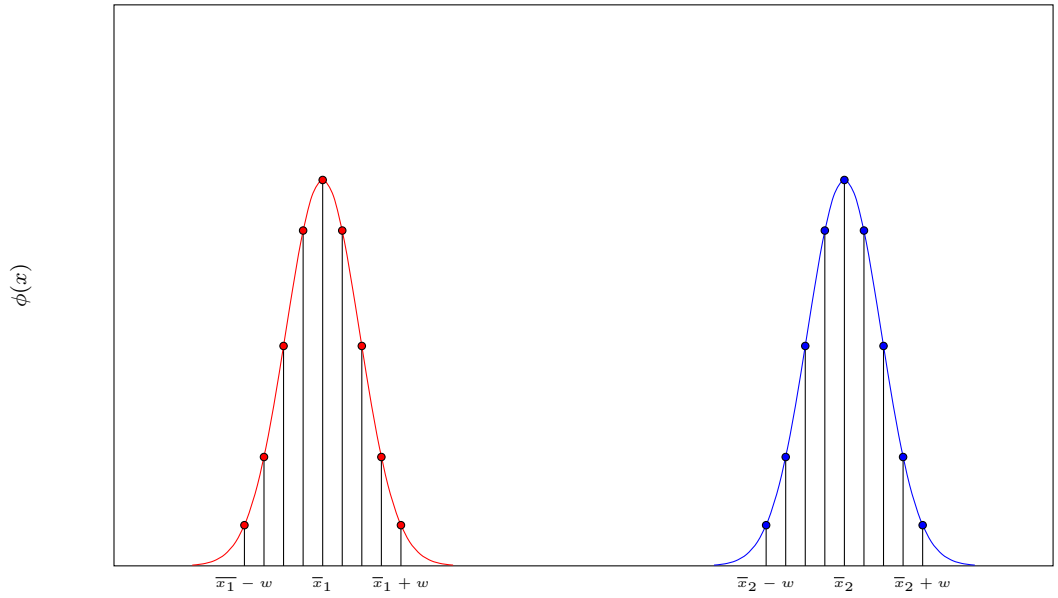


Figura 2.1: Esempio di distribuzioni dei dati biometrici

ma per esigenze rappresentative compaiono vicine. Nel grafico è raffigurato anche l'andamento continuo delle distribuzioni, ma si ricorda che le chiavi segrete hanno una distribuzione discreta corrispondente ai punti evidenziati.

2.2 Stato dell'arte

Oltre a quello proposto dal *paper* [2], esistono numerosi algoritmi di validazione dei dati biometrici che, tuttavia, utilizzano un approccio molto diverso: innanzitutto la maggior parte di questi sistemi ha natura centralizzata. Visto che l'ente verificatore è unico, a differenza dei due protocolli proposti in cui si applica il consenso distribuito, gli obiettivi primari di tali algoritmi sono la velocità e l'efficienza. Al contrario, nell'architettura blockchain suggerita l'operazione di verifica è un *task* computazionalmente difficile sottoposto a un gran numero di solutori o *miners*.
Alcuni degli algoritmi preesistenti si basano sul *pattern recognition*, termine che sottende l'estrazione e l'analisi delle *macro* e *micro features* rilevabili nelle immagini di dati biometrici: appartengono a questa categoria il *Pattern based Fingerprint Recognition Method* e il *Minutiae based Fingerprint Algorithm* sviluppati per le impronte digitali [4]. L'*image processing* può essere supportato anche da tecniche di *Machine Learning* e *Deep Learning*. Come si può notare, il presente lavoro non si focalizza sulle caratteristiche proprie di uno specifico dato biometrico, ma parte dall'ipotesi che ogni dato possa essere convertito in un numero intero.
Per quanto riguarda le *fuzzy signature*, in altri studi [5] si presenta una definizione formale di schema di firma digitale *fuzzy* insieme a istanze concrete di protocolli, tuttavia sono analizzate con maggiore dettaglio le qualità degli schemi di firma digi-

tale, quali proprietà omomorfiche e lo strumento *linear sketch*. La ricerca congiunta su *fuzzy signature* e blockchain ha condotto a soluzioni innovative nell'ambito del *biometric recognition*: in [6] si adottano gli *Smart Contracts*, frammenti di codice integrati nella struttura dei blocchi, per gestire l'autenticazione degli utenti; [7] propone un meccanismo con un *single use biometric token* basato sul *Shamir's Secret Sharing Algorithm* e sulla memorizzazione del *biometric template* all'interno di una blockchain.
In generale l'accostamento di *fuzzy signature* e blockchain si rivela vincente: da un lato si fornisce un algoritmo per l'identificazione univoca degli utenti; dall'altro si soddisfano le proprietà di sicurezza, integrità e non ripudiabilità [1] che un tale meccanismo dovrebbe possedere.

[1] Il non ripudio si riferisce alla condizione in cui l'autore di un'azione non può negare la validità e la paternità dell'azione stessa

Capitolo 3

Firma digitale fuzzy RSA e ECDSA

Per descrivere più nel dettaglio il funzionamento di un sistema di identificazione biometrica, occorre introdurre la definizione di firma digitale. Uno schema di firma digitale è una primitiva crittografica asimmetrica costituita dalle seguenti componenti algoritmiche:

- Generazione delle chiavi: a partire dal dato biometrico vengono generate una chiave privata e una chiave pubblica. La chiave pubblica viene comunicata al verificatore.
- Firma: l'utente utilizza la propria chiave privata per firmare il messaggio inviato dal verificatore. La firma viene inviata al verificatore.
- Verifica: il verificatore utilizza la chiave pubblica e il messaggio originario per stabilire se la firma è valida.

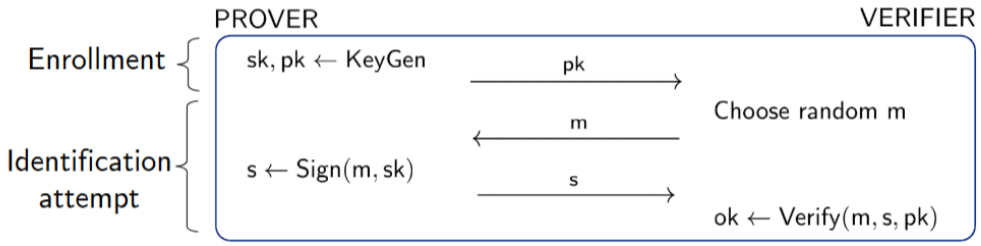


Figura 3.1: Schema di identificazione basato su firma digitale

Le tre fasi sono ripercorse nella figura 3.1: la generazione delle chiavi ha luogo durante la registrazione dell'utente (*enrollment*); la firma e la verifica sono necessarie ad ogni tentativo di identificazione.
Uno schema di firma digitale con queste specifiche non è ancora pronto per funzionare congiuntamente ai dati biometrici: infatti, come descritto, l'utente firma sempre utilizzando la chiave segreta definita nella fase di enrollment. Questa condizione è in conflitto con la *fuzzyness* dei dati biometrici: ad ogni misurazione la chiave segreta cambia, quindi la firma generata, con probabilità molto alta, non supererà la verifica con la chiave pubblica definita nella fase di enrollment.

Le misurazioni effettuate nei tentativi di identificazione producono una chiave segreta che, in base alle assunzioni sulla distribuzione dei dati biometrici, è opportunamente vicina a quella rilevata nella fase di *enrollment*. Si ipotizza che il rumore della nuova chiave segreta si propaghi anche nella firma e nella chiave pubblica, ovvero la nuova firma e la nuova chiave pubblica siano opportunamente vicine a quella rilevate nella fase di *enrollment*. Allora, una possibile soluzione al problema della *fuzzyness* consiste nell'assegnare al verificatore il *task* di "pulire" la firma o la chiave pubblica, cioè rimuoverne il rumore. Il *clearing task* è un problema relativamente difficile da un punto di vista computazionale: da qui l'idea di impiegarlo come *Proof of Work* per una blockchain.

3.1 Firma digitale fuzzy

Come definito nel capitolo 2.1, K è lo spazio delle chiavi, ovvero l'insieme di tutte le possibili chiavi segrete. Siano $s_k \in K$ la chiave segreta e D una distribuzione discreta definita su K. Si definisce schema di firma digitale *fuzzy*, uno schema di firma digitale con le seguenti operazioni:

- KeyGen$(D) \rightarrow s_k, p_k$: estrae in maniera casuale s_k da D poi calcola la relativa chiave pubblica p_k;
- Sign$(m,\ D) \rightarrow \sigma$: estrae in maniera casuale s'_k da D poi firma il messaggio m con s_k;
- Verify$(m,\sigma,p_k) \rightarrow$ True | False: dati il messaggio m e la chiave pubblica p_k, stabilisce se la firma σ è valida.

Uno schema di firma digitale *fuzzy* differisce da un normale schema di firma digitale perché le chiavi segrete utilizzate nella fase di generazione delle chiavi e nella fase di identificazione sono in generale diverse perciò, con molta probabilità, la firma sarà ritenuta invalida. Se si firma un messaggio m con una chiave segreta $s'_k = s_k + e$, dove $|e| \leq w$, producendo in output una firma σ', è chiaro che quest'ultima non verrà verificata da p_k. Per questo motivo è necessario anteporre alla fase di verifica un processo di *clearing* (pulizia). Il *clearing* può essere effettuato sia sulla firma che sulla chiave pubblica.

Definizione 3.1.1 (*Signature clearing*)**.** Sia $dist(a, b)$ una funzione di distanza tra due firme a e b e θ un intero positivo sufficientemente piccolo, allora si definisce *signature clearing* un algoritmo che, dati in input il messaggio m, la firma σ e la chiave pubblica p_k, computi una firma σ' tale che $dist(\sigma, \sigma') \leq \theta$ e Verify$(m,\ \sigma',\ p_k)$ = True.

La *signature clearing* richiede che le firme σ e σ' siano opportunamente vicine: questa condizione è rispettata se si considera che la vicinanza tra le chiavi segrete si propaghi anche nelle relative firme, cioè:

$$s_k \approx s'_k \Rightarrow \sigma \approx \sigma' \tag{3.1}$$

Definizione 3.1.2 (*Public key clearing*)**.** Sia $dist(a, b)$ una funzione di distanza tra due chiavi pubbliche a e b e θ un intero positivo sufficientemente piccolo, allora si definisce *public key clearing* un algoritmo che dati in input il messaggio m, la firma σ e la chiave pubblica p_k computi una chiave pubblica p'_k tale che $dist(p_k, p'_k) \leq \theta$ e Verify(m, σ, p'_k) = True.

In maniera simile alla *signature clearing*, la *public key clearing* presuppone che la vicinanza tra le chiavi segrete si propaghi anche nelle corrispondenti chiavi pubbliche, cioè:

$$s_k \approx s'_k \Rightarrow p_k \approx p'_k \tag{3.2}$$

Analizziamo ora le implementazioni RSA e ECDSA del protocollo di firma digitale *fuzzy*.

3.2 Versione RSA

3.2.1 RSA

RSA è un algoritmo di crittografia asimmetrica, inventato nel 1977 da Ronald Rivest, Adi Shamir e Leonard Adleman[8]. RSA appartiene a una classe di algoritmi composti da due chiavi crittografiche: una chiave pubblica e una chiave privata, da mantenere segreta. Generalmente la chiave pubblica è utilizzata per cifrare, la chiave privata per decifrare. Nei sistemi di firma digitale avviene il contrario: la chiave privata serve a firmare il messaggio, la chiave pubblica a decifrare. La sicurezza degli algoritmi di crittografia asimmetrica, come RSA, risiede nell'impossibilità teorica di ricavare una chiave conoscendo l'altra, nonostante esista una dipendenza fra queste. In particolare RSA sfrutta la difficoltà insita nel problema di fattorizzare il prodotto di due numeri primi grandi. Risolvere l'*RSA problem* diventa più arduo al crescere di tale prodotto: la dimensione oggi ritenuta "sicura" del *public modulus* è 2048 o 4096 bit.

Non è tra gli scopi della tesi dimostrare la correttezza di RSA, dunque nel seguito si riportano soltanto alcune definizioni utili a comprendere lo schema di firma *fuzzy* derivato da RSA:

Definizione 3.2.1 (Funzione di Eulero)**.** La funzione di Eulero o toziente ϕ associa a ogni intero n il numero di interi compresi tra 1 e n che sono coprimi con n. In simboli:

$$\phi(n) = \#\{x \mid x \leq n \wedge gcd(x, n) = 1, x \in \mathbb{N}\} \tag{3.3}$$

dove # è la cardinalità dell'insieme, *gcd* è il massimo comun divisore.

Definizione 3.2.2 (Inverso moltiplicativo modulo n)**.** L'inverso moltiplicativo modulo n di un intero x è l'intero y tale per cui si ha $x \cdot y \equiv 1 \mod n$. Per indicarlo si usa la notazione $y \equiv x^{-1} \mod n$. Una formula per calcolarlo deriva dal teorema di Eulero-Fermat:

$$x^{\phi(n)} \equiv 1 \mod n \tag{3.4}$$

da cui si ottiene:

$$y \equiv x^{\phi(n)-1} \mod n \tag{3.5}$$

3.2.2 Schema di firma digitale fuzzy RSA

Keygen

Nella fase di generazione delle chiavi, all'utente vengono associati due numeri primi grandi p e q, ponendo $n = pq$. n è detto *public modulus*. Si calcola la funzione di Eulero in n cioè $\phi(n) = (p-1)(q-1)$.
L'utente estrae una chiave segreta x dalla distribuzione discreta D definita sull'insieme:

$$K = \{x | x \equiv \overline{x} + e \mod n, e \in [-w, w]\} \tag{3.6}$$

dove $\overline{x}$ è il valore di riferimento dell'utente, w un intero positivo pubblico sufficientemente piccolo. Se x è coprimo con $\phi(n)$ si pone $\widehat{x} = x$ altrimenti si cerca $\widehat{x} \in \{x \pm 1, x \pm 2, ...\}$ che sia coprimo con $\phi(n)$.
Ottenuto $\widehat{x}$ si computa l'esponente pubblico $\delta \equiv \widehat{x}^{-1} \mod n$. In conclusione, la chiave privata è $\{\widehat{x}\}$, la chiave pubblica è $\{n, \delta\}$.

Sign

L'utente firma un messaggio m con una chiave segreta x estratta dalla distribuzione D . Il messaggio viene hashato[2] e successivamente convertito in un numero intero. Posto $c = Hash(m)$, la firma dell'utente è data da $\sigma = c^x \mod n$.

Clearing e Verify

Noti la firma σ e la chiave pubblica $\{n, \delta\}$, si calcola $c' = \sigma^\delta \mod n$. In uno schema di firma digitale *non fuzzy* RSA, la firma è valida se risulta $c' \equiv c \mod n$: nel caso *fuzzy* è richiesta una fase preliminare di *signature clearing*.
Come precedentemente detto, $\widehat{x}$ è la chiave segreta utilizzata nella fase di *enrollment*, x è la chiave segreta utilizzata per firmare. In generale $\widehat{x} \neq x$ ed essendo $x, \widehat{x} \in K$ si

[2]Una funzione di hash è una funzione che, data una stringa di lunghezza arbitraria in ingresso, produce una sequenza univoca di bit di lunghezza fissa in uscita, detta *digest*. Ha la proprietà di essere irreversibile, cioè non è possibile ricavare l'input che genera un particolare *digest*.

può scrivere:

$$\begin{aligned} \widehat{x} &= \overline{x} + e \\ x &= \overline{x} + e' \end{aligned} \tag{3.7}$$

Calcoliamo le firme relative alle chiavi segrete x e $\widehat{x}$:

$$\begin{aligned} \widehat{\sigma} &= c^{\widehat{x}} = c^{\overline{x}+e} \\ \sigma &= c^{x} = c^{\overline{x}+e'} \end{aligned} \tag{3.8}$$

Dalle equazioni (3.8) si deriva che $\widehat{\sigma} = \sigma \cdot c^{e-e'}$. I termini σ e c sono noti; il termine $e - e'$ è incognito ma si ha che:

$$e \in [-w, w], e' \in [-w, w] \Rightarrow e - e' \in [-2w, 2w] \tag{3.9}$$

$\Delta e = e - e'$ è detta distanza tra le firme $\widehat{\sigma}$ e σ e $\theta = 2w$ (definizione 3.1.1) . È facile dimostrare come $\widehat{\sigma}$ superi la verifica. Infatti ricordando che $\widehat{x} \cdot \delta \equiv 1 \mod n$:

$$c'' = \widehat{\sigma}^{\delta} \equiv (\sigma \cdot c^{e-e'})^{\delta} \equiv (c^{\overline{x}+e'} \cdot c^{e-e'})^{\delta} \equiv (c^{\overline{x}+e})^{\delta} \equiv c^{\widehat{x}\delta} \equiv c \mod n \tag{3.10}$$

$c'' \equiv c \mod n$, quindi la verifica è superata.
Obiettivo del verificatore è proprio computare $\widehat{\sigma}$, cercando il *clearing value* Δe che "pulisce la firma". Quindi lo schema di firma digitale *fuzzy* RSA si arricchisce dell'algoritmo RSAClearSign:

RSAClearSign(m, σ, $\{n, \delta\}$)

1. calcola $c = Hash(m)$
2. calcola $c' \equiv \sigma^{\delta} \mod n$
3. estrae $\Delta e \in [-2w, 2w]$
4. calcola $c'' \equiv c' \cdot c^{\Delta e} \mod n$
5. restituisce Δe se $c'' = c$ altrimenti torna al punto 3

Lo schema di firma digitale *fuzzy* RSA richiede che ogni utente registri un username o un altro attributo identificativo pubblico: in questo caso si parla di protocollo di autenticazione perché l'identità dell'utente è nota. L'username e la chiave pubblica di ogni utente sono memorizzati all'interno di un file pubblico consultabile dai verificatori.

3.3 Versione ECDSA

3.3.1 ECDSA

ECDSA o *Elliptic Curve Digital Signature Algorithm*[9, 10] è un algoritmo di crittografia asimmetrica derivato da DSA (*Digital Signature Algorithm*). La sua robustezza si basa sull'intrattabilità dell'*Elliptic Curve Discrete Logarithm Problem* (ECDLP). Al fine di comprendere il significato di alcuni parametri presenti nello schema di firma digitale ECDSA *fuzzy*, introduciamo brevemente alcune nozioni utili.

Definizione 3.3.1 (Campo finito primo)**.** Sia q un numero primo. Il campo finito F_q chiamato *prime field* è l'insieme di interi $\{0, 1, .., q-1\}$ chiuso rispetto alle seguenti operazioni:

- Addizione: se $a, b \in F_q$ allora $c = (a + b) \mod q \in F_q$
- Moltiplicazione: se $a, b \in F_q$ allora $c = (a \cdot b) \mod q \in F_q$
- Inverso moltiplicativo: se $a \in F_q$, $a \neq 0$ allora $c = a^{-1} \mod q \in F_q$

Definizione 3.3.2 (Curva ellittica su F_q)**.** Sia q un numero primo maggiore di 3. Una curva ellittica E su F_q è definita dall'equazione:

$$y^2 = x^3 + ax + b \tag{3.11}$$

dove $a, b \in F_q$ e $4a^3 + 27b^2 \not\equiv 0 \mod q$. L'insieme $E(F_q)$ è dato da tutti i punti $(x, y) \in F_q \times F_q \cup O$ che soddisfano l'equazione (3.11). O è un punto speciale chiamato "punto all'infinito" tale che:

$$\forall Q \in E(F_q) \Rightarrow Q + O = Q \tag{3.12}$$

Per comprendere il significato dell'equazione (3.12) è necessario introdurre l'operazione di addizione nell'insieme $E(F_q)$ [9].

Definizione 3.3.3 (Addizione in $E(F_q)$)**.** Siano $P_1 = (x_1, y_1) \in E(F_q)$ e $P_2 = (x_2, y_2) \in E(F_q)$ tali che $P_1 \neq \pm P_2$. Allora $P_1 + P_2 = (x_3, y_3)$ dove:

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \quad , \quad y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1 \tag{3.13}$$

L'operazione di duplicazione nell'insieme $E(F_q)$ ha la seguente definizione.

Definizione 3.3.4 (Duplicazione in $E(F_q)$)**.** Sia $P = (x_1, y_1) \in E(F_q)$ tale che $P \neq -P$. Allora $2P = (x_3, y_3)$ dove:

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \quad , \quad y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1 \tag{3.14}$$

Nello schema di firma digitale *fuzzy* ECDSA presentato viene utilizzata la curva ellittica $E : y^2 = x^3 + 7$. L'insieme $E(F_q)$ coincide graficamente con una una nuvola di punti.

Definizione 3.3.5 (Parametri di dominio ECDSA)**.** Sia E una curva ellittica e F_q un campo finito primo. Allora sono fissati i seguenti parametri di dominio pubblici:

- q è la cardinalità del campo finito primo F_q;
- n è la cardinalità dell'insieme $E(F_q)$ ed è detto ordine della curva ellittica E;
- il punto G tale che $E(F_q) = \{kG : 0 \leq k \leq n-1\}$ è detto punto generatore della curva ellittica E.

Considerato a titolo d'esempio il campo finito primo F_{11} e la curva ellittica $E : y^2 = x^3 + 7$, si ottiene la figura 3.2.

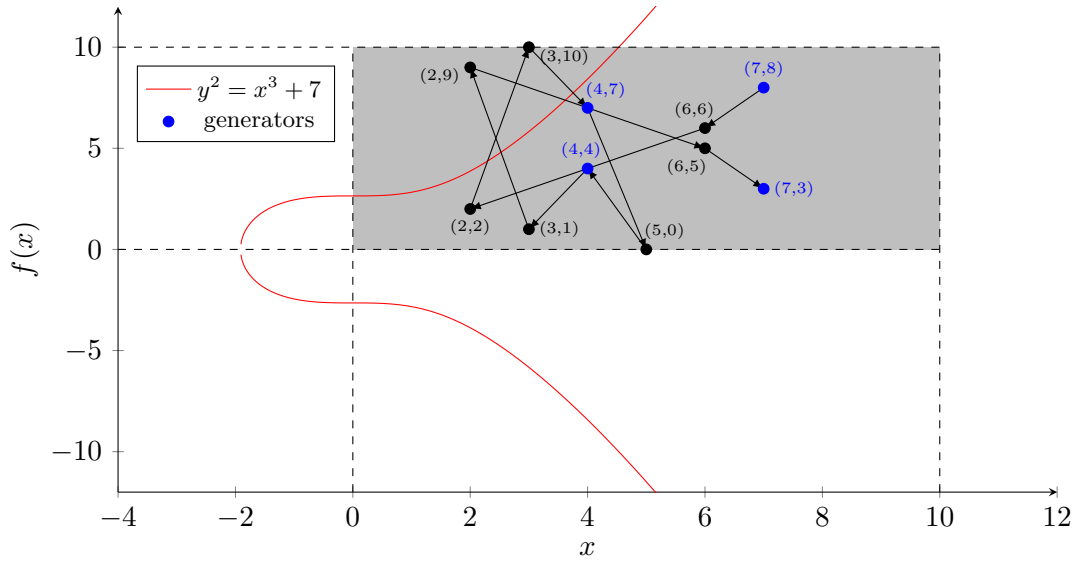


Figura 3.2: Curva ellittica su campo finito primo

Il punto $P = (7, 8)$ è un punto generatore della curva ellittica; le frecce orientate costruiscono il cammino $P \to 2P \to 3P \to ... \to 11P$. I punti evidenziati in blu sono punti generatori della curva ellittica. In questo caso specifico $n = q = 11$. La regione di spazio evidenziata corrisponde a $[0, q-1] \times [0, q-1]$.

Infine possiamo formalizzare l'*Elliptic Curve Discrete Logarithm Problem.*

Definizione 3.3.6 (ECDLP)**.** Si definisce ECDLP il seguente problema: data una curva ellittica E definita su un campo finito F_p, un punto $P \in E(F_q)$ di ordine n e un punto $Q = xP$ con $0 \leq x \leq n-1$, determinare x.

L'ECDLP è un problema computazionalmente intrattabile se q e n sono abbastanza grandi. Per il teorema di Hasse[9]:

$$n = q + 1 - t,\ |t| \leq 2\sqrt{q} \tag{3.15}$$

in altri termini si verifica $n \approx q$. L'incognita x rappresenta la chiave segreta in uno schema di firma digitale ECDSA. Anche in questo caso non si procede nel dimostrare la correttezza dell'algoritmo, ma si illustra direttamente lo schema di firma digitale *fuzzy* ECDSA.

3.3.2 Schema di firma digitale fuzzy ECDSA

Keygen

L'utente estrae una chiave segreta $\widehat{x}$ dalla distribuzione discreta D definita sull'insieme:

$$K = \{x | x \equiv \overline{x} + e \mod n, e \in [-w, w]\} \setminus \{0, 1\} \tag{3.16}$$

dove $\overline{x}$ è il valore di riferimento dell'utente, w un intero positivo pubblico sufficientemente piccolo e n l'ordine della curva ellittica E. Si calcola $Q = \widehat{x} \cdot G$, dove G è il punto generatore della curva ellittica $E : y^2 = x^3 + 7$. La chiave privata è $\{\widehat{x}\}$, la chiave pubblica è $\{Q\}$.

Sign

L'utente estrae una chiave segreta x dalla distribuzione D e computa $c = Hash(m)$. Calcola i parametri $r = (xG)_x \mod n$, dove $(xG)_x$ è l'ascissa del punto xG, $s = x^{-1} \cdot (c + rx) \mod n$ e $v = (xG)_y \mod 2$, dove $(xG)_y$ è l'ordinata del punto xG. Se $s = 0$, ricomputa r, s e v con un nuovo valore di x estratto da D.
Al termine del procedimento, la firma è data da $\{r, s, v\}$.

Clearing e Verify

Noti $c = Hash(m)$ e la firma $\{r, s, v\}$ si calcola il punto P la cui ascissa è r, trovando l'ordinata y dalle soluzioni dell'equazione $y = \pm\sqrt{x^3 + 7} \mod q$. Il parametro v serve a discernere tra le due soluzioni: se $v = 0$ si sceglie la soluzione pari, altrimenti la soluzione dispari. In seguito si computa $Q' = r^{-1}(sP - cG)$ dove r^{-1} è l'inverso moltiplicativo di r modulo n. In uno schema di firma digitale *non fuzzy* ECDSA, la firma è valida se risulta $Q' = Q$ dove Q è la chiave pubblica dell'utente: nel caso *fuzzy* è richiesta la fase preliminare di *public key clearing*.
Analogamente allo schema di firma digitale *fuzzy* RSA, la chiave segreta $\widehat{x} = \overline{x} + e'$, utilizzata nella fase di *enrollment*, e la chiave segreta $x = \overline{x} + e$, utilizzata per firmare,

sono in generale diverse. Le relative chiavi pubbliche sono:

$$\begin{aligned} Q &= \widehat{x}G = \overline{x} + eG = \overline{x}G + eG \\ Q' &= xG = \overline{x} + e'G = \overline{x}G + e'G \end{aligned} \tag{3.17}$$

da cui si deriva che $Q = Q' + (e - e')G$. In modo simile a RSA, $\Delta e = e - e'$ è detta distanza tra le chiavi pubbliche Q e Q' e $\theta = 2w$ (definizione 3.1.2).
Obiettivo del verificatore è proprio computare Q, trovando il *clearing value* che "pulisce" la chiave pubblica. Il valore di Q trovato viene cercato all'interno di un file pubblico L che raccoglie le chiavi pubbliche di tutti gli utenti, ma non specifica a quali utenti appartengano le chiavi. Questa modalità di funzionamento è detta identificazione perché non si conosce a priori l'identità dell'utente. In ultima analisi, allo schema di firma digitale *fuzzy* ECDSA si aggiunge l'algoritmo ECDSAClearPublicKey:

ECDSAClearPublicKey(m, $\{r, s, v\}$)

1. calcola $c = Hash(m)$
2. trova il punto P la cui ascissa è r
3. calcola $Q' = r^{-1}(sP - cG)$
4. estrae $\Delta e \in [-2w, 2w]$
5. calcola $Q = Q' + \Delta eG$
6. restituisce Δe se $Q \in L$ altrimenti torna al punto 4

Nel caso ECDSA, diversamente da RSA, non si conosce a priori l'identità dell'utente, cioè la chiave pubblica registrata nella fase di *enrollment*. Ad ogni tentativo di *clearing* occorre verificare se la chiave pubblica calcolata sia presente nel *public file*.

Dopo aver introdotto i protocolli di firma digitale *fuzzy* RSA e ECDSA, si procede a inserirli in un contesto blockchain.

Capitolo 4

Protocolli blockchain

Una blockchain [11] è una base di dati pubblica, distribuita e immutabile che tiene traccia di tutte le transazioni e gli eventi digitali condivisi da un numero variabile di nodi partecipanti e li organizza in una lista concatenata di blocchi. Appartiene alla categoria delle *Distributed Ledger Tecnologies* (DLT), insieme di infrastrutture e protocolli che implementano "un libro mastro condiviso". Intrinsicamente decentralizzata, questa tecnologia elimina la presenza di terze parti, consentendo a due nodi qualsiasi della rete di interagire senza mediatori. Il *trust principle* (principio di fiducia) è sostituito dal *distributed consensus* (consenso distribuito): la verifica delle transazioni spetta ai nodi della rete e non a un ente centralizzato.
Le reti blockchain possono essere così classificate:

- *Public Blockchain*: i partecipanti sono *untrusted* e il numero di nodi non è controllato. Rappresenta l'idea originale di blockchain, intesa come rete *peer-to-peer* accessibile da chiunque.
- *Permissioned Blockchain*: soltanto i nodi autorizzati dall'organizzazione possono partecipare.

La rete Bitcoin è un esempio di blockchain pubblica, presentata dall'autore dal nome fittizio *Satoshi Nakamoto* nell'articolo *"Bitcoin: A Peer-To-Peer Electronic Cash System"* [12]. Nel *paper*, il protocollo blockchain viene adottato per costruire un sistema di pagamento elettronico decentralizzato indipendente dalle autorità finanziarie. Il meccanismo illustrato prevede anche un incentivo per i nodi che partecipano attivamente alla rete, che, a seguire, si sarebbe evoluto nel concetto di *criptocurrency*.

Le proprietà di trasparenza, anonimato e sicurezza rendono la blockchain una tecnologia emergente anche in applicazioni diverse da *crypto-wallet* ed *exchange*: nella sanità pubblica può essere utilizzata per memorizzare i dati clinici dei pazienti, garantendo che non vengano modificati o consultati da persone non autorizzate; è già stata testata nel voto elettorale, dove potrebbe scongiurare eventuali azioni fraudolente; un recente impiego è rappresentato dagli *Smart contracts*, segmenti di codice inseriti in una blockchain, che vengono eseguiti al raggiungimento di un accordo tra le parti direttamente interessate e che potrebbero automatizzare la negoziazione e l'attuazione di contratti di lavoro, di affitto ecc....

Tornando all'accezione più comune del termine, una blockchain è letteralmente una sequenza o catena di blocchi. Ogni blocco aggiunto alla blockchain non può essere rimosso. Un blocco è una struttura dati composta da un *block header* e un *block body*. Il *block header* include un *timestamp*, alcune altre informazioni relative al blocco e il *parent hash*, ovvero l'hash del blocco che lo precede nella blockchain. Nel caso del primo blocco, chiamato *genesis block*, il *parent hash* assume un valore di default, generalmente 0.

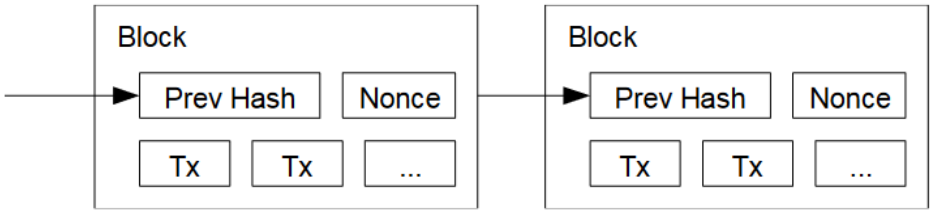


Figura 4.1: Blockchain secondo *Satoshi Nakamoto*

In definitiva ogni blocco è collegato al precendente attraverso il *parent hash*: una piccola variazione in un blocco produrrebbe una grande variazione nel suo hash e quindi, anche in tutti i blocchi successivi. Questa prioprietà della blockchain rende particolarmente difficile la modifica di un blocco da parte di un cyberattaccante.
Il *block body* contiene una lista di transazioni associate al blocco. Le informazioni contenute nell'*header* e il numero massimo di transazioni per blocco possono variare in funzione dello specifico protocollo blockchain considerato.
Per aggiungere nuovi blocchi alla blockchain, i nodi devono raggiungere il consenso, cioè essere d'accordo. In letteratura, esistono varie categorie di algoritmi di consenso [13], tra i quali:

- *Proof of Work* (PoW): per poter aggiungere un blocco alla blockchain è necessario risolvere un problema matematico o crittografico e sottoporre alla rete la relativa soluzione, chiamata valore di *nonce*. Se il blocco è corretto, i nodi lo aggiungono in coda alla propria copia locale della blockchain. La ricerca del valore di nonce richiede un notevole impiego di energia e CPU time: in virtù del grande sforzo computazionale, i nodi sono chiamati miners (minatori).

- Proof of Stake (PoS): è un'alternativa energy-saving alla PoW. Prevede che i nodi, chiamati verifiers, impegnino parte dei propri token (stake) nel processo di verifica delle transazioni. Il sistema predilige gli stakeholders più grandi, in quanto si ritiene che siano meno interessati ad attaccare la rete.

- *Delegated Proof of Stake* (DPoS): variante del PoS, stabilisce che i nodi eleggano altri nodi, chiamati delegati, per la creazione e la verifica dei blocchi. La

conferma delle transazioni è più rapida perchè i nodi verificatori sono in numero minore.

4.1 Complessità computazionale della PoW RSA e ECDSA

Nel capitolo precedente relativamente ai protocolli RSA e ECDSA, si è discusso del cosiddetto "obiettivo del verificatore" come di un compito da svolgere per il successo delle operazioni di riconoscimento biometrico. Ora noti i concetti e la terminologia delle blockchain, i "verificatori" non sono altro che i *miners* e il "compito" è, più nello specifico, una *Proof of Work*.
La complessità computazionale delle PoW RSA e ECDSA merita un'analisi dettagliata. In entrambe le versioni del protocollo, i *miners* estraggono randomicamente Δe dall'insieme $[-2w, 2w]$. Il Δe corretto è soltanto uno tra $4w + 1$ valori distinti. Supponendo che vi siano M miners il numero medio di tentativi per trovare il valore corretto è pari a:

$$\frac{4w + 1}{M} \tag{4.1}$$

Il numero di tentativi nella *clearing proof* decresce all'aumentare del numero di *miners*. Il *task* diventa triviale se $\frac{4w+1}{M} < 1$: in tal caso occorre rieffettuare il *tuning* dei parametri coinvolti nel *mining* delle transazioni per scalare la difficoltà. Il tempo totale impiegato nel *clearing* di una transazione è proporzionale al numero di tentativi effettuati. Nella versione ECDSA, al tempo per un tentativo di *clearing* va sommato il tempo necessario per cercare la chiave pubblica all'interno del *public file*. Supponendo che il file pubblico contenente le chiavi pubbliche sia ordinato, e ammesso di utilizzare il Binary Search Algorithm[3], la complessità computazionale di quest'operazione è $O(\log_2 N)$, dove N è il numero di utenti registrati. Nella versione ECDSA, dunque il tempo di *clearing* è proporzionale, oltre che al numero di tentativi, anche a $\log_2 N$.

4.2 Utenti malevoli

Fino a questo punto non è stata analizzata la sicurezza dei protocolli blockchain proposti. Nello specifico, si è implicitamente ipotizzato che tutti gli utenti della rete fossero onesti. In contesti reali è necessario tenere conto di potenziali nodi malevoli che potrebbero destabilizzare la rete blockchain. A questo scopo, le blockchain odierne prevedono meccanismi di incentivo e disincentivo che spingono gli utenti a comportarsi secondo le regole, in quanto più vantaggioso rispetto a infrangerle.
In teoria dei giochi, esiste un problema noto con il nome "*Byzantine Generals problem*" che risulta molto efficace nel descrivere la questione del consenso all'interno di una blockchain.

[3]Ricerca binaria (o dicotomica) è un algoritmo di ricerca per trovare un elemento in un *array* ordinato.

"*The Byzantine Generals problem*" [14] viene descritto nell'omonimo articolo di ricerca da Leslie Lamport, Robert Shostak e Marshall Pease nel 1982. Il problema è presentato attraverso un'analogia di tattica militare: alcune armate bizantine, ciascuna comandata da un generale, circondano una città nemica. I generali devono raggiungere un accordo sulla prossima mossa: attaccare o ritirarsi. Per comunicare, i generali hanno a disposizione dei messaggeri e possono essere leali, in tal caso decideranno per un piano comune, o traditori, in tal caso agiranno in maniera indipendente. La guerra ha successo soltanto se il piano dei generali leali viene seguito da tutti, quindi l'influenza dei generali traditori non è sufficiente a cambiare i piani.

A partire dal *Byzantine Generals problem*, si definisce *Byzantine Fault Tolerance*

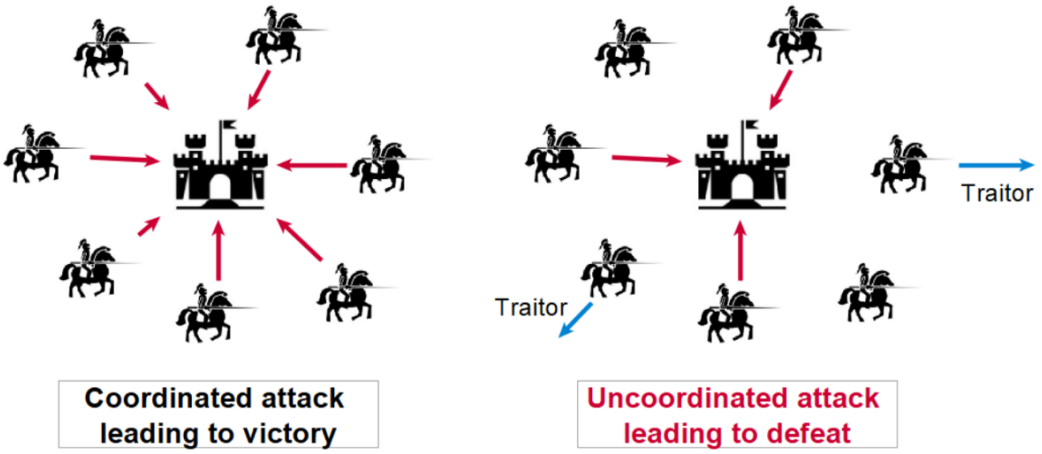


Figura 4.2: Byzantine Generals Problem

(BFT) la capacità di una rete informatica di raggiungere il consenso nonostante la presenza di nodi malevoli che potrebbero generare *faults* (errori). Nelle blockchain, BFT indica la percentuale di nodi "cattivi" o "bizantini" presenti sul totale, superata la quale i nodi "cattivi" producono blocchi più velocemente dei nodi "buoni". Generalmente BFT = 50%. Avviene un attacco del 50%+1 quando un *miner* o un gruppo di *miners* assumono il controllo della maggioranza assoluta della blockchain. In questo caso, vantando un *hashing rate* maggiore del resto dei nodi, questi potrebbero creare un $fork$ privato della blockchain a partire da un blocco già "minato" e riscrivere tutti i blocchi successivi, sostituendo infine la blockchain pubblica. Nei sistemi che utilizzano criptovalute, questa situazione può condurre a un *double-spending attack*, cioè gli *evil miners* sarebbero in grado di usare due o più volte le stesse monete virtuali inserendo nella blockchain dei blocchi alterati.

Negli schemi di firma RSA e ECDSA del capitolo precedente il processo di *clearing* corrisponde alla *Proof of Work* eseguita dai verificatori o *miners*. Rimuovendo l'ipotesi di utenti onesti, tuttavia, i *miners* potrebbero riuscire a produrre nuovi blocchi bypassando la fase di *clearing*. Sia u un utente colluso con un miner m:

- u memorizza la chiave segreta $x = \overline{x} + e'$ con cui ha firmato un messaggio
- la rete pulisce la firma (o la chiave pubblica) e produce in output $\Delta e = e - e'$
- u può ricavare la chiave segreta $\widehat{x} = x + \Delta e$ usata nella fase di *enrollment*
- nelle transazioni successive, u firma con una chiave segreta x' e conosce in anticipo il *clearing value* $\Delta e = \widehat{x} - x'$
- u comunica Δe a m
- m "mina" la transazione senza eseguire la *Proof of Work*

I miner, che conoscono in anticipo il *clearing value* Δe, possono produrre un nuovo blocco in tempo pressoché nullo. Nel seguito saranno chiamati *evil miners* i *miners* in possesso di quest'informazione e *fair miners* i restanti. Per ridurre il vantaggio computazionale degli *evil miners* è necessario modificare l'algoritmo di *clearing* negli schemi di firma digitale fuzzy RSA e ECDSA.

4.3 PRNG

Un *PseudoRandom Number Generator* (PRNG) è un algoritmo che genera numeri o sequenze di bit in modo pseudo-casuale. Un PRNG non è totalmente casuale perchè richiede che venga specificato un *seed*, ovvero un valore iniziale che rappresenta lo stato interno del generatore. Per gli schemi di firma *fuzzy* RSA e ECDSA, introduciamo due PRNG crittograficamente sicuri[4].

4.3.1 RSA PRNG

Con l'introduzione del PRNG, l'algoritmo RSAClearSign diventa il seguente:

RSAClearSign$^{(PRNG)}(m,\ aux,\ \sigma,\ \mathrm{p}_k = \{\mathrm{n},\delta\})$:

1. calcola $c = Hash(m)$
2. calcola $c' \equiv \sigma^{\delta} \mod n$
3. estrae *seed* in maniera casuale
4. calcola $\Delta e = \mathrm{RSAPRNG}(m.seed.aux)$
5. calcola $c'' \equiv c' \cdot c^{\Delta e} \mod n$
6. restituisce *seed* se $c'' = c$ altrimenti torna al punto 3

[4] Un PRNG crittograficamente sicuro (CSPRNG)[15] è un PRNG dotato di proprietà adatte all'uso in crittografia. In particolare, un CSPRNG soddisfa il *next-bit test*, cioè non è possibile predire in tempo polinomiale il $k + 1 - esimo$ bit di una sequenza random di k bit con accuratezza maggiore del 50%, e resiste alle *state compromise extensions*, cioè conoscere lo stato del PRNG nel presente non dà informazioni sugli output casuali generati in futuro.

dove *aux* è una stringa ausiliaria, *seed* è una stringa random e $m.seed.aux$ è la concatenazione delle stringhe m, *seed* e *aux*.
La stringa *aux* è scelta in modo tale che il suo valore non sia noto a priori del processo di *mining*: in particolare qui si pone $aux = parentHash$. Il suo scopo è quello prevenire gli attacchi basati su precomputazione: un utente malevolo potrebbe infatti preparare in anticipo una transazione da sottoporre alla blockchain, effettuarne il mining e comunicare la Proof of Work a un *evil miner*. Questa possibilità è sventata proprio dalla presenza della stringa *aux*: l'utente non ha modo di conoscerne il valore ed eseguire la precomputazione.

A questo punto non è ancora definita la funzione RSAPRNG, tuttavia si può dedurre che RSAPRNG prende in input una stringa random *seed* e restituisce un intero compreso nell'insieme discreto $[-2w, 2w]$, ovvero nel dominio di Δe. Inoltre il *golden value* della *Proof of Work* non è più uguale a Δe ma coincide con la stringa *seed*. Un *evil miner* a conoscenza di Δe deve comunque eseguire gli step 3, 4 e 6 della funzione RSAClearSign$^{(PRNG)}$. Per ridurre il vantaggio dei *evil miners* rispetto ai *fair miners* è opportuno modellare una funzione PRNG che abbia un costo computazionale sufficientemente elevato. In RSAPRNG $\widetilde{a}$, $\widetilde{b}$ e $\widetilde{n}$ sono parametri pubblici interi scelti casualmente in modo che $\widetilde{a}$, $\widetilde{b}$, $\widetilde{n} \approx 2^{bits}$ dove *bits* è la dimensione in bit del *public modulus*. X è il numero di iterazioni della funzione RSAPRNG.

RSAPRNG(*seed*):

1. calcola $seed = Hash(seed)$
2. per X volte:
   - converte $seed'$ in un intero in $[0, \widetilde{n} - 1]$
   - calcola $y = \widetilde{b} \cdot \widetilde{a}^{seed'} \mod \widetilde{n}$
   - pone $seed' = Hash(y)$
3. converte $seed'$ in un intero Δe in $[-2w, 2w]$
4. restituisce Δe

Indicando con t_{PRNG} il tempo di esecuzione dello step 4 della funzione RSAClearSign$^{(PRNG)}$ e con t_{CRYPTO} il tempo di esecuzione dello step 5, e trascurando il costo computazionale delle operazioni di hash e di conversione, si ha che $\frac{t_{PRNG}}{t_{CRYPTO}} = X$: infatti RSAPRNG ripete X volte lo step 4 di RSAClearSign$^{(PRNG)}$. Il parametro pubblico X può dunque essere settato per regolare il costo computazionale di RSAPRNG e, vedremo in seguito, anche la BFT. Nel testo per riferirsi all'esecuzione dello step 4 e dello step 5, si utilizzano rispettivamente i termini fase PRNG e fase CRYPTO.

4.3.2 ECDSA PRNG

Si riporta la funzione ECDSAClearPublicKey con PRNG:

ECDSAClearPublicKey$^{(PRNG)}$(m, $p_k = \{r, s, v\}$):

1. calcola $c = Hash(m)$
2. trova il punto P la cui ascissa è r
3. calcola $Q' = r^{-1}(sP - cG)$
4. estrae *seed* random
5. calcola Δe = ECDSAPRNG($m.seed.aux$)
6. calcola $Q = Q' + \Delta eG$
7. restituisce *seed* se $Q \in L$ altrimenti torna al punto 4

ECDSAPRNG(*seed*):

1. calcola $seed = Hash(seed)$
2. pone $\widetilde{Q} = \widetilde{a}G$
3. per X volte:
   - converte $seed'$ in un intero in $[0, n-1]$
   - calcola $Y = \widetilde{Q} + seed'G$
   - pone $seed' = Hash(Y)$
4. converte $seed'$ in un intero Δe in $[-2w, 2w]$
5. restituisce Δe

In ECDSAPRNG $\widetilde{a}$ è scelto casualmente in modo che $\widetilde{a} \approx n$ dove n è il parametro pubblico del protocollo ECDSA.
Un *miner* a conoscenza del *clearing value* Δe deve comunque eseguire gli step 4, 5 e 7 di ECDSAClearPublicKey$^{(PRNG)}$. Analogamente alla versione RSA, ECDSAPRNG è costruita in modo tale che $\frac{t_{PRNG}}{t_{CRYPTO}} = X$, dove t_{PRNG} è il tempo di esecuzione dello step 5 della funzione ECDSAClearPublicKey$^{(PRNG)}$, t_{CRYPTO} è il tempo di esecuzione dello step 6 della stessa funzione. Anche in questo caso si utilizzano rispettivamente i termini fase PRNG e fase CRYPTO.

4.3.3 BFT con PRNG

Come già anticipato, il parametro pubblico X ha un ruolo anche nel calcolo della BFT. Con le necessarie approssimazioni, il tempo necessario a un *fair miner* per effettuare un tentativo di *clearing* è dato da $t_{PRNG} + t_{CRYPTO}$. Per un *evil miner* tale tempo sarà uguale al solo termine t_{PRNG}, in quanto è già a conoscenza del *clearing value* Δe. Il tempo medio di mining di una transazione corrisponde a:

$$\frac{(4w+1)\cdot(t_{PRNG}+t_{CRYPTO})}{M-M_e} \tag{4.2}$$

per i *fair miners*, dove M è il numero di *miner* totali, M_e è il numero di *evil miners* e

$$\frac{(4w+1)\cdot t_{PRNG}}{M_e} \tag{4.3}$$

per gli *evil miners*. In entrambi i casi il tempo medio di *mining* di una transazione è dato dal prodotto del numero medio di tentativi e del tempo necessario per un tentativo.
Nel protocollo di consenso non è accettabile che gli *evil miners* impieghino meno tempo dei *fair miners* per "minare" una transazione, cioè dev'essere:

$$\begin{gathered}\frac{(4w+1)\cdot t_{PRNG}}{M_e} > \frac{(4w+1)\cdot(t_{PRNG}+t_{CRYPTO})}{M-M_e} \Rightarrow \\ \frac{M_e}{M} < \frac{1}{2+\frac{t_{CRYPTO}}{t_{PRNG}}}\end{gathered} \tag{4.4}$$

Dalla disequazione (4.4) si ricava che il rapporto tra *evil miners* e *miners* totali dev'essere strettamente minore di $\frac{1}{2+\frac{t_{CRYPTO}}{t_{PRNG}}}$. Questo valore corrisponde alla BFT. Da notare che se $t_{PRNG} \gg t_{CRYPTO}$ allora BFT $\approx 50\%$.

4.4 Proof of Work classica

Nei protocolli citati, il tempo di *mining* di un blocco è direttamente proporzionale al numero di transazioni che contiene. Un miner *greedy* [5] sceglierebbe senz'altro di minare i blocchi contenenti il minor numero di transazioni, accaparrandosi la *Proof of Work* "più facile". Il risultato desiderato è altresì che tutti i blocchi richiedano lo stesso sforzo computazionale: per questo motivo si impone ai *miners* di superare una *Proof of Work* finale con difficoltà parametrizzata in base al numero di transazioni presenti nel blocco. Adottiamo come *Proof of Work* finale una *HashCash Proof of*

[5] Il termine *greedy*, letteralmente "avido", si riferisce a una classe di algoritmi che seguono la strategia di effettuare la scelta locale ottima. In generale un algoritmo *greedy* non produce una soluzione globale ottima.

Work.
L'*HashCash Proof of Work* [16] venne proposta nel 1997 dal crittografo Adam Back. Utilizzata inizialmente come contromisura all'*email-spam* e agli attacchi *denial-of-service*, oggi è nota per essere la *Proof of Work* della blockchain Bitcoin. Detta anche una *Proof of Work* classica, presenta una forte asimmetria tra la complessità computazionale richiesta nella ricerca di una soluzione e nella verifica.
Il funzionamento è abbastanza semplice: il *miner* deve cercare un numero speciale chiamato *nonce* e aggiungerlo nell'*header* del blocco, poi computare l'hash di tutto il blocco. Se l'hash trovato inizia con un numero prestabilito di zeri, chiamato *difficulty*, allora la *Proof of Work* è superata, altrimenti si ripete il procedimento con un nuovo valore di *nonce*. L'utilizzo di una funzione crittografica di hash costringe i *miner* a procedere per tentativi. Se la *difficulty*, ovvero il numero di *leading zeros*, è k, la complessità computazionale della PoW è $O(2^k)$.
Per uniformare il tempo di *mining* di blocchi con un numero diverso di transazioni, bisogna intervenire proprio sul parametro k di *difficulty*. Il costo computazionale per "minare" un blocco può essere stimato in:

$$T = 2^k + (4w + 1) \cdot N_{tx} \tag{4.5}$$

dove N_{tx} è il numero di transazioni nel blocco. In questa equazione si fa uso dell'ipotesi semplificativa che un'operazione di hash e un tentativo di *clearing* siano computazionalmente equivalenti.
Se $\overline{N_t x}$ è il numero massimo di transazioni per blocco si può scegliere:

$$\begin{cases} k = 0 \quad \text{se} \quad N_{tx} = \overline{N_{tx}} \\ k = \text{round}(\, \log_2(\, (4w + 1) \cdot (\overline{N_{tx}} - N_{tx})\,)\,) \quad \text{altrimenti} \end{cases} \tag{4.6}$$

dove round(x) è l'approssimazione intera del numero reale x.
Per $k = 0$ e $N_{tx} = \overline{N_{tx}}$ si ha che:

$$\begin{aligned} T = 1 + (4w + 1) \cdot \overline{N_{tx}} \approx \\ (4w + 1) \cdot \overline{N_{tx}} \end{aligned} \tag{4.7}$$

Per $k = \text{round}(\, \log_2(\, (4w + 1) \cdot (\overline{N_{tx}} - N_{tx})\,)\,)$ e trascurando l'approssimazione intera si ha che:

$$\begin{aligned} T \approx (4w + 1) \cdot (\overline{N_{tx}} - N_{tx}) + (4w + 1) \cdot N_{tx} = \\ (4w + 1) \cdot \overline{N_{tx}} \end{aligned} \tag{4.8}$$

In entrambi i casi $T \approx (4w+1) \cdot \overline{N_{tx}}$. Al variare di N_{tx} la complessità computazionale corrispodente al *mining* di un blocco rimane approssimativamente costante.

Le specifiche dei protocolli blockchain basati su firma digitale *fuzzy* RSA e ECDSA

descritte all'interno del capitolo completano la formulazione dei modelli teorici: gli attori (*users* e *miners*), e gli algoritmi visti fin'ora (*keygen*, firma, verifica, *Proof of Work ad-hoc*, PRNG e *Proof of Work* classica) sono stati riprodotti in fase di implementazione.

Capitolo 5

Simulatore Python3

In questo capitolo viene presentata la progettazione e l'implementazione del simulatore in linguaggio Python3. La codifica dei protocolli blockchain ha richiesto alcuni accorgimenti e adattamenti necessari per lo sviluppo di un'applicazione *standalone*.

5.1 Requisiti

Per il software di simulazione sono stati fissati i seguenti requisiti:

- riprodurre lo schema di firma digitale *fuzzy* nelle varianti RSA e ECDSA con PRNG;
- utilizzare la programmazione concorrente per simulare un numero variabile di *miners* impegnati nella *Proof of Work*;
- permettere all'utente di simulare sia *fair miners* che *evil miners*;
- consentire all'utente di configurare i parametri di funzionamento del simulatore;
- fornire statistiche e metriche per la valutazione dell'efficienza dei protocolli blockchain;
- implementare un'interfaccia grafica.

Non è stata considerata tra gli obiettivi del simulatore, la realizzazione di un'infrastruttura blockchain completa: infatti il programma verrà eseguito su un'unica macchina e non come software distribuito, perciò sono stati trascurati aspetti a basso livello come ad esempio il protocollo di comunicazione tra i nodi. Queste ipotesi semplificative hanno facilitato lo sviluppo del software e permesso di focalizzarsi sulle caratteristiche salienti del progetto.

5.2 Progettazione e implementazione

La fase di progettazione ha prodotto il seguente algoritmo di simulazione:

1. lettura del file di configurazione e *setup* della blockchain

2. creazione della *pool* di *users*
3. creazione della *pool* di *miners*
4. per un numero di blocchi prestabilito dall'utente:
   i. creazione di un pacchetto di transazioni
   ii. invio del pacchetto alla *miner pool*
   iii. *mining* parallelo dei *miners*
   iv. attesa fino alla creazione di un blocco valido
   v. aggiunta del blocco alla blockchain
5. generazione delle statistiche
6. visualizzazione dell'interfaccia grafica

Innanzitutto per *pool* si intende, abusando della terminologia OOP [6], un insieme iterabile di oggetti della stessa classe. Dallo step 4.i si evince l'utilizzo di una struttura dati *custom* chiamata "pacchetto": in un pacchetto vengono inserite tutte le informazioni utili per effettuare il *mining* di un blocco, inviate ai *miners* (step 4.ii). Nello step 4.iii, i *miners* competono simultaneamente nella *Proof of Work*: questo passaggio implica l'utilizzo della programmazione parallela. Per motivi di semplicità, i *miners* non posseggono una copia individuale della blockchain ma esiste soltanto un'unica copia riferita alla *miner pool*.
L'architettura dell'applicazione si basa sul paradigma OOP e integra alcuni *Design Pattern* [7]. Il software è fortemente orientato all'uso di meccanismi di ereditarietà e al riutilizzo del codice.
L'applicativo è suddiviso in due package principali:

- **core** contiene la logica dell'applicazione e le strutture dati utilizzate nella simulazione;
- **gui** contiene l'implementazione dell'interfaccia grafica.

Si presenta il contenuto dei *sub-package* del package **core**.

5.2.1 Package users

Il package **users** definisce le classi **user** e **user_pool,** specializzandoli nelle versioni RSA e ECDSA.
Per notazione, il simbolo @ indica un metodo astratto, cioè non implementato direttamente nella classe ma nelle relative sottoclassi.

[6] *Object Oriented Programming* è un paradigma di programmazione basato sul concetto di oggetto, ovvero un modello dotato di proprietà, metodi e una classe di appartenenza.
[7] I *design pattern* sono soluzioni progettuali a problemi ricorrenti nell'Ingegneria del Software. Essi individuano un insieme di *best practices* utili nello sviluppo e nella progettazione di applicazioni.

Figura 5.1: Suddivisione dell'applicazione in *package*

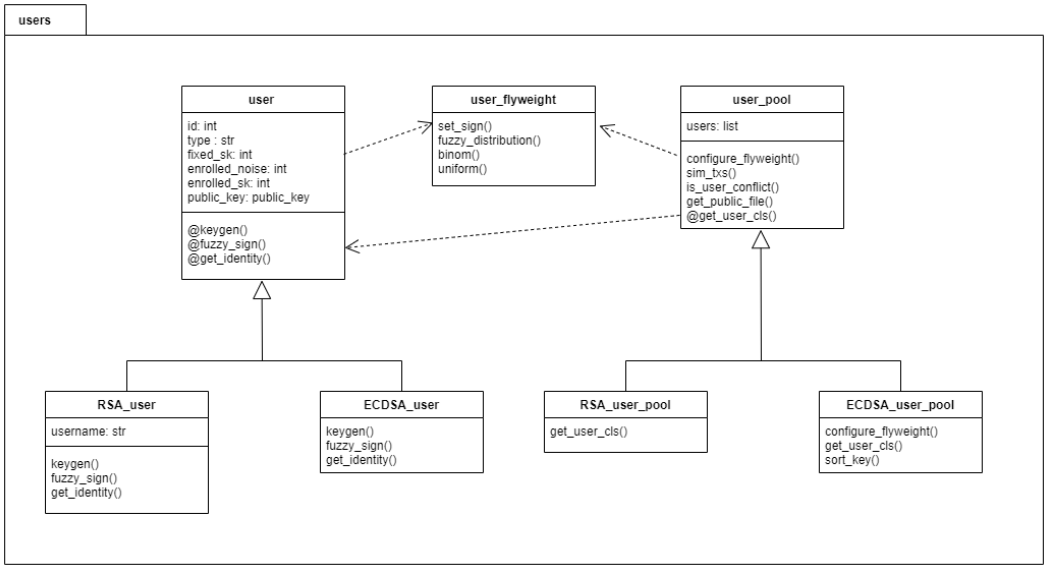


Figura 5.2: Package users

Ogni **user** è caratterizzato da un id numerico, un tipo ("RSA" o "ECDSA"), una chiave privata e una chiave pubblica. Si tiene traccia anche della chiave privata di riferimento (**enrolled_sk**) e del rumore misurato nella fase di *enrollment* (**enrolled_noise**). Tra le operazioni di un user ci sono la generazione delle

chiavi (**keygen()**) e la firma *fuzzy* (**fuzzy_sign()**) di un messaggio. Il metodo **get_identity()** restituisce l'identificativo univoco dell'utente. Le classi **RSA_user** e **ECDSA_user** ereditano da **user** e forniscono un'implementazione dei metodi **keygen()**, **fuzzy_sign()** e **get_identity()**. **RSA_user** aggiunge l'attributo **username**, richiesto nella versione RSA.
La classe **user_pool** contiene il riferimento a una lista di user ed è provvista di alcuni metodi: **is_user_conflict()** determina se esiste sovrapposizione tra le distrubuzioni *fuzzy* segrete degli utenti e in tal caso rimuove l'ultimo utente creato; **sim_txs()** sceglie random alcuni *user* dalla *pool* e genera un pacchetto di transazioni; **get_public_file()** restituisce la lista delle chiavi pubbliche di tutti gli *user*. **RSA_user_pool** e **ECDSA_user_pool** sono rispettivamente *pool* di **RSA_user** e **ECDSA_user**. **ECDSA_user_pool** ordina il *public_file* per chiavi pubbliche crescenti con il metodo **sort_key()**.
Il metodo **configure_flyweight()** imposta alcuni parametri della classe **user_flyweight** per il corretto funzionamento in entrambe le versioni del protocollo. **user_flyweight** include attributi e metodi comuni a tutte le istanze della classe **user**: il suo scopo, come suggerisce il nome, è quello di "alleggerire" gli oggetti user, spostando le proprietà ripetute in una classe di supporto. Tuttavia, a differenza del *design pattern Flyweight* [8], gli oggetti user non mantengono un riferimento al relativo *flyweight* e user_flyweight è una classe con metodi e attributi statici. Il metodo **set_sign()** setta il metodo di firma con chiave segreta a seconda del tipo di distribuzione, che può essere uniforme (**uniform()**) o binomiale (**binom()**); **fuzzy_distribution()** estrae una chiave segreta random dalla distribuzione scelta.

5.2.2 Package mining

Prima di descrivere dettagliatamente il package e le sue componenti, si vuole analizzare com'è stata riprodotta la concorrenza dei *miners*.

Produttore-consumatore

Nel lavoro in parallelo dei *miners* intenti a trovare la *Proof of Work*, si possono trovare numerose analogie con un noto problema informatico: il problema dei produttori e dei consumatori. In tale problema, i produttori creano risorse destinate ai consumatori. Produttori e consumatori condividono un *buffer* limitato per la comunicazione: i produttori creano nuovi *item* e li aggiungono al *buffer*; i consumatori prelevano *item* dal *buffer* e li utilizzano. Bisogna assicurare che il produttore non elabori nuovi dati se il *buffer* è pieno, e che il consumatore non cerchi dati se il *buffer* è vuoto.

[8] Flyweight è un *design pattern* strutturale [17] che permette a più oggetti di condividere parti di stato comuni risparmiando in termini di memoria utilizzata.

Con alcuni necessari adattamenti, è possibile applicare il problema produttore-consumatore al presente contesto operativo dividendo l'interazione tra *miner_pool* e *miners* in due fasi:

Fase I. la *miner_pool* invia sul *buffer packets* tante copie del pacchetto di transazioni quanti sono i *miners*. I *miners* prelevano dal buffer una copia del pacchetto ciascuno e iniziano la *Proof of Work* .

Fase II. il primo miner che completa la *Proof of Work* invia sul *buffer blocks* il blocco generato. La *miner_pool* riceve il blocco e se valido sospende l'attività dei restanti miners.

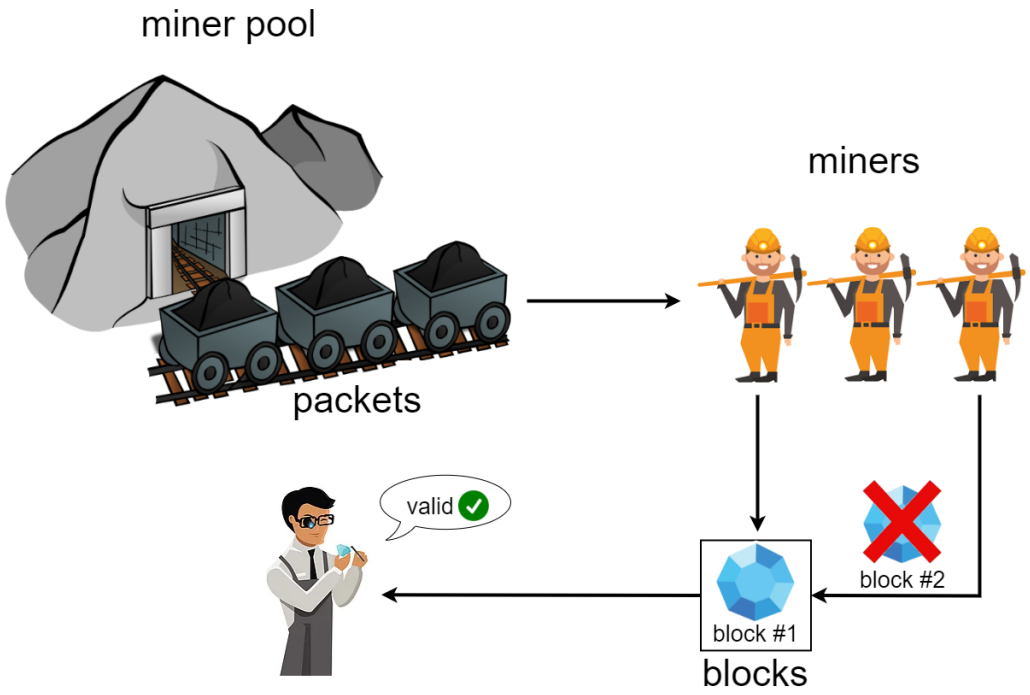


Figura 5.3: Miner pool e miners

Durante la prima fase, la *miner_pool* si comporta da produttore, mentre i *miners* sono i consumatori. Viceversa, nella seconda fase, i *miners* sono i produttori, mentre la *miner_pool* ha il ruolo di consumatore. L'altra importante differenza consiste nell'utilizzo di due *buffer* anziché uno: il *buffer packets* contiene "pacchetti" e ha dimensione pari al numero di *miners*, il *buffer blocks* contiene blocchi "minati" e ha dimensione unitaria.
Per coordinare le operazioni di inserimento e rimozione di *item* dal *buffer*, si ricorre a una primitiva di sincronizzazione chiamata semaforo. Un semaforo è una variabile intera contatore S che può essere acceduta soltanto tramite le seguenti operazioni:

- wait(S): attende fintanto che il contatore è minore uguale a 0, poi decrementa il contatore di 1

- signal(S): incrementa il contatore di 1

Le operazioni wait e signal sono eseguite atomicamente, cioè da solo un processo per volta. Il semaforo è utilizzato per regolare l'accesso a un'area di memoria condivisa da due o più processi in problemi di corsa critica [9].
Nel caso presente si adoperano due semafori binari chiamati *free* e *item*: *free* assume valore 1 se il *buffer blocks* è vuoto e valore 0 se il *buffer blocks* è pieno; viceversa *item* assume valore 1 se il *buffer blocks* è pieno e valore 0 se il *buffer blocks* è vuoto. Inizialmente il *buffer blocks* è vuoto quindi *free* = 1 e *item* = 0. Ricapitolando, il *workflow* di *miner_pool* e *miners* è descritto dal seguente pseudocodice:

```
miner_pool():                          miner():

for 1 to n_miners:                     packet = packets.get()
 packets.add(packet)                   block = mining()
item.wait()                            free.wait()
block = blocks.get()                   blocks.add(block)
if block is valid:                     item.signal()
 //save stats and logs
 //stop miners
free.signal()
```

dove i metodi **add()** e **get()** aggiungono e rimuovono *item* dal *buffer*. Il *buffer* è una struttura dati FIFO (First In First Out): il metodo **get()** rimuove l'*item* presente da più tempo nel *buffer*, il metodo **add()** aggiunge il nuovo *item* in ultima posizione. Le statistiche e i log vengono salvati in *buffers* supplementari.

Multiprocessing in Python

Il linguaggio Python mette a disposizione due package per la computazione parallela: *multiprocessing* e *threading* [18].
Il multiprocessing è una tecnica di programmazione concorrente basata sull'uso di una molteplicità di processi. I processi sono programmi in esecuzione dotati di spazi di memoria indipendenti. Ogni processo può essere suddiviso in un certo numero di *thread* o "processi leggeri" che condividono lo spazio di memoria del processo genitore. L'esecuzione parallela di thread è definita multithreading.
In Python il multithreading non supporta il "vero" parallelismo: due o più thread non sono eseguiti contemporaneamente e non è possibile sfruttare tutti i CPU-core presenti sulla macchina. Questa limitazione deriva da una *feature* di Python, il *Global*

[9] La corsa critica o *race condition* è una situazione tipica dei sistemi concorrenti in cui il risultato finale dipende dall'ordine di esecuzione dei processi. La porzione di codice che accede alla risorse condivise è detta sezione critica.

Interpreter Lock (GIL), ovvero un *mutex* (semaforo binario) che consente a un solo thread per volta di avere il controllo dell'interprete Python. Il GIL, introdotto per proteggere la memoria da possibili stati di corruzione dovuti a *race condition* nel multithreading, pone un freno a tutte le applicazioni CPU-bound che fanno un uso intensivo del processore, come ad esempio il mining.
Per superare queste restrizioni, si ricorre al multiprocessing: i vari processi posseggono ognuno un'istanza diversa dell'interprete Python quindi il GIL non costituisce più un problema.
Nel progetto è stata effettuata proprio questa scelta: ogni miner è un processo distinto. Ad ogni iterazione, la *miner_pool* istanzia i processi miner e li distrugge quando viene prodotto un blocco valido. Vengono utilizzate le seguenti classi del package multiprocessing:

- **Process**: crea un'istanza di processo. Il metodo **start()** avvia il processo; il metodo **run()** definisce il task che il processo deve completare; il metodo **terminate()** interrompe il processo.

- **Queue**: costruisce una coda FIFO *process-safe*, cioè in grado di gestire la concorrenza dei processi. Il metodo **put()** aggiuge elementi alla coda; il metodo **get()** rimuove elementi alla coda e attende fintanto che la coda è vuota.

- **Semaphore**: rappresenta un semaforo. I metodi **acquire()** e **release()** sono rispettivamente implementazioni di wait e signal.

Struttura

Il package mining definisce le classi **miner** e **miner_pool**, specializzandoli nelle versioni RSA e ECDSA. Ogni miner ha un id numerico, un tipo ("RSA" o "ECDSA"),

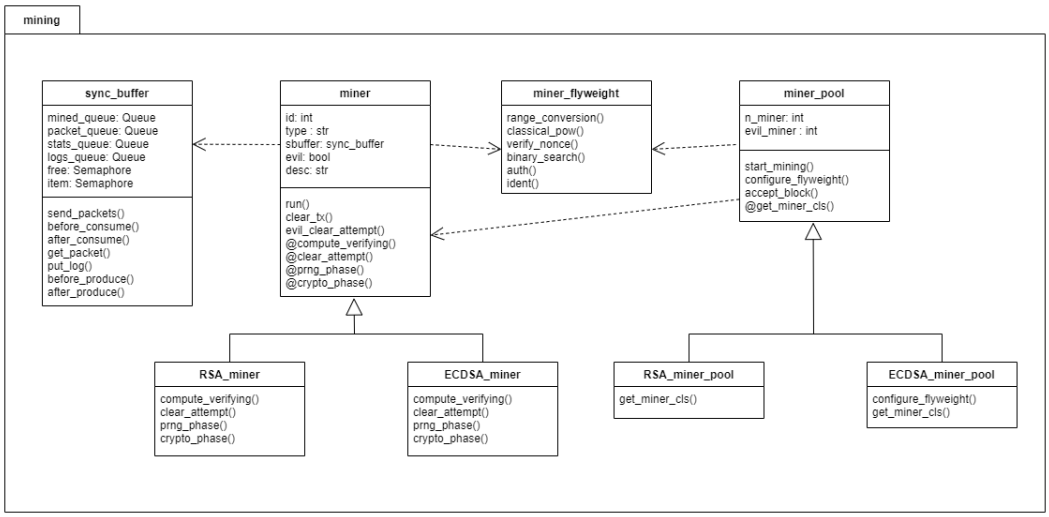


Figura 5.4: Package mining

un "gestore" dei buffers e un valore booleano posto a *true* se è *evil* e *false* se è *fair*. **run()** contiene il task principale del miner; **clear_tx()** "pulisce" una transazione e **evil_clear_attempt()** effettua un tentativo di *clearing* secondo la logica dell' *evil miner*. Le classi **RSA_miner** e **ECDSA_miner** forniscono un'implementazione dei metodi **compute_verifying()**, che computa i valori *non-fuzzy* di firma o chiave pubblica, **clear_attempt()**, che effettua un tentativo di *clearing* secondo la logica del *fair miner*, **prng_phase()**, che corrisponde alla fase PRNG, e **crypto_phase()**, che corrisponde alla fase CRYPTO.
La classe **sync_buffer** espone un'interfaccia per la gestione dell'interazione tra *miner_pool* e *miners* descritta precedentemente. Ha per attributi i buffers *packets*, *blocks*, **stats** e **logs** e i semafori *free* e *item*.
Miner_pool rappresenta un insieme di miners. L'attributo n_miner è il numero di *miners* totali, evil_miner è il numero di *evil miners*. I *miners* vengono istanziati ad ogni iterazione all'interno del metodo **start_mining()** che prepara la fase di mining. Il metodo **accept_block()** provvede a verificare la correttezza dei blocchi "minati". Le classi **RSA_miner_pool** e **ECDSA_miner_pool** ereditano da miner_pool e implementano il metodo **get_miner_cls()** che restituisce la classe dei miners.
Il metodo **configure_miner_flyweight()** configura la classe **miner_flyweight** a seconda del protocollo RSA o ECDSA. Qui valgono le osservazioni già fatte per la classe user_flyweight. **range_conversion()** trasforma un *digest* pseudo-casuale in un intero nell'insieme $[-2w, 2w]$; **classical_pow()** effettua il mining di una *Proof of Work* classica; **verify_nonce()** controlla la correttezza del nonce di una PoW classica; **binary_search()** esegue una ricerca binaria sul file pubblico; **auth()** e **ident()** compiono rispettivamente un tentativo di autenticazione e di identificazione cercando il *test value* nel file pubblico.

5.2.3 Package stats

Il package **stats** è dedicato al calcolo delle statistiche e alla creazione di un report della simulazione. Insieme al blocco "minato", il miner produce anche un oggetto **block_stats** che contiene le seguenti informazioni:

- l'id del miner
- il booleano *evil*
- la lista delle transazioni
- il tempo impiegato per "pulire" ogni transazione
- il tempo totale impiegato nella fase PRNG per ogni transazione
- il tempo totale impiegato nella fase CRYPTO per ogni transazione
- il numero di tentativi effettuati per ogni transazione

- il tempo impiegato per la PoW classica
- il numero di tentativi effettuati per la PoW classica

A questa struttura dati si aggiungono i log generati dal miner stesso e da tutti gli altri miners.
Al termine della simulazione, i vari block_stats generati, uno per ogni blocco, sono utilizzati per generare alcune statistiche globali, memorizzate in un oggetto **global_stats** che include:

- il tempo totale della simulazione
- il numero di blocchi "minati"
- il tempo medio di creazione di un blocco
- il tempo medio di *clearing* di un pacchetto di transazioni
- il tempo totale PRNG medio per un pacchetto di transazioni
- il tempo totale CRYPTO medio per un pacchetto di transazioni
- il rapporto tra il tempo totale PRNG medio e il tempo totale CRYPTO medio
- il numero medio di tentativi per il *clearing* di un pacchetto di transazioni
- il tempo medio della PoW classica
- il numero medio di tentativi per la PoW classica

Tutte le statistiche, eccetto la prima, sono disponibili in quattro modalità:

- *total*: viene mostrato un dato cumulativo
- *fair-evil*: i dati sono distinti tra *evil* e *fair miners*
- *transactions*: i dati sono distinti per numero di transazioni
- *combined*: combinazione delle modalità *fair-evil* e *transactions*

La classe **report** è un contenitore per block_stats e global_stats e presenta metodi per la compilazione di un resoconto della simulazione; l'oggetto **stats_factory** si occupa del calcolo delle statistiche globali.

5.2.4 Interfaccia grafica

L'interfaccia grafica, sviluppata con il *framework* Tkinter, mostra un semplice riepilogo con i risultati della simulazione. La finestra è organizzata in tab:

- nel tab *general* sono riportati il tempo totale della simulazione e le impostazioni utilizzate

- nel tab *users* è presente uno *slider* che permette di navigare tra gli users e leggere le relative informazioni
- nel tab *blocks* è presente un *explorer* della blockchain che mostra per ogni blocco i log, il contenuto in formato JSON e alcune informazioni statistiche
- nel tab *stats* si possono visualizzare le statistiche globali in forma di grafici, selezionando il tipo di statistica e la modalità

Nei tab *users* e *blocks* i pulsanti "prev" e "next" mostrano l'elemento precedente e successivo. Il pulsante "Go to" mostra l'elemento specificato nella spinbox. Nel tab *stats* i grafici vengono visualizzati selezionando un valore nelle combobox e premendo il pulsante "Plot".
La Gui viene generata soltanto al termine della simulazione per evitare di aggiungere ulteriori *overhead* all'esecuzione del simulatore.

5.2.5 Dettagli implementativi

Il package **datastruct** contiene strutture dati di base per il protocollo: **block** (blocco), **packet** (pacchetto), **public_key** (chiave pubblica), **raw_tx** (transazione), **signature** (firma). Un packet racchiude tutte le informazioni necessarie a effettuare il mining di un blocco: il numero del nuovo blocco, il parentHash, la lista di transazioni e il file pubblico. I blocchi hanno la seguente struttura JSON:


```
[
    {
        "parentHash": "a6a0fe92bc4f1fb88f23f15cd6e7c001e78d2342f0f3aae5eb6087f3ec4460cf",
        "blockNumber": 2,
        "NumTx": 3,
        "difficulty": 0,
        "nonce": ""
    },
    [
        {
            "Tx#": 0,
            "identity": "4GA63Nr9CVDLxdR",
            "timestamp": "2022-09-14 18:14:53.512646",
            "signature": {
                "s": "0x4f6b1dfc5e06c3e224f244d02c90663965e388c51b11b3138e248bbdc1b55be6"
            },
            "PoW": "0x95a6d33ba84eee25b10759eb8dcc44393fc6b387d7d0c1b83589478525a356e"
        },
        {
            "Tx#": 1,
            "identity": "baZdDq8Cw7d1n",
            "timestamp": "2022-09-14 18:14:53.513062",
            "signature": {
```

```
                "s": "0x1bedb7e668460e6a22ab8c0811e242d4f79331dcb6c32127fc3e17a5dec63b63"
            },
            "PoW": "0x26ff444d2efed13bdc0b1c1df61e7c9180c1cc53a6b45f6b0292c17397e962ce"
        },
        {
            "Tx#": 2,
            "identity": "0BwqYS8eCvtd",
            "timestamp": "2022-09-14 18:14:53.513290",
            "signature": {
                "s": "0x4e1158f7f4e197fe6ab85704f682f5b8fd521cdc5a57074e3058cf79ec9acb7c"
            },
            "PoW": "0x14be3d5e3b687d7f4ec370f2fec5eddb5f09d982948a5a1e3709a80e1cb146d4"
        }
    ]
]
```


IL *block header* contiene il *parentHash*, il numero di blocco, il numero di transazioni, la *difficulty* della PoW classica e il valore di *nonce*. Il *block body* è costituito da *NumTx* transazioni ognuna con un id numerico, l'identità dell'utente, un *timestamp* corrispondente al messaggio, la firma dell'utente e la *Proof of Work*. Il blocco precedente è un esempio di blocco nel protocollo RSA. Nel protocollo ECDSA la struttura del blocco rimane la stessa ma le firme relative alle transazioni sono formate dai parametri r, s, v. Nel caso presente la *difficulty* è uguale a 0 e il *nonce* non è valorizzato perché il blocco contiene il numero massimo di transazioni consentite dunque non è previsto mining aggiuntivo.
La generazione dei pacchetti di transazioni è gestita in modo uniforme, cioè vengono creati in egual numero i pacchetti che contengono $1, 2, ...\overline{N_{tx}}$ transazioni. Se il numero di blocchi da simulare non è divisibile per $\overline{N_{tx}}$ la distribuzione dei pacchetti in base al numero di transazioni è approssimativamente uniforme.
Il package **util** implementa alcuni strumenti ausiliari. **factory** è una classe che, come suggerisce l'omonimo design pattern, crea oggetti, in particolare **user_pool** e **miner_pool**; **tools** è una classe statica contenente un metodo di hash, basato sulla funzione *keccak* del modulo *Crypto.Hash*, e un metodo **crono()** da usare come decorator [10] per misurare il tempo di esecuzione di altre funzioni.
Il file **main.py** è l'*entry point* dell'applicazione: si occupa di validare le impostazioni presentate in formato JSON e creare un oggetto **blockchain**. Il metodo **simulate()** della classe **blockchain** viene invocato dal main per avviare una simulazione. La classe **sync_buffer** implementa la logica produttore-consumatore descritta nella sottosezione 5.2.2.
Per ulteriori approfondimenti si consiglia di consultare gli *snippet* di codice presenti

[10] Il *decorator design pattern*[19] permette di aggiungere dinamicamente funzionalità ad un oggetto senza modificare la classe originale. La relativa implementazione prevede la costruzione di un *wrapper*, ovvero una classe o un metodo astratto che "avvolge" l'oggetto.

in appendice A.1 o in alternativa il sorgente del progetto al seguente repository github.

Capitolo 6

Risultati

Il capitolo presenta e discute i risultati ottenuti con il simulatore.

6.1 Setup

Per un corretto uso del software, osservare i seguenti passaggi:

1. disporre di una versione recente di Python3 e del *package installer* pip
2. scaricare il sorgente dal link e posizionarsi nella cartella del progetto
3. in Linux eseguire da terminale il comando

   ```
   apt-get install libgmp3-dev python3-tk python3-pil python3-
   pil.imagetk
   ```

4. eseguire da terminale il comando

   ```
   pip3 install -r requirements.txt
   ```

   (Linux) o

   ```
   pip install -r requirements.txt
   ```

   (Windows) per l'installazione delle dipendenze
5. spostarsi nella cartella "simulator"
6. (opzionale) modificare il file di configurazione **config.json**
7. eseguire da terminale il comando

   ```
   python3 main.py
   ```

   (Linux) o

   ```
   python main.py
   ```

   (Windows) per avviare l'applicazione.

Il file config.json contiene i parametri di configurazione del simulatore.

```
{
    "general" : {
        "type" : "<< RSA | ECDSA >>",
        "distr" : "<< uniform | binom >>",
        "n_blocks" : "<< from 1 to 10000 >>",
        "n_miners" : "<< from 1 to 16 >>",
        "n_evils" : "<< from 0 to 16 >>",
        "n_users" : "<< from 1 to 10000 >>",
        "modulus_bit_length" : "<< 256 | 512 | 1024 | 2048 >>",
        "w" : "<< from 1 to 10000 >>",
        "max_tx" : "<< from 1 to 10 >>",
        "base_diff" : "<< from 1 to 10 >>"
    },
    "PRNG" : {
        "a" : "<< near to 2^mbl >>",
        "b" : "<< near to 2^mbl >>",
        "n" : "<< near to 2^mbl >>",
        "X" : "<< from 1 to 100 >>"
    },
    "ECDSA" : {
        "Gx" : "<< G x coordinate >>",
        "Gy" : "<< G y coordinate >>",
        "n" : "<< near to 2^mbl >>",
        "q" : "<< near to 2^mbl >>"
    }
}
```

Nella sezione "general" devono essere specificati in ordine il tipo di protocollo, il tipo di distribuzione delle chiavi segrete ("uniforme" o "binomiale"), il numero di blocchi, il numero di miners, il numero di *evil_miners*, il numero di utenti, la lunghezza in bit dei parametri RSA, ECDSA e PRNG, il parametro pubblico w, il numero massimo di transazioni per blocco, il parametro *base_diff*, il cui significato è spiegato in seguito 6.3.1.
Nella sezione "PRNG" devono essere indicati i parametri pubblici a, b, n e X e nella sezione "ECDSA", i parametri di configurazione specifici per la versione del protocollo ECDSA: le coordinate del punto generatore G, l'ordine n e la cardinalità del campo finito primo q.
Il programma genera un'eccezione se:

- la struttura del file JSON è alterata
- le impostazioni non rispettano i vincoli
- il numero di *evil miners* è maggiore del numero totale di *miners*
- i parametri PRNG a, b, n sono maggiori di 2^{mbl} dove mbl è il *modulus bit length*
- i parametri ECDSA n, q sono sono maggiori di 2^{mbl}

I parametri G_x e G_y devono corrispondere alle coordinate di un punto generatore della curva ellittica **secp256k1** del modulo **fastecdsa.curve**.
Il parametro mbl indica anche la lunghezza in bit del *public modulus* delle chiavi pubbliche RSA. mbl è utilizzato per specificare un limite superiore per i parametri pubblici RSA, ECDSA e PRNG, che per una resa efficace dei protocolli devono avere tutti lo stesso ordine di grandezza.

6.2 Simulazione d'esempio

Le seguenti schermate illustrano un esempio di simulazione.
La figura 6.1 mostra l'esecuzione del simulatore da terminale.

Figura 6.1: Terminale simulatore

Dopo il *check* del file di configurazione e la creazione di *user* e *miner pool*, il programma notifica la creazione sequenziale dei blocchi. Al termine della simulazione, viene generata un'interfaccia grafica riassuntiva.
Ad esempio la finestra in figura 6.2 riporta le impostazioni della simulazione. In questo caso è stata eseguita la versione RSA.

L'interfaccia in figura 6.3 mostra il primo blocco creato dalla simulazione. Il blocco è stato prodotto dal miner 3 che è stato il primo a concludere la PoW classica. I logs evidenziano che i miner 0 e 1 hanno "minato" la transazione presente nel blocco prima del miner 3, ma non sono riusciti a terminare per primi la PoW classica. L'unica transazione presente nel blocco si riferisce all'utente "6qlZcfK9wwFuUKA" mostrato in figura 6.4.

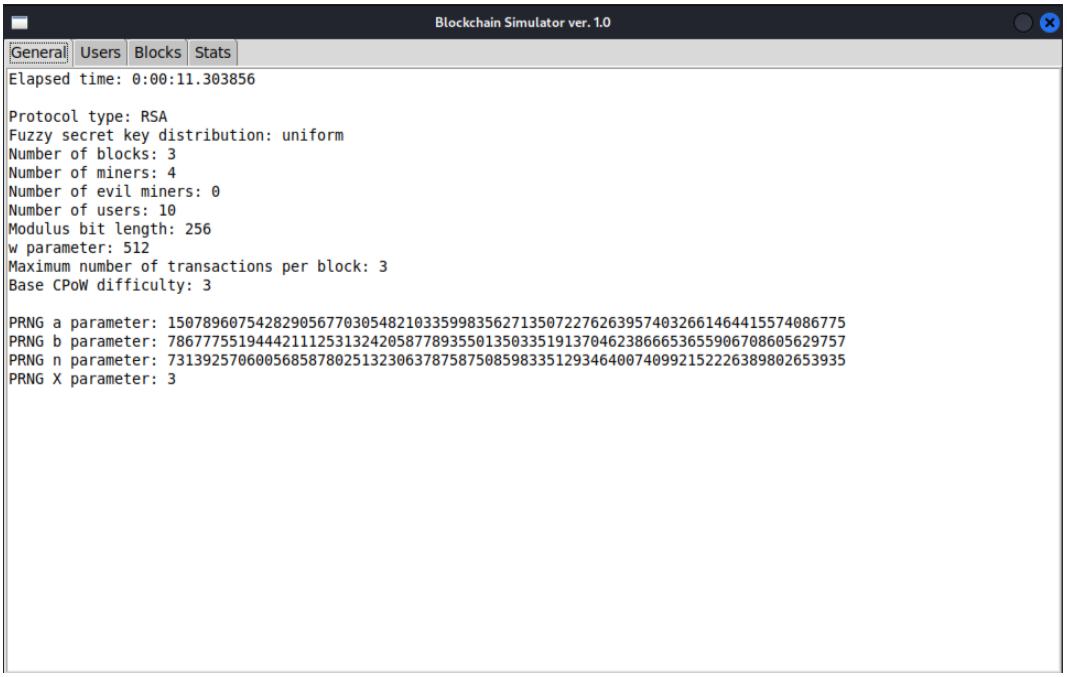


Figura 6.2: Impostazioni simulazione d'esempio

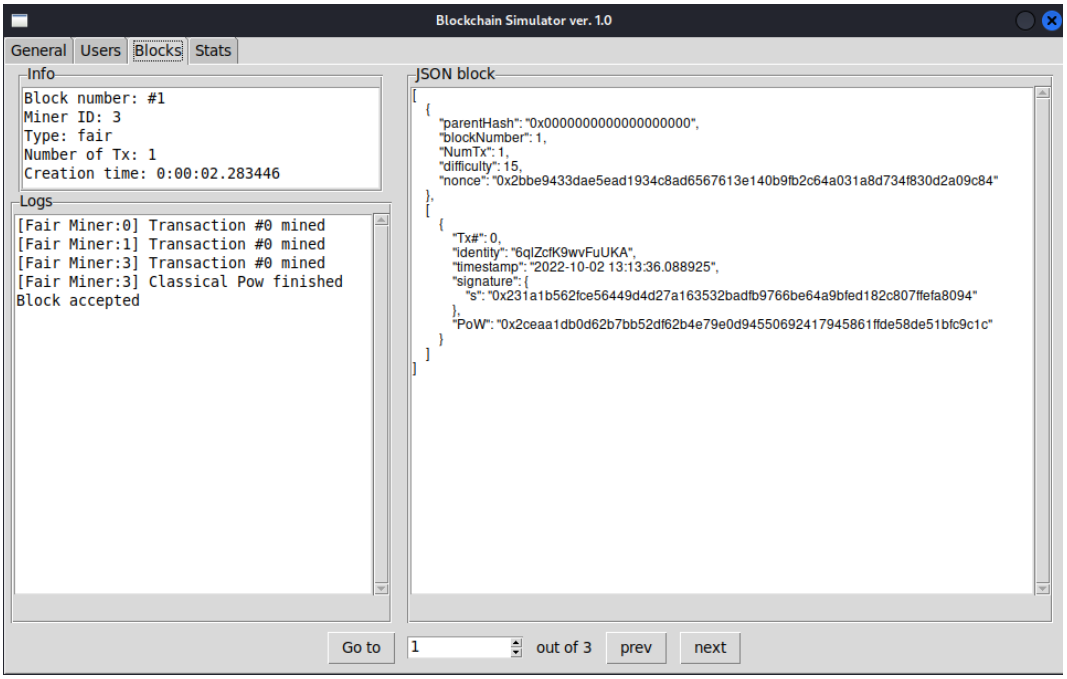


Figura 6.3: Blocco d'esempio

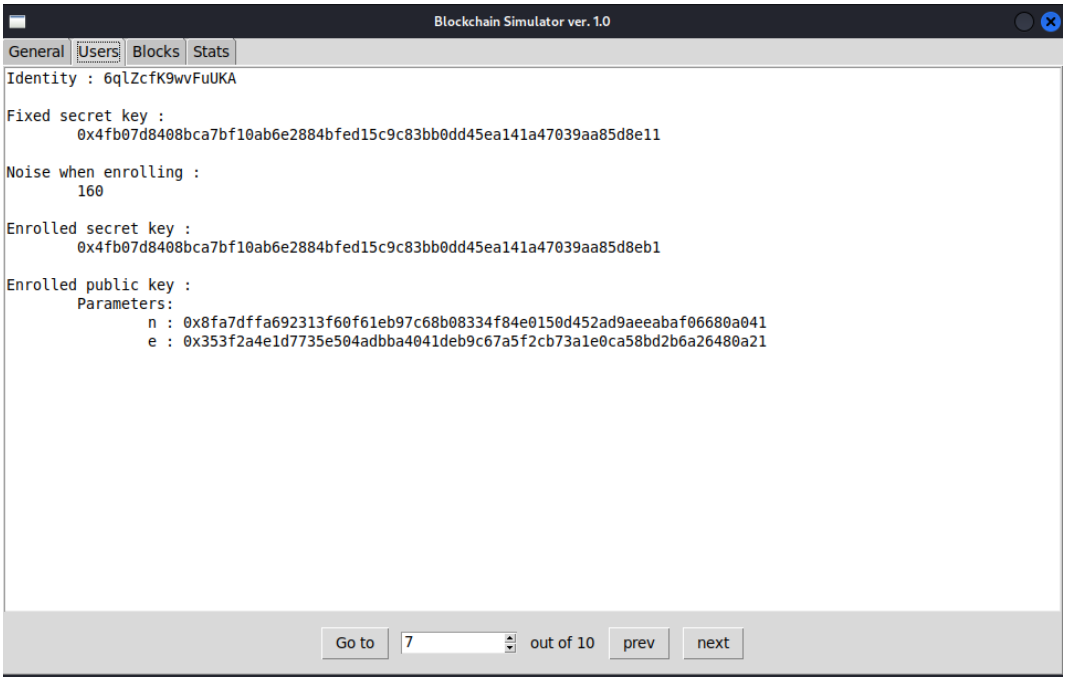


Figura 6.4: Utente d'esempio

6.3 Osservazioni

6.3.1 Parametro difficulty base

I primi esperimenti con il simulatore hanno evidenziato una disparità nel tempo di creazione dei blocchi con un diverso numero di transazioni. In particolare si è notato che il tempo di creazione di un blocco cresceva proporzionalmente con il numero di transazioni. Quest'effetto indesiderato deriva da un *tuning* impreciso della *difficulty* della PoW classica: infatti l'ipotesi semplificativa che il tempo di esecuzione di una funzione di hash e il tempo impiegato da un tentativo di *clearing* siano uguali, nella pratica non si verifica. Correggendo l'equazione (4.5) il mining di un blocco impiega un tempo:

$$T = 2^k \cdot t_{hash} + (4w+1)N_{tx} \cdot t_{clear} \tag{6.1}$$

dove $t_{clear} > t_{hash}$. Per uniformare il tempo di creazione dei blocchi, possiamo porre:

$$\begin{cases} k = 0 \quad \text{se} \quad N_{tx} = \overline{N_{tx}} \\ k = k_{base} + \text{round}(\log_2((4w+1) \cdot (\overline{N_{tx}} - N_{tx}))) \quad \text{altrimenti} \end{cases} \tag{6.2}$$

dove k_{base} è il parametro difficulty base corrispondente alla voce "base_diff" nel file di configurazione.
Per $k = 0$ e $N_{tx} = \overline{N_{tx}}$ si ha che:

$$\begin{aligned} T = t_{hash} + (4w+1)\overline{N_{tx}} \cdot t_{clear} \approx \\ (4w+1)\overline{N_{tx}} \cdot t_{clear} \end{aligned} \tag{6.3}$$

Per $k = \text{round}(\log_2((4w+1)\cdot(\overline{N_{tx}} - N_{tx}))$ e trascurando l'approssimazione intera si ha che:

$$\begin{aligned} T = 2^{k_{base}} \cdot (4w+1)(\overline{N_{tx}} - N_{tx}) \cdot t_{hash} + (4w+1)N_{tx} \cdot t_{clear} = \\ (4w+1)t_{hash}\,(2^{k_{base}} \cdot (\overline{N_{tx}} - N_{tx}) + N_{tx} \cdot \frac{t_{clear}}{t_{hash}}) = \\ (4w+1)t_{hash}\,(2^{k_{base}}\overline{N_{tx}} + (\frac{t_{clear}}{t_{hash}} - 2^{k_{base}}) \cdot N_{tx}) \end{aligned} \tag{6.4}$$

Per minimizzare il termine $(\frac{t_{clear}}{t_{hash}} - 2^{k_{base}}) \cdot N_{tx}$, dipendente dal numero di transazioni, scegliamo:

$$k_{base} = \text{round}(\, log_2(\frac{t_{clear}}{t_{hash}})\,) \tag{6.5}$$

Trascurando l'approssimazione intera di k_{base} il termine dipendente da N_{tx} nell'equazione (6.4) si elide, dunque:

$$\begin{aligned} T \approx (4w+1)t_{hash} \cdot 2^{k_{base}}\overline{N_{tx}} \approx \\ (4w+1)t_{hash} \cdot \frac{t_{clear}}{t_{hash}}\overline{N_{tx}} = \\ (4w+1)\overline{N_{tx}} \cdot t_{clear} \end{aligned} \tag{6.6}$$

In entrambi i casi $T \approx (4w+1)\overline{N_{tx}} \cdot t_{clear}$.
Negli esperimenti con il simulatore, applicando l'equazione (6.5), è stato trovato $k_{base} = 3$ per la versione RSA del protocollo e $k_{base} = 6$ per la versione ECDSA. Questi valori di k_{base} sono anche stati utilizzati negli esperimenti.

6.3.2 t_{PRNG} e t_{CRYPTO}

Nella sezione 4.3 si specifica che nei protocolli RSA e ECDSA la funzione PRNG è costruita in modo da garantire $\frac{t_{PRNG}}{t_{CRYPTO}} = X$. Negli esperimenti con il simulatore, risulta $\frac{t_{PRNG}}{t_{CRYPTO}} \gtrapprox X$ nella versione RSA e $\frac{t_{PRNG}}{t_{CRYPTO}} \gg X$ nella versione ECDSA. Il rapporto $\frac{t_{PRNG}}{t_{CRYPTO}}$ è maggiore di X per i seguenti motivi:

- nella funzione PRNG sono presenti operazioni di hash e conversione dei *seed* che aumentano il tempo di esecuzione;
- nella versione RSA, confrontando il calcolo $y = \widehat{b} \cdot \widehat{a}^{seed'} \mod \widehat{n}$ della funzione RSAPRNG e il calcolo $c'' \equiv c' \cdot c^{\Delta e} \mod n$ della funzione RSAClearSign, si può concludere che il primo è computazionalmente più oneroso. Infatti sebbene gli altri parametri abbiano lo stesso ordine di grandezza, in generale vale $seed' \gg \Delta e$ perché $seed' \in [1, 2^n]$ mentre $\Delta e \in [-2w, 2w]$. L'elevamento a potenza per numeri grandi può impiegare più CPU time.
- nella versione ECDSA, confrontando il calcolo $Y = \widehat{Q} + seed' \cdot G$ della funzione ECDSAPRNG e il calcolo $Q = Q' + \Delta e \cdot G$ della funzione ECDSAClearPublicKey, si può che il primo è computazionalmente più oneroso. Infatti sebbene gli

altri parametri abbiano lo stesso ordine di grandezza, in generale abbiamo $seed' \gg \Delta e$ perché $seed' \in [1, 2^n]$ mentre $\Delta e \in [-2w, 2w]$. La moltiplicazione per numeri grandi può impiegare più CPU time.

Il valore di $\frac{t_{PRNG}}{t_{CRYPTO}}$ dipende fortemente dall'implementazione dei protocolli, e in particolare, a basso livello, dalle ottimizzazioni relative alle operazioni di moltiplicazione ed elevamento a potenza modulo n. Nel caso del presente simulatore, il maggiore rapporto $\frac{t_{PRNG}}{t_{CRYPTO}}$ ha l'effetto positivo di incrementare la BFT.

6.4 Raccolta dati

Per testare funzionamento e performance del simulatore, sono stati eseguiti alcuni esperimenti variando le configurazioni del programma. L'ambiente di testing è una macchina virtuale Ubuntu 22.04 LTS istanziata su una macchina con processore AMD Ryzen 5 2600X Six-Core con frequenza di *clock* di 3.6 GHz.
Al fine di ottenere dei risultati sperimentali significativi, è stata simulata la creazione di un campione statistico di 300 blocchi.
Per tutti test il *modulus bit length* è fissato a 256 bit: ciò implica che anche le chiavi RSA sono lunghe 256 bit con conseguente riduzione della *security* del protocollo nella versione RSA. Questa misura si è resa necessaria per evitare simulazioni dai tempi eccessivamente lunghi.
Per ogni test si riportano le impostazioni applicate, le statistiche relative a tempo totale della simulazione, tempo medio di *clearing* e di PoW classica, numero medio di tentativi per *clearing* e PoW classica e alcuni grafici generati mediante l'interfaccia grafica. I grafici relativi ai test sono disponibili nell'appendice A.2. Per i test 1-10, 11, 13 i grafici rappresentano il tempo medio di creazione dei blocchi al variare del numero di transazioni; per i test 12, 14 i grafici rappresentano il numero di blocchi "minati" e il tempo medio di creazione dei blocchi al variare del numero di transazioni per *fair* e *evil miners*. I grafici aggiuntivi per il test 1 e il test 6 mostrano la suddivisione del tempo medio di *clearing* in fase PRNG e fase CRYPTO e il rapporto tra queste al variare del numero di transazioni.
Nel dettaglio delle impostazioni dei test sono state omesse le seguenti:

```
Fuzzy secret key distribution: uniform
Number of blocks: 300
Number of users: 100
Modulus bit length: 256
Maximum number of transactions per block: 3

PRNG a parameter: 150789607542829056770305482103359983562713507227626395740326614644155740867 75
PRNG b parameter: 786777551944421112531324205877893550135033519137046238666536559067086056297 57
PRNG n parameter: 731392570600568587802513230637875875085983351293464007409921522263898026539 35
PRNG X parameter: 3

ECDSA Gx parameter: 550662630222773436695787188951685343262506034537775941755001873603891167292 40
ECDSA Gy parameter: 326705100207588169780830851305070431844712733806592432759389043357573374824 24
ECDSA n parameter: 115792089237316195423570985008687907852837564279074904382605163141518161494337
ECDSA q parameter: 115792089237316195423570985008687907853269984665640564039457584007908834671663
```

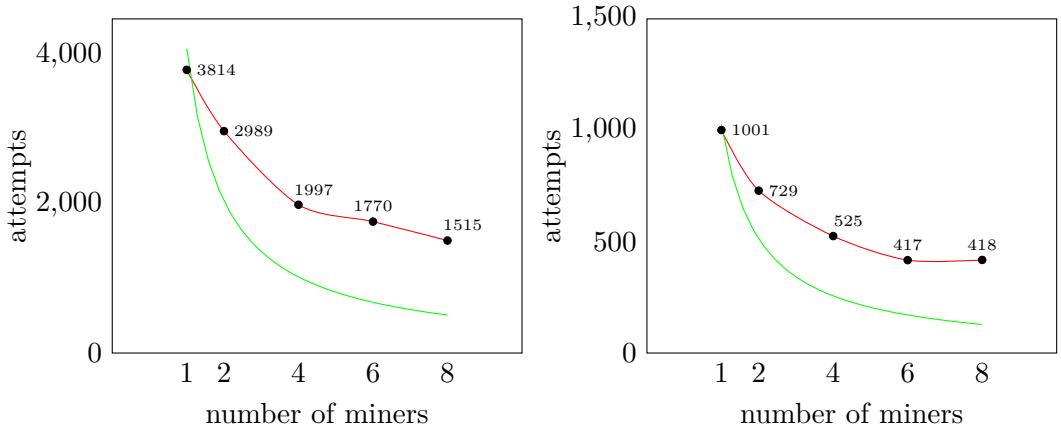
perché risultano uguali per tutti i test.

6.5 Analisi dei dati

In questa sezione si analizzano i dati ottenuti dai precedenti test.

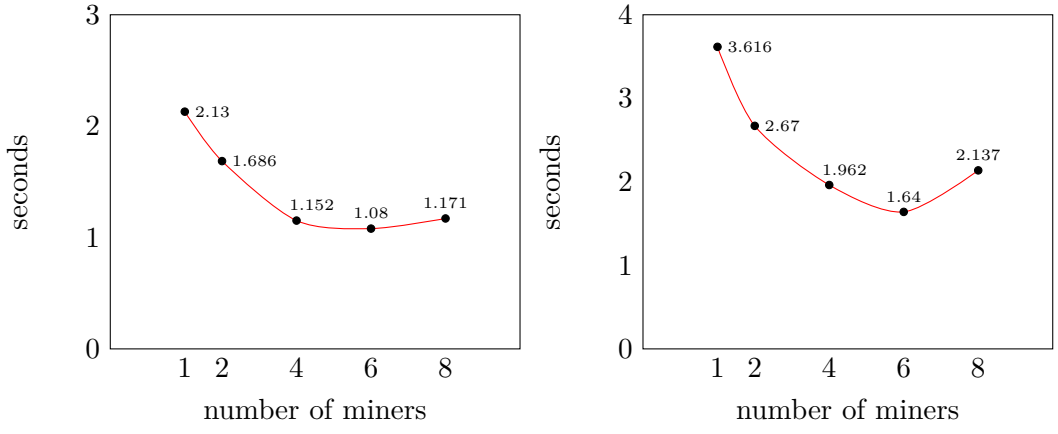
6.5.1 Tempi e tentativi di clearing per miners multipli

I grafici seguenti rappresentano il tempo impiegato e i tentativi di *clearing* effettuati al variare del numero di *miners*. Nelle statistiche sono stati considerati tutti i blocchi simulati, quindi i blocchi contenenti 1, 2 e 3 transazioni. Le curve ottenute decresco-



(a) RSA version (b) ECDSA version

Figura 6.5: Average clearing attempts



(a) RSA version (b) ECDSA version

Figura 6.6: Average clearing time

no all'aumentare del numero di miners. Con più di 6 *miners*, cioè un numero pari ai

CPU core presenti sulla macchina, i tempi e i tentativi impiegati subiscono soltanto piccole variazioni. Una spiegazione di questo andamento è la seguente: superato il numero di CPU core, il multiprocessing non contribuisce più all'abbassamento dei tempi e del numero di tentativi impiegati nel mining, perché i processi-miner non sono più mappati 1 a 1 sui CPU core e sono necessarie operazioni di *context-switch* [11]. Dunque simulare un numero di miners maggiore dei CPU core è poco significativo.

Nei grafici relativi al numero di tentativi di *clearing*, la curva verde rappresenta l'andamento ideale della caratteristica, cioè la funzione:

$$y = \frac{(4w+1) \cdot \widehat{N_{tx}}}{x} \tag{6.7}$$

dove x è il numero di miners e $\widehat{N_{tx}}$ è il numero medio di transazioni per blocco.
Nei test effettuati i blocchi contengono da 1 a 3 transazioni perciò $\widehat{N_{tx}} = 2$. Per la versione RSA $w = 512$, mentre per la versione ECDSA $w = 128$.
La curva reale si discosta dalla curva ideale per motivi intriseci all'implementazione e ai parametri utilizzati: una possibile causa può consistere negli *overhead* legati al *multiprocessing*, e in particolare alla gestione dell'accesso a strutture dati condivise tra i processi; un altro fattore determinante può essere la scelta di un parametro w piccolo per contenere i tempi di simulazione.

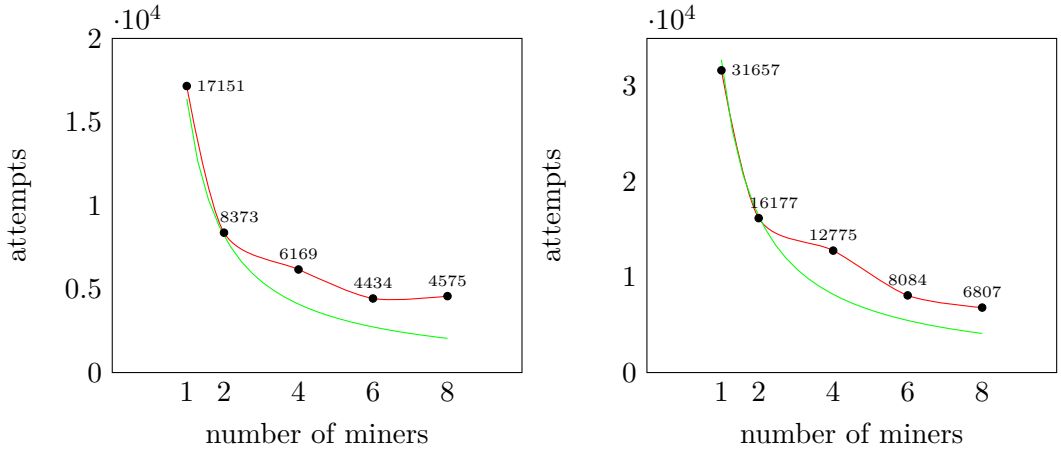
6.5.2 Tempi e tentativi della PoW classica per miners multipli

I grafici seguenti rappresentano i tempi impiegati e i tentativi della PoW classica effettuati al variare del numero di *miners*. Si considerano tutti i blocchi simulati.
Analogamente ai tempi e ai tentativi relativi alla fase di *clearing*, anche per la PoW classica si può osservare una saturazione delle prestazioni al superamento di 6 miners simultanei. Nei grafici relativi al numero di tentativi della PoW classica, la curva verde rappresenta l'andamento ideale della caratteristica, cioè la funzione:

$$y = \frac{\sum_{N_{tx}=1}^{\overline{N_{tx}}-1} 2^{k_{base}+\text{round}(\log_2((4w+1)\cdot(\overline{N_{tx}}-N_{tx})))}}{\overline{N_{tx}} \cdot x} \tag{6.8}$$

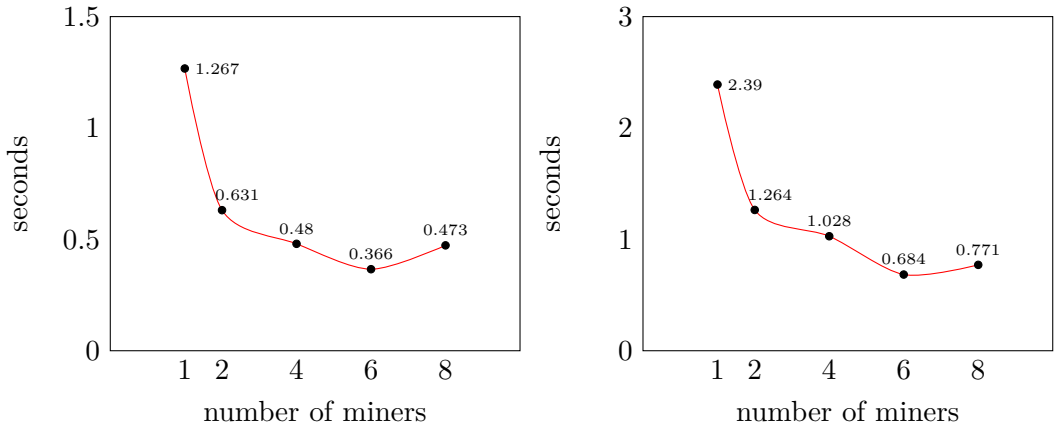
dove x è il numero di miners e $\overline{N_{tx}}$ è il numero massimo di transazioni per blocco.
Il termine al numeratore è la sommatoria del numero medio di tentativi della PoW classica per blocchi con diverso numero di transazioni: per $N_{tx} = \overline{N_{tx}}$ il numero di tentativi è 0.
Nei test RSA effettuati $w = 512, k_{base} = 3, \overline{N_{tx}} = 3$ dunque $y = \frac{2^{14}}{x}$. Nei test ECDSA effettuati $w = 128, k_{base} = 6, \overline{N_{tx}} = 3$ dunque $y = \frac{2^{15}}{x}$.

[11] Le commutazioni di contesto o *context-switch* [20] indicano un insieme di operazioni svolte dal sistema operativo per salvare lo stato di esecuzione di un *thread* o un processo, in modo da poter essere ripreso in un momento successivo.



(a) RSA version (b) ECDSA version

Figura 6.7: Average classical PoW attempts



(a) RSA version (b) ECDSA version

Figura 6.8: Average classical PoW time

6.5.3 Difficulty base

Nei test effettuati si nota che la scelta di $k_{base} = 3$ per la versione RSA e $k_{base} = 6$ per la versione ECDSA garantisce un tempo di creazione dei blocchi omogeneo al variare del numero di transazioni. Di seguito si vuole illustrare come sono stati ottenuti questi valori applicando la formula (6.5). Sfruttando i dati dei test, il rapporto $\frac{t_{clear}}{t_{hash}}$ è stato stimato nel seguente modo:

$$\frac{t_{clear}}{t_{hash}} \approx \frac{\overline{T}_{clear}}{\overline{A}_{clear}} \cdot \left(\frac{\overline{T}_{cPoW}}{\overline{A}_{cPoW}} \right)^{-1} \tag{6.9}$$

dove $\overline{T}_{clear}$ è il tempo medio di *clearing*, $\overline{T}_{cPoW}$ è il tempo medio della PoW classica, $\overline{A}_{clear}$ è il numero medio di tentativi impiegati per il *clearing* e $\overline{A}_{cPoW}$ è il numero

medio di tentativi impiegati per la PoW classica.
Per il test 1 ad esempio si ha che $\frac{t_{clear}}{t_{hash}} \approx \frac{2.13}{3814} \cdot \left(\frac{1.267}{17151}\right)^{-1} \approx 7,56$ da cui $k_{base} = 3$. Il medesimo valore di k_{base} si ottiene anche per i test 2-5, relativi alla versione RSA. Per il test 6 $\frac{t_{clear}}{t_{hash}} \approx \frac{3.616}{1001} \cdot \left(\frac{2.39}{31657}\right)^{-1} \approx 47,85$ da cui $k_{base} = 6$. Il medesimo valore di k_{base} si ottiene anche per i test 7-10, relativi alla versione ECDSA.

6.5.4 Fair e evil miners

Per confrontare le prestazioni di *fair* e *evil miners* sono stati eseguiti i test 11-14. Si ricorda che gli *evil miners* traggono vantaggio dalla possibilità di scartare la fase CRYPTO in un tentativo di *clearing*. Una misura del vantaggio degli *evil miners* è data dal rapporto PRNG/CRYPTO, illustrato nei grafici aggiuntivi per il test 1 (RSA) e il test 6 (ECDSA). Nella versione RSA il rapporto $\frac{t_{PRNG}}{t_{CRYPTO}} \approx 3$, corrispondente al valore del parametro pubblico X, nella versione ECDSA il rapporto $\frac{t_{PRNG}}{t_{CRYPTO}} \approx 64$. In altri termini nei test effettuati, un *evil miner* RSA è mediamente il 33% più veloce di un *fair miner* RSA, mentre un *evil miner* è l'1,6% più veloce di un *fair miner* ECDSA. A differenza di RSA, nella versione ECDSA gli *evil miners* non sono molto avvantaggiati.
I test 11 (ECDSA) e 13 (RSA) simulano 6 *evil miners* simultanei: confrontando i risultati ottenuti rispettivamente con i test 9 e 4, si possono corroborare le precedenti considerazioni. I test 12 (ECDSA) e 14 (RSA) simulano una *miner pool* costituita da 2 *evil* e 4 *fair miners*: nel caso RSA gli *evil miners*, nonostante siano numericamente inferiori, vantano un maggior numero di blocchi "minati" e un minor tempo di creazione dei blocchi rispetto ai *fair miners* proprio a causa del loro considerevole vantaggio.

Capitolo 7

Conclusioni

Il lavoro di tesi si è concluso con la realizzazione di un simulatore blockchain in Python3 funzionante che implementa i protocolli di firma digitale *fuzzy* RSA e ECDSA. Il maggior contributo al progetto riguarda l'adozione di alcune scelte implementative per la simulazione dei protocolli all'interno di un'applicazione *standalone*, come ad esempio il multiprocessing, che ha permesso di riprodurre il *mining* in parallelo di molteplici nodi. Un altro spazio di approfondimento concerne l'analisi delle differenze tra le performance previste dai modelli di firma digitale *fuzzy* e le vere prestazioni del simulatore; questo studio ha condotto a una rettifica del parametro di difficoltà nella PoW classica. I risultati illustrati nella trattazione sono stati ottenuti da numerosi test eseguiti con il programma.
La ricerca svolta non esaurisce gli spunti di riflessione sull'argomento. Ne possiamo individuare alcuni ancora da esplorare:

- introdurre il parametro di *artificial fuzzyness* [2] per regolare la complessità computazionale dell'operazione di *clearing*
- progettare e realizzare un'applicazione blockchain distribuita dei protocolli *fuzzy* RSA e ECDSA. Questa soluzione, molto più vicina a un'infrastruttura blockchain reale, supererebbe le limitazioni di un software *standalone* e permetterebbe la simulazione di un maggior numero di miners
- utilizzare dati biometrici misurati da un sensore reale al posto della generazione pseudo-casuale. L'utilizzo dei protocolli in casi pratici potrebbe suggerire modifiche implementative
- ricodificare il simulatore in un linguaggio di programmazione compilato (C, Rust) consentirebbe di porre maggiore attenzione nelle prestazioni, garantendo esecuzioni più veloci e ottimizzate

Appendice A

Appendice

A.1 Codice sorgente

Classe tools

```
from Crypto.Hash import keccak
from time import perf_counter

class tools:

    '''Defines some useful functions'''

    @classmethod
    def hash(cls, m : str, db : int = 256):
        '''Hashes given message'''
        d = keccak.new(digest_bits=db)
        d.update(str(m).encode('utf8'))
        return d.hexdigest()

    @classmethod
    def hashint(cls, m : str, db : int = 256):
        '''Hashes given message and converts it to integer'''
        return int( cls.hash(m, db), 16)

    def crono(func):
        '''Implements a decorator for timing functions'''
        def inner(*args, **kwargs):
            start = perf_counter()
            x = func(*args, **kwargs)
            end = perf_counter()
            return (end - start, ) + x
        return inner
```

Metodo simulate() della classe blockchain

```
@tools.crono
    def simulate(self) ->None:

        '''Simulates blockchain'''

        public_file = self.user_pool.get_public_file()
```

```
        r_blocks = [ ceil(self.n_blocks/self.max_tx) for _ in range(
    self.max_tx) ]

        for block_number in range(1,self.n_blocks+1):
            print("Creating block {}/{}".format(block_number, self.
    n_blocks), end = " ... ")
            aux = self.retrieve_aux()
            flag = False

            while(not flag):
                n_tx = randrange(1, self.max_tx+1)
                flag = r_blocks[n_tx-1]

            r_blocks[n_tx-1] -= 1

            raw_txs = self.user_pool.sim_txs(n_tx)
            p = packet(block_number, aux, raw_txs, public_file)

            block, stats = self.miner_pool.start_mining(p)
            self.blocks.append(block)
            self.report.put_stats(stats)
            print("Block created")

        return ()
```

Classe sync_buffer

```
from multiprocessing import Queue, Semaphore

from core.datastruct.packet import packet

class sync_buffer:

    '''
        Implements producer-consumer logic for miner process
    synchronization.
    '''

    def __init__(self) -> None:
        '''
        Setups synchronization primitives and queues.
        Creates semaphores free and item.
        Creates four queues for blocks, packets, stats and logs.
        '''
        self.mined_queue = Queue()
        self.packet_queue = Queue()
        self.stats_queue = Queue()
        self.logs_queue = Queue()

        self.free = Semaphore(1)
        self.item = Semaphore(0)
```

```
    def send_packets(self, packet : packet, n : int) -> None:
        '''
        Inserts packets in packet queue.
        '''
        for _ in range(n):
            self.packet_queue.put(packet)

    def before_consume(self) -> tuple:
        '''
            Miner pool gets block, stats and logs from respective
    queues
        '''
        self.item.acquire()
        block = self.mined_queue.get()
        stats = self.stats_queue.get()
        logs = []
        while not self.logs_queue.empty():
                logs.append(self.logs_queue.get())
        return block, stats, logs

    def after_consume(self) -> None:
        '''
            Miner pool releases queues
        '''
        self.free.release()

    def get_packet(self) -> None:
        '''Miner gets packet from packet queue'''
        return self.packet_queue.get()

    def put_log(self, log) -> None:
        '''Miner puts log into logs queue'''
        self.logs_queue.put(log)

    def before_produce(self) -> None:
        '''Miner acquires block queue'''
        self.free.acquire()

    def after_produce(self, block, block_stats) -> None:
        '''Miner puts block and stats into respective queues and
    releases them'''
        self.mined_queue.put(block)
        self.stats_queue.put(block_stats)
        self.item.release()
```

Classe miner

```
from abc import abstractmethod
from multiprocessing import Process
from random import seed

from core.stats.block_stats import block_stats
```

```
from core.datastruct.public_key import public_key
from core.datastruct.packet import packet
from core.datastruct.signature import signature
from core.mining.miner_flyweight import miner_flyweight as mf
from core.mining.sync_buffer import sync_buffer
from core.util.tools import tools

class miner(Process):

    '''Defines a miner process.'''

    @abstractmethod
    def compute_verifying(self, message: str, signature : signature,
    public_key : public_key):
        '''Makes some computations needed by clearing process'''
        pass

    @abstractmethod
    def clear_attempt(self, message: str, aux : str, ver,
                      public_file : list[public_key], public_key:
    public_key,
                      PoW : str) -> tuple:
        '''Makes a fair clearing attempt'''
        pass

    @abstractmethod
    def prng_phase(self, message : str, aux : str, public_key :
    public_key, PoW : str) -> tuple:
        '''Executes PRNG phase'''
        pass

    @abstractmethod
    def crypto_phase(*args, **kwargs) -> tuple[int]:
        '''Executes CRYPTO phase'''
        pass

    def  __init__(self, id : int, type : str, evil : bool, sbuffer :
    sync_buffer) -> None:

        '''Creates a miner process'''

        super(miner,self).__init__()

        self.id = id
        self.type = type
        self.sbuffer = sbuffer
        self.evil = evil
        self.desc = "Evil" if evil else "Fair"

    def run(self) -> None:
```

```
        '''Executes miner's task'''
        seed(self.id)
        packet = self.sbuffer.get_packet()
        bs = block_stats(packet.raw_txs)
        seed_values = []
        num_tx = len(packet.raw_txs)

        for i in range(num_tx):

            eta, eta_prng, eta_crypto, n_attempts, test_seed = self.
    clear_tx(packet, i)

            seed_values.append(test_seed)

            bs.tx_times.append(eta)
            bs.tx_attempts.append(n_attempts)
            bs.prng_times.append(eta_prng)
            bs.crypto_times.append(eta_crypto)

            msg = "[{} Miner:{}]".format(self.desc, self.id) + "
    Transaction #{} mined".format(i)

            self.sbuffer.put_log(msg)

        eta, n_attempts, b = mf.classical_pow(packet, seed_values)


        bs.cpow_attempts = n_attempts
        bs.cpow_time = eta
        bs.evil = self.evil
        bs.num_tx = num_tx
        bs.miner_id = self.id

        msg = "[{} Miner:{}] Classical Pow finished".format(self.desc,
     self.id)
        self.sbuffer.put_log(msg)

        self.sbuffer.before_produce()
        bs.raw_block = str(b)
        self.sbuffer.after_produce(b, bs)

    @tools.crono
    def clear_tx(self, packet : packet, tx_id : int) -> tuple:

        '''Clear a raw transaction'''

        n_attempts = 0
        eta_prng = 0
        eta_crypto = 0

        message = packet.raw_txs[tx_id].message
```

```
        aux = packet.aux
        public_file = packet.public_file
        user_id = packet.raw_txs[tx_id].user_id
        public_key = public_file[user_id]

        flag_cleared = 0

        if self.evil:
            clearing = packet.raw_txs[tx_id].clearing

            while flag_cleared == 0:
                n_attempts += 1;
                prng_time, flag_cleared, test_seed = self.
    evil_clear_attempt(message, aux, clearing);
                eta_prng += prng_time
        else:
            signature = packet.raw_txs[tx_id].signature
            ver = self.compute_verifying(message,signature, public_key
    )
            while flag_cleared == 0:
                n_attempts += 1;
                prng_time, crypto_time, flag_cleared, test_seed = self
    .clear_attempt(message,aux,ver,public_file, public_key);
                eta_prng += prng_time
                eta_crypto += crypto_time

        return eta_prng, eta_crypto, n_attempts, test_seed

    def evil_clear_attempt(self, message : str, aux: str, clearing :
    int) -> tuple:
        '''Makes an evil clearing attempt'''
        eta_prng, test_seed, test_delta_e = self.prng_phase(message,
    aux)
        ok = mf.auth(test_delta_e, clearing, test_delta_e)
        return eta_prng, ok, test_seed;
```

A.2 Test

Test 1

```
Settings
Protocol type: RSA
Number of miners: 1
Number of evil miners: 0
w parameter: 512
Base CPoW difficulty: 3

Stats
Elapsed time: 0:17:03.263259
avg clearing time: 2.13
avg cpow time: 1.267
avg clearing attempts: 3814
avg cpow attempts: 17151
```

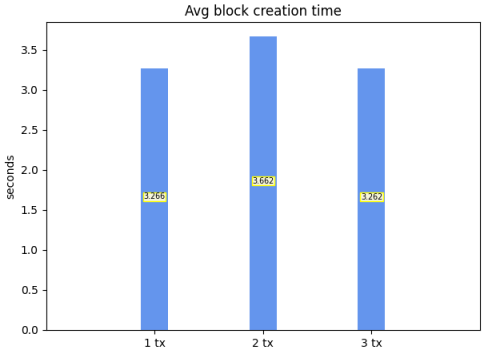


Figura A.1: Grafico test 1

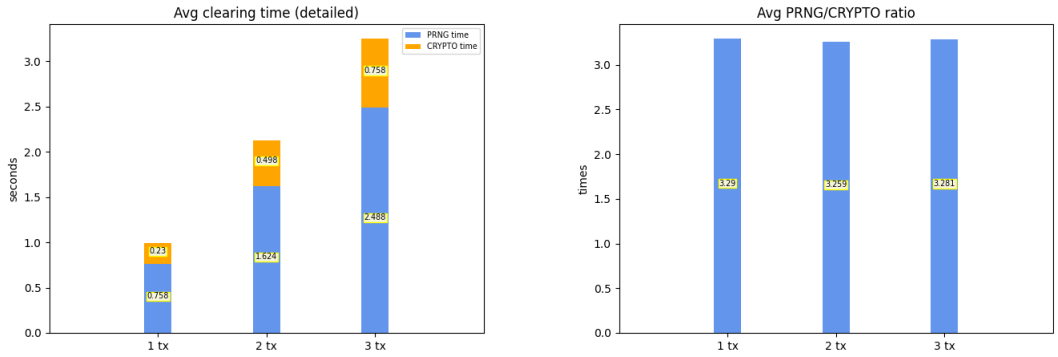


Figura A.2: Grafici aggiuntivi Test 1

Test 2

```
Settings
Protocol type: RSA
Number of miners: 2
Number of evil miners: 0
w parameter: 512
Base CPoW difficulty: 3

Stats
Elapsed time: 0:11:40.090137
avg clearing time: 1.686
avg cpow time: 0.631
avg clearing attempts: 2989
avg cpow attempts: 8373
```

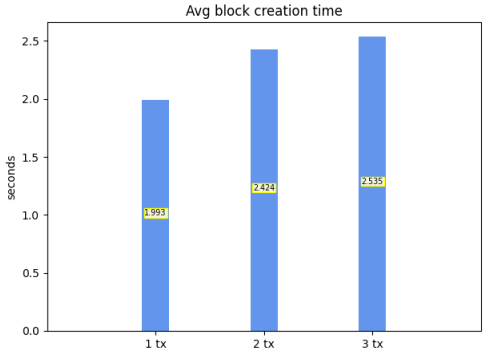


Figura A.3: Grafico test 2

Test 3

```
Settings
Protocol type: RSA
Number of miners: 4
Number of evil miners: 0
w parameter: 512
Base CPoW difficulty: 3

Stats
Elapsed time: 0:08:16.773932
avg clearing time: 1.152
avg cpow time: 0.48
avg clearing attempts: 1997
avg cpow attempts: 6169
```

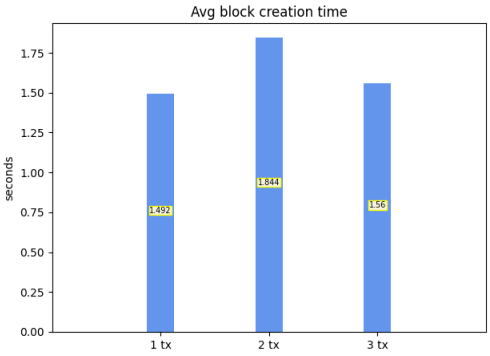


Figura A.4: Grafico test 3

Test 4

```
Settings
Protocol type: RSA
Number of miners: 6
Number of evil miners: 0
w parameter: 512
Base CPoW difficulty: 3

Stats
Elapsed time: 0:07:24.984275
avg clearing time: 1.08
avg cpow time: 0.366
avg clearing attempts: 1770
avg cpow attempts: 4434
```

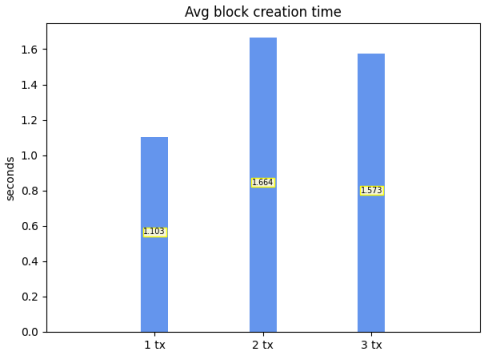


Figura A.5: Grafico test 4

Test 5

```
Settings
Protocol type: RSA
Number of miners: 8
Number of evil miners: 0
w parameter: 512
Base CPoW difficulty: 3

Stats
Elapsed time: 0:08:35.607701
avg clearing time: 1.171
avg cpow time: 0.473
avg clearing attempts: 1515
avg cpow attempts: 4575
```

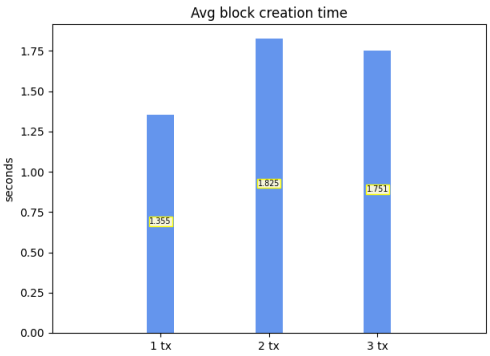


Figura A.6: Grafico test 5

Test 6

```
Settings
Protocol type: ECDSA
Number of miners: 1
Number of evil miners: 0
w parameter: 128
Base CPoW difficulty: 6

Stats
Elapsed time: 0:30:09.764101
avg clearing time: 3.616
avg cpow time: 2.39
avg clearing attempts: 1001
avg cpow attempts: 31657
```

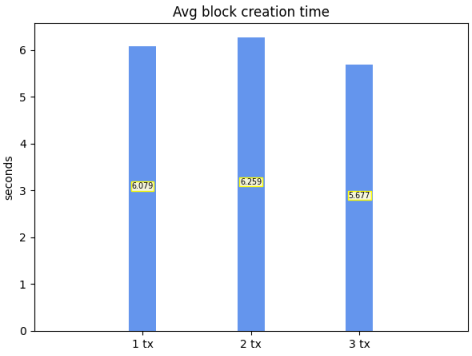


Figura A.7: Grafico test 6

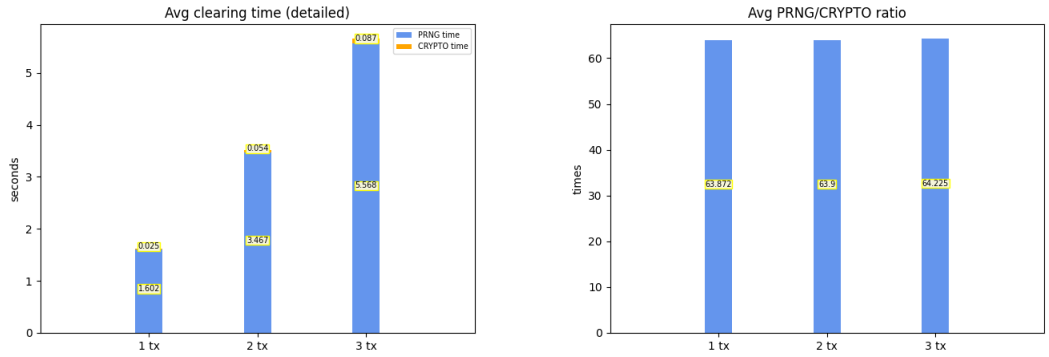


Figura A.8: Grafici aggiuntivi Test 6

Test 7

```
Settings
Protocol type: ECDSA
Number of miners: 2
Number of evil miners: 0
w parameter: 128
Base CPoW difficulty: 6

Stats
Elapsed time: 0:19:49.348401
avg clearing time: 2.67
avg cpow time: 1.264
avg clearing attempts: 729
avg cpow attempts: 16177
```

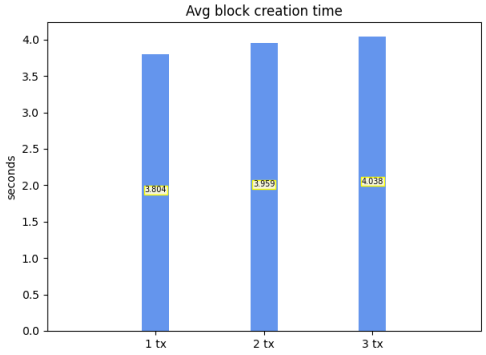


Figura A.9: Grafico test 7

Test 8

```
Settings
Protocol type: ECDSA
Number of miners: 4
Number of evil miners: 0
w parameter: 128
Base CPoW difficulty: 6

Stats
Elapsed time: 0:15:09.350545
avg clearing time: 1.962
avg cpow time: 1.028
avg clearing attempts: 525
avg cpow attempts: 12775
```

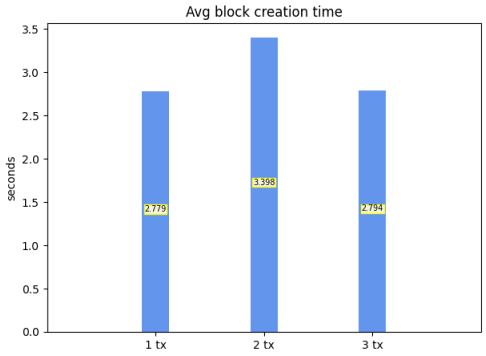


Figura A.10: Grafico test 8

Test 9

```
Settings
Protocol type: ECDSA
Number of miners: 6
Number of evil miners: 0
w parameter: 128
Base CPoW difficulty: 6

Stats
Elapsed time: 0:11:53.421454
avg clearing time: 1.64
avg cpow time: 0.684
avg clearing attempts: 417
avg cpow attempts: 8084
```

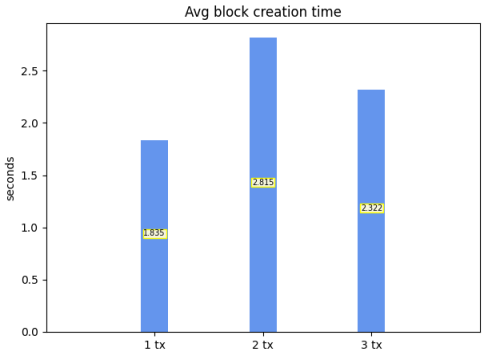


Figura A.11: Grafico test 9

Test 10

```
Settings
Protocol type: ECDSA
Number of miners: 8
Number of evil miners: 0
w parameter: 128
Base CPoW difficulty: 6

Stats
Elapsed time: 0:15:02.243048
avg clearing time: 2.137
avg cpow time: 0.771
avg clearing attempts: 418
avg cpow attempts: 6807
```

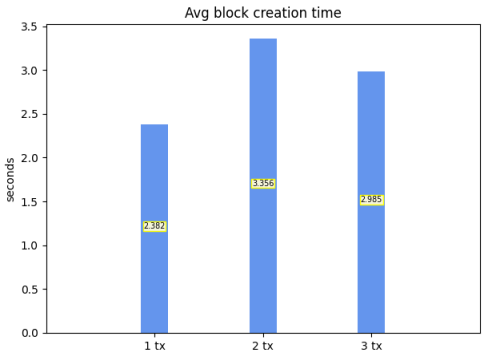


Figura A.12: Grafico test 10

Test 11

```
Settings
Protocol type: ECDSA
Number of miners: 6
Number of evil miners: 6
w parameter: 128
Base CPoW difficulty: 6

Stats
Elapsed time: 0:12:30.192630
```

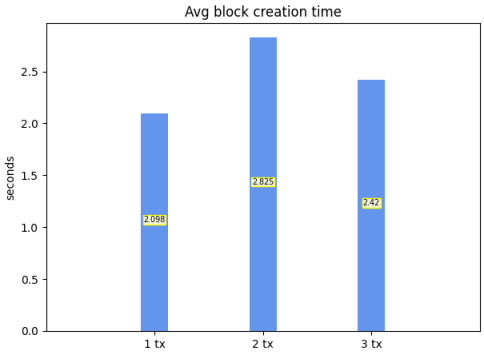


Figura A.13: Grafico test 11

Test 12

```
Settings
Protocol type: ECDSA
Number of miners: 6
Number of evil miners: 2
w parameter: 128
Base CPoW difficulty: 6

Stats
Elapsed time: 0:13:42.334171
```

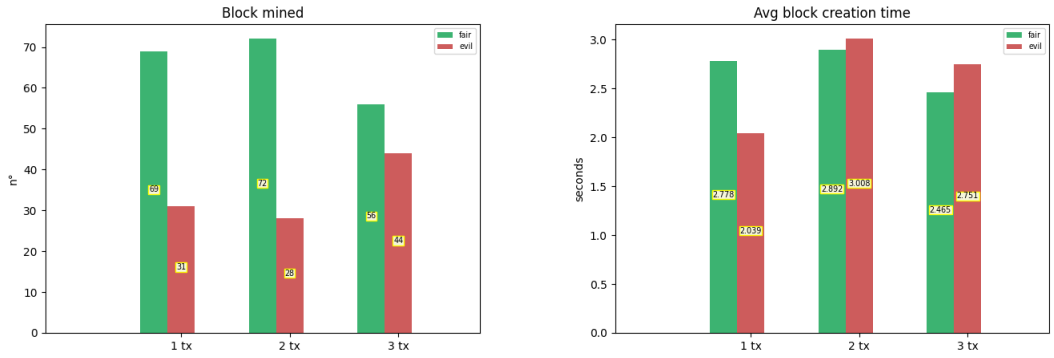


Figura A.14: Grafici Test 12

Test 13

```
Settings
Protocol type: RSA
Number of miners: 6
Number of evil miners: 6
w parameter: 512
Base CPoW difficulty: 3

Stats
Elapsed time: 0:06:03.574057
```

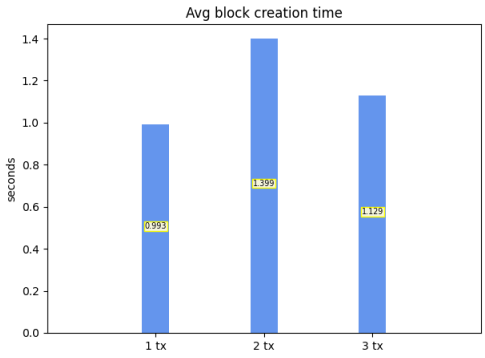


Figura A.15: Grafico test 13

Test 14

```
Settings
Protocol type: RSA
Number of miners: 6
Number of evil miners: 2
w parameter: 512
Base CPoW difficulty: 3

Stats
Elapsed time: 0:07:52.368681
```

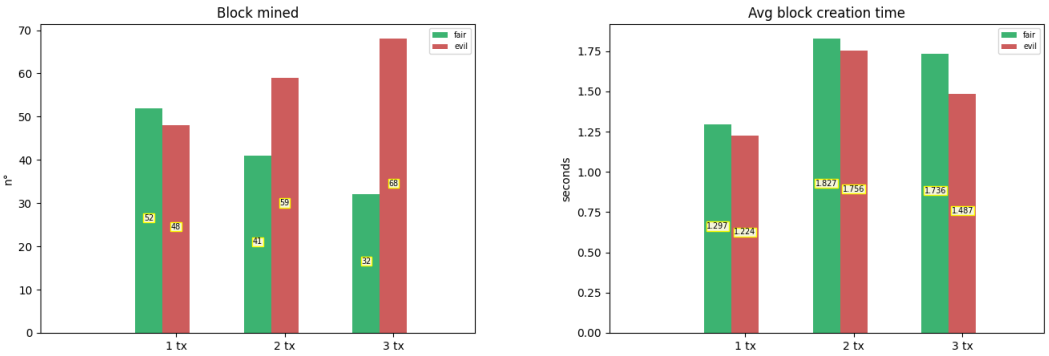


Figura A.16: Grafici Test 14

Bibliografia

[1] Wikipedia. *Multi-factor Authentication*. https://en.wikipedia.org/wiki/Multi-factor_authentication.

[2] P. Santini, G. Rafaiani, M. Battaglioni, M. Baldi, and F. Chiaraluce. "*Working to Clear Fuzzy Signatures: Blockchain Protocols for Biometric Identification and Authentication*". preprint.

[3] Anil K. Jain, Arun Ross, and Salil Prabhakar. "*An Introduction to Biometric Recognition*". 2004.

[4] Rahul Kumar Jaiswal and Gaurav Saxena. "*Biometric Recognition System (Algorithm)*". 2018.

[5] Kenta Takahashi, Takahiro Matsuda, Takao Murakami, Goichiro Hanaoka, and Masakatsu Nishigaki. "*Signature Schemes with a Fuzzy Private Key*". 2017.

[6] O. Delgado-Mohatar, J. Fierrez, R. Tolosana, and R. Vera-Rodriguez. "*Blockchain and biometrics: A first look into opportunities and challenges*". 2019.

[7] K. Nandakumar, N. Ratha, S. Pankanti, and S. Darnell. "*Secure one-time biometric tokens for non-repudiable multi-party transactions*". 2006.

[8] R.L. Rivest, A. Shamir, and L. Adleman. "*A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*". 1978.

[9] Don Johnson, Alfred Menezes, and Scott Vanstone. "*The Elliptic Curve Digital Signature Algorithm (ECDSA)*". 2001.

[10] Bernhard Linke. "*The Fundamental of an ECDSA Authentication System*". 2014.

[11] Adam Hayes. *Blockchain Facts: What Is It, How It Works, and How It Can Be Used*. https://www.investopedia.com/terms/b/blockchain.asp, 2022.

[12] Satoshi Nakamoto. "*Bitcoin: A Peer-to-Peer Electronic Cash System*". 2008.

[13] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. "*An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends*". 2017.

[14] Leslie Lamport, Robert Shostak, and Marshall Pease. "*The Byzantine Generals Problem*". 1982.

[15] *Cryptographically secure pseudorandom number generator*. https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator.

[16] Adam Back. "*Hashcash - A Denial of Service Counter-Measure*". 2002.

[17] *Flyweight*. https://refactoring.guru/design-patterns/flyweight.

[18] *Difference Between Multithreading vs Multiprocessing in Python*. https://www.geeksforgeeks.org/difference-between-multithreading-vs-multiprocessing-in-python/, 2020.

[19] *Decorator Design Pattern in Java with Example*. https://www.geeksforgeeks.org/decorator-design-pattern-in-java-with-example/, 2022.

[20] *Commutazione di contesto*. https://it.wikipedia.org/wiki/Commutazione_di_contesto.