UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

Implementazione su piattaforma ARM Cortex-M4 di funzionalità blockchain conformi con Ethereum Beacon Chain

Implementation on ARM Cortex-M4 platform of Ethereum Beacon Chain-compliant blockchain functions

Candidato:
Valeria Vetrano

Relatore:
Chiar.mo Prof. Marco Baldi

Correlatore:
Dott. Paolo Santini

Anno Accademico 2021-2022

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

Implementazione su piattaforma ARM Cortex-M4 di funzionalità blockchain conformi con Ethereum Beacon Chain

Implementation on ARM Cortex-M4 platform of Ethereum Beacon Chain-compliant blockchain functions

Candidato:
Valeria Vetrano

Relatore:
Chiar.mo Prof. Marco Baldi

Correlatore:
Dott. Paolo Santini

Anno Accademico 2021-2022

Indice

INTRODUZIONE

I recenti sviluppi tecnologici dimostrano che il mercato dell'Internet of Things (IoT) a partire da un semplice concetto sia diventato uno dei più potenti meccanismi di industrializzazione e non solo.
Per definizione l'IoT descrive la rete di oggetti fisici, ossia di "cose", incorporati con sensori, software e altre tecnologie allo scopo di connettere e scambiare dati con altri dispositivi e sistemi sfruttando la rete Internet [1].
Grazie alla fattibilità del collegamento a Internet e l'integrazione con il cloud computing, i big data e l'apprendimento automatico, è possibile una comunicazione senza soluzione di continuità tra persone, processi e cose, realizzando servizi digitali incentrati sui dati. Questi dispositivi spaziano dai normali oggetti domestici alle piattaforme di sviluppo hardware sino ai sofisticati strumenti industriali. La versatilità che li contraddistingue unita alla miniaturizzazione elettronica li rende protagonisti di scenari come smart city, assistenza sanitaria, ambito militare e agricoltura.

La connessione di un ecosistema IoT così automatizzato, in cui i device comunicano e scambiano dati attraverso internet senza la necessità dell'intervento umano, purtroppo rende questi "oggetti" vulnerabili ad attacchi esterni. Considerando che i dati vengono immagazzinati in server centralizzati ci potrebbe essere il rischio che vi sia un "punto di fallimento", dunque non vi sia una garanzia di disponibilità del servizio e conseguentemente di integrità e autenticità dei dati, che potrebbero essere soggetti a manomissione.
Inoltre, il trade-off tra i big data che essi producono, in aumento esponenziale, e l'essere "lightweight devices", ovvero troppo limitati di risorse, rende complicato applicare in modo efficiente complesse politiche di sicurezza e privacy dei dati.

Una tra le soluzioni di maggior rilievo, per ovviare ai limiti esposti, prevede l'incorporazione di tecnologie di registro distribuito, come la blockchain, nei dispositivi IoT[2]. Proprio per sua intrinseca natura, la tecnologia blockchain permetterebbe la gestione e lo scambio dei dati in via decentralizzata.

Usufruendo della tecnologia blockchain si potrebbero superare i limiti di cui discusso, purché i dispositivi possano interagire con la blockchain mediante l'implementazione di funzionalità conformi alle possibilità da loro offerte e alle regole della rete peer-to-peer.

Il seguente lavoro mira a sviluppare e testare un'implementazione che permetta di combinare la tecnologia blockchain con una piattaforma hardware dedicata, attraverso la generazione di strutture dati, denominate transazioni.
Questo potrebbe consentire al dispositivo embedded, posto in uno scenario specifico, quale ad esempio quello delle *Supply Chain*, di acquisire i dati da sensori ad esso connessi e organizzare questi dati sotto forma di semplici transazioni. Queste ultime verrebbero poi trasmesse ad un nodo più potente sulla rete che potrà convalidarle e registrarle.

Si vuol conferire al dispositivo la capacità di implementare funzionalità conformi con Ethereum Beacon Chain, a fronte della recente variazione del protocollo del consenso, passato dal Proof of Work al Proof of Stake.
Il dispositivo su cui si è sviluppato il lavoro è un microcontrollore STM32F439ZI basato su core ARM CORTEX-M4, idoneo per le applicazioni IoT grazie al suo equilibrio tra basso consumo energetico, alta efficienza e prestazioni [3].

La trattazione è organizzata nella seguente forma:

- Nel capitolo 1 viene presentata la tecnologia blockchain, dunque gli aspetti che la contraddistinguono e le sue caratteristiche di base; vengono approfonditi i meccanismi di consenso e i vantaggi che si riscontrano a fronte dell'interazione tra i dispositivi IoT e la tecnologia in questione.

- Nel capitolo 2 la discussione si concentra sulla blockchain Ethereum in particolare, esponendo in quali modalità e secondo quali regole e formati vengono generate le transazioni. Inoltre, vengono analizzati i cambiamenti sulla rete conseguenti alla variazione del meccanismo di consenso.

- Il capitolo 3 è dedicato alla presentazione degli strumenti hardware e software di cui si è usufruito per l'implementazione; viene illustrata la modalità di interazione tra il device scelto e la blockchain Ethereum, partendo dalla scelta del nodo.

- Il capitolo 4 comprende la fase di benchmarking, ovvero di analisi e valutazione delle performance del dispositivo, prima in relazione alla libreria *micro-ecc* e successivamente considerando la generazione delle transazioni.

- Infine, nel capitolo 5 vengono studiate le tempistiche relative alla trasmissione dei formati di transazione in esame verso la rete e posti i risultati a confronto.

Capitolo 1

LA TECNOLOGIA BLOCKCHAIN

1.1 Dalle DLT alle Blockchain

Le soluzioni tecnologiche definite come Blockchain, traduzione letteraria che sta per "catena di blocchi", rientrano nell'ambito delle *Distributed ledger technology* (DLT). Nello specifico, il DLT è un database condiviso, replicato e sincronizzato tra i membri di una rete decentralizzata (rete peer-to-peer).
Questo registro distribuito memorizza le varie transazioni, come lo scambio di beni o dati, tra i partecipanti della rete.

Ogni DLT differisce per:

1. Struttura dati.
2. Protocollo che permette la registrazione delle transazioni.
3. Tipo di algoritmo del consenso.

Bitcoin[1], ha per primo usufruito di una struttura dati che prende il nome di blockchain, per implementare le DLT.
Una blockchain è un registro contabile digitale condiviso a prova di manomissione che registra le transazioni in una rete peer-to-peer di device, pubblica o privata. Distribuito tra tutti i nodi della rete, il libro registra in maniera sequenziale i blocchi crittografici e la storia delle transazione che avvengono tra i nodi della rete (Figura 1.1).

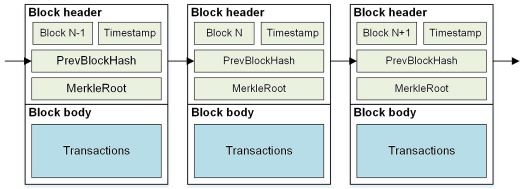


Figura 1.1: Struttura dati di una blockchain [5].

Quindi, una blockchain è una lista concatenata che presenta alcune peculiarità:

- Ogni nuovo blocco che viene aggiunto alla catena presenta un puntatore al blocco precedente. L'unico blocco a non presentare un puntatore è il blocco da cui parte la catena, ovvero blocco di *genesi*.
- Il puntatore è un hash[2] del blocco precedente.
- All'interno di ogni blocco è presente un *timestamp*, che identifica quando il blocco è stato creato ed un *nonce*, un valore pseudocasuale che è legato univocamente al blocco.

[1] Il Bitcoin è una criptovaluta e un sistema di pagamento valutario internazionale creato nel 2009 da un anonimo inventore (o gruppo di inventori), noto con lo pseudonimo di Satoshi Nakamoto. Per convenzione se il termine Bitcoin è utilizzato con l'iniziale maiuscola si riferisce alla tecnologia e alla rete, mentre se minuscola (bitcoin) si riferisce alla valuta in sé.[4]

[2] Un hash è una funzione crittografica che mappa una stringa di dimensioni arbitrarie in un'altra stringa di lunghezza fissata, in modo non invertibile

- Infine, sono presenti le varie transazioni, cui trattazione è rimandata ai capitoli successivi.

E' importante sottolineare che i dati di un blocco non possono cambiare se prima non vengono modificati tutti i blocchi successivi; questo richiederebbe il consenso dell'intera rete.

Con lo sviluppo e la diffusione della blockchain Bitcoin è aumentata la necessità di implementare un linguaggio di scripting, che fosse più completo e generale di quello di Bitcoin, capace di supportare lo sviluppo di applicazioni decentralizzate. Questo desiderio ha contribuito alla nascita della rete Ethereum, piattaforma open source basata su blockchain. L'obiettivo del progetto risiedeva nell'esecuzione di *Smart Contract*. Questi ultimi sono programmi/algoritmi general purpose, immutabili e non repudiabili, che consentono di facilitare, verificare o imporre la negoziazione e l'esecuzione di un contratto in assenza di terze parti.

1.2 Meccanismi di Consenso

Come accennato nel paragrafo precedente, le blockchain necessitano di un meccanismo di consenso, per determinare l'accordo distribuito, tra i partecipanti. Questo in virtù del fatto che, ogni device nella rete deve acconsentire ad ogni nuovo blocco, in cui vi sono le transazioni e alla catena nel complesso. Quando vengono generate delle transazioni da un nodo, affinché possano essere inserite nel registro, devono essere validate dai restanti peer della rete.

Vi sono diversi algoritmi di consenso, tuttavia dato lo scopo dell'elaborato verranno trattati solo il Proof of Work (PoW) e il Proof of Stake (PoS). Entrambi hanno lo scopo di aiutare a raggiungere l'approvazione della maggioranza tra i membri, ma con meccanismi differenti.

1.2.1 Proof of Work

Il termine identifica una "prova di lavoro" in cui tutti i nodi partecipanti della rete, definiti *miners*, competono tra loro nella risoluzione di un puzzle crittografico,ovvero nel calcolo di una funzione hash, dove ognuno usufruisce delle proprie risorse computazionali.
Il vincitore della prova diventa il *leader*, può quindi inviare il blocco, contenente le transazioni, mediante un meccanismo di *gossip*, all'intera rete e ottenere come ricompensa una nuova moneta coniata (es. Bitcoin). Il blocco viene aggiunto definitivamente nella blockchain solo se la maggioranza (>50%) dei partecipanti ritiene che sia valido.

Questo meccanismo ha dimostrato la sua fattibilità dal punto di vista della sicurezza e affidabilità della rete. Per commettere frodi sulle transazioni in una blockchain, un soggetto dovrebbe possedere il 51% dell'intera rete, il che non è realistico. Inoltre,

all'aumentare del valore di una criptovaluta, sempre più miners sono incentivati a unirsi alla rete, determinando un incremento del numero dei partecipanti con conseguente riduzione della probabilità che i nodi si accordino per raggiungere il consenso.

Per usufruire di un'ingente quantità di risorse computazioni sufficienti a svolgere il mining servono dei cluster di server e questo richiede un grande dispendio energetico. Inoltre, bisogna considerare che una transazione, prima di essere trascritta in un blocco, necessita di un'enorme quantità di tempo, dunque l'efficienza che ne consegue è minima. Tali limitazioni e svantaggi vengono sopperiti grazie al Proof of Stake.

1.2.2 Proof of Stake

Se nel PoW i miner sfruttano le risorse energetiche per la creazione di nuovi blocchi, nel PoS i *Validators* pongono in staking, attraverso una transazione, una somma prestabilita sotto forma di criptovaluta, per proporre nuovi blocchi. Quando la transazione viene confermata, il validatore può puntare alcune monete per competere con gli altri. Nel frattempo, ogni nodo è responsabile della trasmissione delle transazioni ricevute. Quando viene creata una quantità sufficiente di transazioni, i validatori eleggono un leader con il massimo numero di monete puntate. Il leader eletto crea quindi un blocco e lo trasmette alla rete. Ogni nodo convalida il blocco, esegue tutte le transazioni del blocco e aggiunge il blocco alla catena. Il blocco ha anche una speciale transazione di ricompensa. Infine, il leader del turno riceve come ricompensa le commissioni delle transazioni presenti nel blocco [6].

Lo staking rende più semplice la partecipazione al meccanismo, perché si focalizza sulla quantità di moneta messa a disposizione dai partecipanti. Questo incrementa la decentralizzazione del network. Per evitare che il leader sia chi, banalmente, possiede più risorsa economica, i titolari delle monete vengono scelti mediante un approccio deterministico, per l'appunto lo staking. Fondamentale vantaggio sta nell'evitare lo spreco di risorse energetiche, come invece avviene con il PoW.

I modelli PoS potrebbero risultare meno sicuri perchè di divulgazione più recente, inoltre potrebbero essere più vulnerabili al cosiddetto *bribe attack*, ovvero l'attacco tangente. Si verifica quando un attaccante esegue una transazione che intende annullare a posteriori. Non appena la transazione viene approvata procede al fork della blockchain sulla base dell'ultimo blocco verificato prima della transazione.
L'attaccante continua poi a costruire la catena fino a quando non è più lunga dell'originale. Una volta raggiunto questo traguardo, l'attaccante pubblica la blockchain nel suo complesso, accettata come valida dalla rete, e inverte la transazione iniziale dell'attacco[6].

Nonostante ciò, grazie ai vantaggi legati alla customizzazione, alle prestazioni, al costo e alla sicurezza cripto-economica, attualmente,viene considerato il meccanismo di consenso che richiede minori barriere tecniche per partecipare alla rete, i nodi sono più distribuiti e di conseguenza la sicurezza è maggiore. Per tali ragioni, sta via via sostituendo il Proof of Work.

1.3 IoT e Blockchain

I dispositivi IoT stanno riformando e semplificando i flussi di lavoro standard in flussi di lavoro digitalizzati e completamente automatizzati. La vasta mole di dati che essi producono permette di sviluppare applicazioni intelligenti, come il miglioramento dei servizi nelle smart city o il miglioramento della gestione e della qualità della vita per l'uomo. L'integrazione della tecnologia di cloud computing ha contribuito a funzioni essenziali dell'IoT per l'analisi e l'elaborazione dei dati, trasformandoli in azioni e conoscenze in tempo reale. Nonostante il contributo del cloud computing si sia dimostrato inestimabile, una delle vulnerabilità più importanti scaturite da questa integrazione consiste nella mancanza di fiducia, causata dall'architettura centralizzata, di cui consta questa tecnologia. I partecipanti alla rete non hanno una visione chiara di dove e come utilizzare le informazioni che forniscono.

La tecnologia Blockchain, grazie alla sua natura decentralizzata, può offrire, oltre a servizi di condivisione affidabili, garanzie di sicurezza come: integrità, autenticità e tracciabilità dei dati. Dunque, combinare la tecnologia blockchain con il mondo IoT può rappresentare indubbi vantaggi, in diversi scenari.
Ad esempio, la tracciabilità approfondita in molti prodotti alimentari è un fattore chiave per garantire la sicurezza alimentare. La tracciabilità degli alimenti può richiedere la partecipazione di un gran numero di partecipanti dedicati alla produzione, alimentazione, trattamento, distribuzione, ecc. Una perdita in qualsiasi parte della catena potrebbe portare a una violazione e rallentare il processo di ricerca di prodotti contaminati, che potrebbero essere causa di infezioni. Questo può avere un impatto devastante sulla vita dei cittadini e causare enormi costi economici per aziende, settori e paesi [2].

Nello scenario delle smart city o smart car la condivisione affidabile delle informazioni può essere utile per includere nuovi partecipanti all'ecosistema e contribuire a migliorare il servizio. Pertanto, l'uso della blockchain può integrare l'IoT con dati affidabili e sicuri.
Inoltre, nell'ambito dell'Industria 4.0 la blockchain integrata ai dispositivi IoT permetterebbe di supportare ogni tassello che compone la Supply Chain, dalla produzione alla logistica [7], in cui i dati verrebbero raccolti nel registro distribuito a prova di manomissione e con la massima trasparenza.

Quelli esposti sono solo alcuni dei molteplici contesti in cui possono essere sfruttati i vantaggi di questa nuova tecnologia per applicazioni IoT.
Il fine è superare le limitazioni che questi dispositivi possiedono per fornire garanzie di sicurezza nella condivisione dei dati.

Capitolo 2

LA BLOCKCHAIN ETHEREUM

Prima di passare all'analisi del lavoro svolto, in questo capitolo verrà illustrato, come vengono generate le transazioni sulla rete Ethereum, quali sono i parametri caratteristici che le compongono e a fronte della fusione e dunque, del cambiamento della logica del consenso,i nuovi paradigmi su cui si basano altre tipologie di transazioni, rispetto a quelle definite legacy.

2.1 Crittografia basata su Curve Ellittiche: ECDSA

La tecnologia blockchain, non solo per quanto riguarda Ethereum, necessita di crittografia alla base per una sua corretta funzionalità, permettendo in modo sicuro e anonimo lo scambio di beni (come ad esempio le criptovalute) tra due utenti.

In generale, la crittografia viene largamente utilizzata per garantire l'autenticità, la non ripudiabilità, la confidenzialità dei dati e la capacità di provare la conoscenza di un segreto senza risalire al segreto stesso (attraverso le digital signature).

Nel contesto in questione, la crittografia basata su curve ellittiche *ECDSA* (Elliptic Curve Digital Signature Algorithm) è, come preannuncia il nome stesso, una crittografia di firma digitale, cui due ingredienti fondamentali identificano una curva ellittica e una funzione hash. Il suo grado di sicurezza è garantito dalla complessità matematica che ha alla base. Si noti che la sicurezza di un algoritmo crittografico non si basa sulla sua complessità matematica bensì nel modo in cui esso stesso viene implementato in uno scenario reale.
Tuttavia, il fatto che la crittografia su curve ellittiche sia legata al problema del logaritmo discreto, che attualmente risulta di complessità esponenziale, conferisce alla stessa una sicurezza intrinseca; probabilmente solo la crittografia post quantum potrà superare tale limite.

L'utilizzo di ECDSA per Ethrerum ha due scopi fondamentali:

- Generazione di una coppia di chiavi pubblica-privata.
- Generazione della digital signature associata ad una transazione.

Nello specifico, il network lavora con la curva ellittica secp256k1, raccomandata dallo Standards for Efficient Cryptography Group [8] e la funzione hash SHA256.

2.1.1 Parametri della curva secp256k1

I parametri del dominio della curva ellittica su Fp associati a una curva di Koblitz secp256k1 sono specificati dalla sestupla T = (p, a, b, G, n, h) dove il campo finito Fp è definito da:

p = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F =

$$2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$$

La curva $E : y^2 = x^3 + ax + b$ su Fp è definita da:
a = 00000000 00000000 00000000 00000000
b = 00000000 00000000 00000000 00000007

Il punto base G in forma compressa è:
G = 02 79BE667E F9DCBBAC 59F2815B 16F81798
e in forma non compressa è:
G = 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8.

Infine, l'ordine n di G e il cofattore sono:
n = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141
h= 01 [8].

2.1.2 Generazione delle chiavi

Il punto generatore G definisce la lunghezza delle chiavi; in particolare il punto G può anche essere rappresentato come segue, dalle sue coordinate x_0 e y_0:

G ($x_0 = 55066263022277343669578718895168534326250603453777594175500187360389116729240$, $y_0 = 32670510020758816978083085130507043184471273380659243275938904335757337482424$)

La coppia di chiavi ECDSA è composta da:

- Chiave privata (numero intero): **privKey**
- Chiave pubblica (punto Elliptic Curve (EC)): **pubKey = privKey * G**

La chiave privata è generata come un numero intero casuale nell'intervallo [0, n-1]. La chiave pubblica pubKey è un punto della curva ellittica, calcolato mediante la moltiplicazione del punto EC: pubKey = privKey * G (la chiave privata, moltiplicata per il punto generatore G). Il punto EC della chiave pubblica x, y può essere rappresentato in forma compressa più un bit (parità).
Per la curva secp256k1, la chiave privata è un intero a 256 bit (32 byte) e la chiave

pubblica compressa è un intero a 257 bit (33 byte).

2.1.3 Generazione e verifica della firma

L'algoritmo di generazione della firma ECDSA di un messaggio m, considerando la chiave privata $privKey$, segue i successivi passaggi [9]:

1. Calcolo del digest del messaggio $h = hash(m)$ e conversione della stringa in un intero e
2. Generazione di un numero casuale k, con $1 \leq k \leq n - 1$
3. Calcolo del punto casuale $kG = x_1, y_1$ e assegnazione di $r = x_1$
4. Calcolo della prova di firma $s = k^{-1}(e + r\mathbf{privKey})$ modulo p, dove k^{-1} è l'Inverso moltiplicativo di k modulo p
5. Generazione della firma (r, s) per il messaggio m.

La verifica della firma accetta come input il messaggio firmato m, la corrispondente firma (r, s) e la chiave pubblica $pubKey$.
La procedura si compone dei seguenti passi:

1. Calcolo del digest del messaggio $h = hash(m)$ e conversione della stringa in un intero e
2. Calcolo dell'inverso moltiplicativo di s modulo p, dove $w = s^{-1}$ modulo p
3. Calcolo $u_1 = ew$ mod p e $u_2 = rw$ mod p
4. Calcolo $X = u_1G + u_2pubKey$
5. Se X = 0 la firma viene rifiutata, altrimenti viene assegnata la coordinata xx_1 di X a v, cioè $v = x_1$
6. Se $v = r$ allora la firma è accettata.

2.2 Ethereum Beacon Chain

Al suo avvio, la rete Ethereum si avvaleva del meccanismo di consenso Proof of Work, consentendo ai nodi della rete Ethereum di concordare sullo stato di tutte le informazioni registrate sulla blockchain Ethereum e impediva alcuni tipi di attacchi. Tuttavia, Ethereum ha disattivato ufficialmente il Proof of Work nel Settembre del 2022 e ha iniziato, invece, a usufruire del Proof of Stake [10].

Nel gergo, questo processo di transizione è stato definito *merge*, in quanto prima che questo accadesse, gli sviluppatori hanno introdotto e testato a lungo il meccanismo (logica) del PoS usufruendo di una rete distaccata dalla mainnet principale, che ha preso il nome di *Beacon Chain*. Quest'ultima era a tutti gli effetti un registro che conduceva e convalidava la rete di staker di Ethereum, prima che questi iniziassero a generare transazioni reali sulla rete.

La disattivazione del Proof of Work e l'attivazione del Proof of Stake su Ethereum ha richiesto di istruire la Beacon Chain ad accettare le transazioni dalla catena Ethereum originale, a raggrupparle in blocchi e organizzarle in una blockchain usando il meccanismo di consenso basato sul Proof of Stake. Allo stesso momento, i client originali di Ethereum hanno disattivato il proprio mining, la propagazione dei blocchi e la logica di consenso, passando tutti questi aspetti alla Beacon Chain. Questo evento è noto come il Merging. Una volta verificatasi "La Fusione", non esistevano più due blockchain; era stata creata un'unica catena Proof of Stake di di Ethereum [11].

Tramite la risorsa, messa a disposizione dalla rete Ethereum, "Ethereum Charts & Statistics" (`https://etherscan.io/charts`) è stato possibile evidenziare le differenze, causate dalla fusione, rispetto all'Ethereum del Proof of Work.
Per far funzionare il Proof of Stake servono 16.384 validatori. Se con il Proof of Work la tempistica dei blocchi è determinata dalla difficoltà di mining, nel Proof of Stake il tempo invece è fisso. Il tempo in Ethereum di Proof of Stake è diviso in slot ed epoche.

Nello specifico, un'epoca è un periodo di tempo che detta quando si verificheranno determinati eventi. Gli esempi includono la velocità con cui vengono distribuiti i premi o quando verrà assegnato un nuovo gruppo di validatori per convalidare le transazioni. Con PoS Ethereum, un'epoca si verifica ogni 32 slot (6,4 minuti). Ogni slot in un'epoca rappresenta un tempo prestabilito, pari a 12 secondi, per un comitato di validatori (gruppi di almeno 128 validatori) per proporre e attestare (votare su) la validità di nuovi blocchi [12].
In base agli studi evidenziati dal lavoro illustrato in [13], il tempo medio per validare un blocco, in data 01 Aprile 2022, è stato identificato pari a 13.2 s, mentre il tempo medio per validare un blocco in data 13 Novembre 2022 risulta essere pari a 12.05s.

In Tabella 2.1 sono evidenziati alcuni valori caratteristici.

Tabella 2.1: Dati Blockchain

Ethereum PoS		
NUMBER OF BLOCKS	AVERAGE BLOCK SIZE	AVERAGE BLOCK TIME
15,968,771	75,008 B	12.05 s

Il numero di blocchi, dopo il merge, è aumentato di circa il 18%, infatti, in data 29 Agosto 2022, il numero di blocchi rilevato era pari a 6,327 al giorno passando, in data 29 Settembre 2022, ad un numero di blocchi pari a 7,154.

Il grafico del conteggio dei blocchi e delle ricompense di Ethereum mostra il numero storico di blocchi prodotti quotidianamente sulla rete Ethereum e la ricompensa totale del blocco (Figura 2.1).
Come si può osservare, vi è un netto incremento del numero di blocchi in corrispondenza di Settembre 2022 (`https://etherscan.io/charts`).

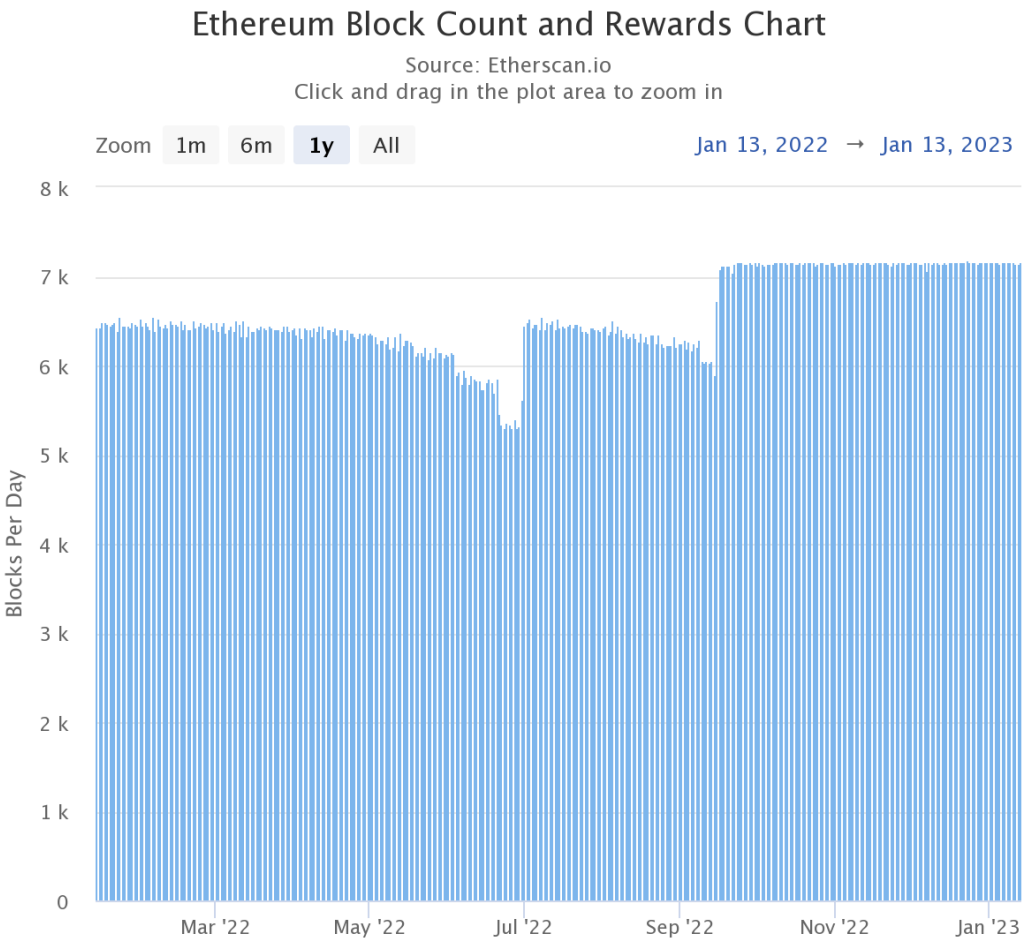


Figura 2.1: Conteggio dei Blocchi e delle ricompense Ethereum

2.3 Transazioni Ethereum

Una transazione Ethereum è definita come un'istruzione crittografica firmata da un account che viene registrata nella blockchain.
Anzitutto, si possono distinguere due tipi di account Ethereum:

- Externally-owned account (EOA): Account controllati dal possessore della chiave privata.
- Contract account: Account controllati dal codice e possono essere attivati solo da un EOA. In altri termini, sono account riferiti agli smart contract.
  Lo smart contract deve essere associato esso stesso ad un account, affinché possa essere raggiunto.

Un account è costituito da una coppia di chiavi crittografiche: pubbliche e private, di cui discusso nei paragrafi precedenti, che permettono di dimostrare che una transazione sia stata effettivamente firmata dal mittente. Ciò impedisce ad attori malevoli di trasmettere transazioni false, in quanto è sempre possibile verificarne il mittente. L'indirizzo pubblico corrispondente all'account è ottenuto prendendo gli ultimi 20 byte dell'hash Keccak-256 della chiave pubblica e aggiungendo 0x all'inizio.

La struttura di una transazione è costituita da una serie di parametri, che vengono settati a seconda del formato della transazione che si vuol implementare[1].

2.3.1 Transazioni EIP-155

Il primo ed unico formato di transazione, di cui in origine Ethereum ha usufruito, prende il nome di *Legacy Transaction*, la cui struttura dati contiene i seguenti campi [14]:

- ***Nonce***: Quantità scalare che rappresenta il numero di transazioni inviate dall'account del mittente.
- ***Gas Price***: Quantità scalare pari al numero di Wei[2] da pagare per unità di gas per tutti i costi di calcolo sostenuti a seguito dell'esecuzione della transazione.
- ***Gas Limit***: Quantità scalare pari alla quantità massima di gas che deve essere utilizzato per l'esecuzione di questa transazione. Questo valore viene definito in anticipo, prima di qualsiasi calcolo, e non può essere aumentato in seguito.
- ***to***: L'indirizzo a 20 byte corrispondente al destinatario della transazione.
- ***Value***: Valore scalare pari al numero di Wei da trasferire all'indirizzo del destinatario.

[1] Nelle sezioni successive verrà approfondita la discussione circa i parametri che compongono la transazione

[2] Il Gwei (gigawei) è pari a 1^{-9} ETH ed è l'unità di misura del carburante GAS mentre il wei, l'unità di misura più piccola dell'ETH pari a 10^{-18} ETH, è il valore indicato nei balance, ossia nei saldi dei conti.

- ***data***: Campo opzionale, identifica un array di byte di dimensioni illimitate che specifica dati arbitrari.
- ***v***: Valore indicato come parametro di recupero della chiave pubblica e viene calcolato dopo aver ricavato r ed s.
- ***r***: Valore corrispondente alla firma della transazione e utilizzato per determinare il mittente della transazione.
- ***s***: Valore corrispondente alla firma della transazione e utilizzato per determinare il mittente della transazione.

I primi cinque campi identificano il payload della transazione, mentre i restanti compongono i parametri di firma.

La struttura presentata segue lo standard EIP-155. EIP è l'acronimo di Ethereum Improvment Proposal e definisce degli standard di miglioramento del protocollo Ethereum, che vengono discussi tra gli sviluppatori. Questo modello si focalizza nel dimensionamento del parametro v, per aumentare la protezione al reply attack.

Ethereum si è evoluto per supportare più tipi di transazioni e per consentire nuove implementazioni con delle migliorie, come lo standard EIP-1559, senza influire sugli altri formati di transazione.

2.3.2 Transazioni EIP-1559

Nell'esecuzione delle transazioni legacy vige un meccanismo ad asta, basato sul *Gas Price* inserito dall'utente. I validatori daranno priorità alle transazioni con il *Gas Price* più alto; questo significa che le transazioni con *Gas Price* basso potrebbero impiegare diverso tempo prima di essere inserite in un blocco. Tuttavia, è noto che tali aste sono altamente inefficienti, in quanto portano a un'estrema volatilità delle commissioni e a frequenti pagamenti eccessivi.

Queste carenze vogliono essere superate con EIP-1559, evitando così che le transazioni, soprattutto quelle con *Gas Price* più basso, restino in pending per molto tempo o non vengano eseguite.
Con la sua introduzione il 5 Agosto 2021 sulla mainnet di Ethereum, la dimensione massima del blocco è stata raddoppiata da circa 15 milioni a T=30 milioni di gas. Tuttavia, la dimensione media del blocco a lungo termine che il sistema si prefigge è la metà di tale limite, vale a dire T/2.

Vengono introdotti alcuni nuovi concetti:

- **Base Fee**: Parametro principale che viene regolato dinamicamente, mediante la seguente formulazione matematica [15]

$$b_{t+1} = b_t \left(1 + d \cdot \frac{G_t - T/2}{T/2}\right). \tag{1}$$

  b_t identifica la tariffa base che viene aggiornata dopo ogni blocco, indicizzata per altezza del blocco $t > 0$.
  d rappresenta il tasso di apprendimento (o parametro di aggiustamento), attualmente impostato a $d = 0.125$, mentre G_t è il gas totale utilizzato dalle transazioni incluse nel blocco $t > 0$ [15].
  Se

  1. $G_t > T/2$: ossia se in un blocco sono incluse più transazioni di quelle previste, la tariffa base aumenta.
  2. $G_t < T/2$: la tariffa base diminuisce.

- **Max Fee per Gas** (f): Importo massimo per unità di gas che l'utente è disposto a pagare affinchè la sua transazione sia inclusa.

- **Max Priority Fee per Gas** (p): Massima mancia (tip) per unità di gas che l'utente è disposto a pagare per incentivare il validatore che include la sua transazione. In tal modo si pone quindi un limite alla ricompensa del validatore.

Dunque, la *Max Fee per Gas* e *Max Priority Fee per Gas* sostituiscono completamente il *Gas Price* presente nel formato EIP-155. Inoltre, affinché una transazione sia inclusa in un blocco è necessario che $f > b_t$, permettendo ai validatori di guadagnare la cosiddetta mancia del validatore, intesa come

$$tip = min = (f - b_t, p). \tag{2}$$

È importante sottolineare che (e in netto contrasto con il mercato delle tasse originali) le commissioni di base non vengono trasferite ai validatori: invece vengono bruciate e rimosse in modo permanente dalla fornitura totale di ETH. La somma della commissione base e della tassa di mancia prese per unità di gas è considerata il prezzo effettivo del gas ed è equivalente al prezzo del gas che un utente avrebbe pagato prima dell'introduzione di EIP-1559 [15].

Oltre ai nuovi concetti descritti, che riguardano il lato economico della transazione, vengono introdotti altri parametri nella struttura della transazione:

- ***Chain ID***: Corrisponde all'identificatore della rete, che viene codificato separatamente anziché incluso nel valore della firma v, come veniva fatto per le transazioni legacy.

- ***Access List***: Specifica un array di indirizzi e chiavi di memorizzazione a cui la transazione intende accedere.
- ***Signature y parity***: Il valore della firma v è ora un semplice bit di parità, che è 0 o 1, a seconda del punto della curva ellittica da utilizzare.

Il formato di transazione legacy è ancora valido per le nuove transazioni sulla rete Ethereum, ma poiché l'EIP-1559 ha cambiamenti significativi nel funzionamento delle tariffe del gas, non è direttamente compatibile con le transazioni legacy. Per mantenere la retrocompatibilità, EIP-1559 descrive un modo per aggiornare le transazioni legacy alle transazioni compatibili con EIP-1559. Lo fa utilizzando il prezzo del gas legacy sia come *Max Priority Fee per Gas* che come *Max Fee per Gas*.

Le transazioni che seguono lo standard EIP-1559, vengono anche definite di *tipo 2*, così da essere immediatamente riconoscibili rispetto a quelle legacy (o di *tipo 0*).

La seguente Tabella 2.2 riassume i parametri che compongono la struttura dati della transazione EIP-1559.

Tabella 2.2: Struttura Dati: Transazione EIP-1559

Payload	ChainID
	Nonce
	Max Priority fee per Gas
	Max Fee Per Gas
	Gas Limit
	to
	Value
	data
	Access List
Signature	Signature y Parity
	Signature r
	Signature s

2.4 Serializzazione dei dati: RECURSIVE-LENGTH PREFIX

Per garantire che le transazioni si propaghino attraverso la rete senza errori vengono utilizzate forme di codifica per ridurre le dimensioni dei payload e per prevenire le più comuni forme di errore (soprattutto quelle causate da connessioni di rete difettose o scadenti).

Come definito dallo *Yellow Paper* [14] di Ethereum, il prefisso di lunghezza ricorsivo (RLP) è un metodo di serializzazione utilizzato per codificare "dati binari arbitrariamente strutturati". RLP offre semplicità definendo solo i tipi di dati byte e array e lasciando ai protocolli di livello superiore la definizione di altri tipi di dati. RLP è deterministico, in quanto prevede un ordinamento esplicito degli elementi all'interno degli array, per garantire che i dati in ingresso risultino codificati sempre nello stesso modo, questo risulta fondamentale per la coerenza degli hash.

Gli elementi validi che l'algoritmo accetta in input sono:

- Stringhe (rappresentazione dell'array di byte della stringa).
- Liste di elementi (elenco di elenchi, elenco di elementi, combinazioni di entrambi).

Il processo di serializzazione segue una serie di casi condizionali, in base alle caratteristiche dell'input fornito all'algoritmo, coerente con i tipi di dati di cui sopra.

Prima di proseguire nella trattazione è opportuno specificare che quando viene codificato ogni carattere di una stringa viene convertito il carattere ASCII nel codice di carattere corrispondente. Esempio: La lettera 'r' diventa il codice carattere con valore 114 in una matrice di byte esadecimali questa singola lettera corrisponderebbe a 0x72 (rappresentazione esadecimale della lettera 'r').

Un input valido soddisferà sempre una di queste regole [16]:

1. Se l'input p è un valore *non-valido* (null , ” , false), allora la codifica RLP è pari a $[0x80]$.
2. Se l'input p è una *Lista vuota* ([]), allora la codifica RLP è pari a $[0xc0]$.
3. Se l'input p è un *singolo byte* in cui $p \in [0x00, 0x7f]$ (decimale $[0 - 127]$), allora la codifica RLP è pari a $[p]$.
4. Se l'input s è una stringa lunga 0-55 byte, la codifica RLP è pari a:
   - len = lunghezza di s
   - Calcolo del primo byte $f = 0x80 + len|$ dove $f \in [0x81, 0xb7]$.

5. Se l'input s è una stringa più lunga di 55 byte, la codifica RLP è pari a:
   - len = lunghezza di s (espressa in esadecimale)
   - Calcolo dei byte richiesti per memorizzare len, b (byte necessari per rappresentare la lunghezza della stringa in esadecimale).
6. Se l'input l è una lista che ha un payload di lunghezza compresa tra 0-55 byte, la codifica RLP è pari a:
   - len = lunghezza di ogni elemento della lista codificato RLP sommata (espressa in esadecimale).
   - Calcolo del primo byte $f = 0xc0 + len$ dove $f \in [0xc1, 0xf7]$.
   - $Concat$ = concatenazione di codifiche RLP di elementi della lista.
7. Se L'input l è una lista che ha un payload di lunghezza superiore a 55 byte, la codifica RLP è pari a:
   - len = lunghezza di ogni elemento della lista codificato RLP sommata (espressa in esadecimale).
   - Calcolo dei byte necessari per memorizzare len, b (bytes richiesti per memorizzare len).
   - Calcolo del primo byte $f = 0xf7 + len$ dove $f \in [0xf8, 0xf7]$.
   - $Concat$ = concatenazione di codifiche RLP di elementi della lista.

Il metodo di serializzazione pone dei limiti, infatti non possono essere codificati:

- Array di byte contenenti 2^{64} o più byte.
- Sequenze i cui elementi serializzati concatenati contengono 2^{64} o più byte.

Queste restrizioni assicurano che il primo byte della codifica di un array di byte sia sempre inferiore a 192, quindi può essere facilmente distinto dalle codifiche di sequenze; mentre per quanto riguarda le liste che il primo byte della codifica non superi 255 (altrimenti non sarebbe un byte).

Capitolo 3

CREAZIONE DI TRANSAZIONI SU DISPOSITIVO ARM CORTEX-M4

Questo capitolo ha lo scopo di riportare le scelte progettuali e i passaggi che hanno permesso di implementare delle funzionalità conformi con Ethereum Beacon Chain. Il flusso dei paragrafi inizierà con la presentazione degli strumenti hardware software dedicati per poi illustrare quale sia stata la metodologia e la procedura che ha consentito l'interazione del dispositivo con la rete di prova ed infine le periferiche configurate per la generazione del codice.

3.1 Materiale e Ambiente di Sviluppo

3.1.1 Hardware

La piattaforma hardware scelta corrisponde alla scheda di sviluppo STM32 Nucleo-144, con microcontrollore STM32439ZI (Figura 3.1).

Il dispositivo STM32F439ZI è basato sulle performance elevate dell' Arm® Cortex®-M4 a 32 bit RISC che opera a una frequenza fino a 180 MHz.
Il core Cortex-M4 è dotato di una Floating-Point Unit (FPU) a precisione singola che supporta tutte le istruzioni e i tipi di dati Arm® a precisione singola. Implementa inoltre una serie completa di istruzioni Digital Signal Processing (DSP) e una Memory Protection Unit(MPU) che migliora la sicurezza delle applicazioni. Il microcontrollore incorpora memorie integrate ad alta velocità (memoria Flash fino a 2 Mbyte, fino a 256 Kbyte di static random access memory(SRAM)), fino a 4 Kbyte di SRAM di backup e un'ampia gamma di I/O e periferiche avanzate collegate a due bus Advanced Peripheral Bus (APB), due bus Advanced High-performance Bus (AHB) e una matrice di bus multi-AHB a 32 bit. Inoltre, offre tre Analog to Digital Converter(ADC) a 12 bit, due Digital to Analog Converter (DAC), un Real-Time Clock(RTC) a basso consumo, dodici timer generici a 16 bit, tra cui due timer Pulse Width Modulation (PWM) per il controllo dei motori, due timer generici a 32 bit, un vero generatore di numeri casuali Random Number Generator (RNG) e una cella di accelerazione crittografica, come l'accelerazione hardware per AES 128, 192, 256, Triple DES, HASH (MD5, SHA-1, SHA-2) e HMAC. Il device supporta fino a 21 interfacce di comunicazione standard e avanzate.

Tutte le caratteristiche del dispositivo in questione possono essere visionate nel datasheet, al seguente link: `https://www.st.com/en/microcontrollers-microprocessors/stm32f439zi.html`.

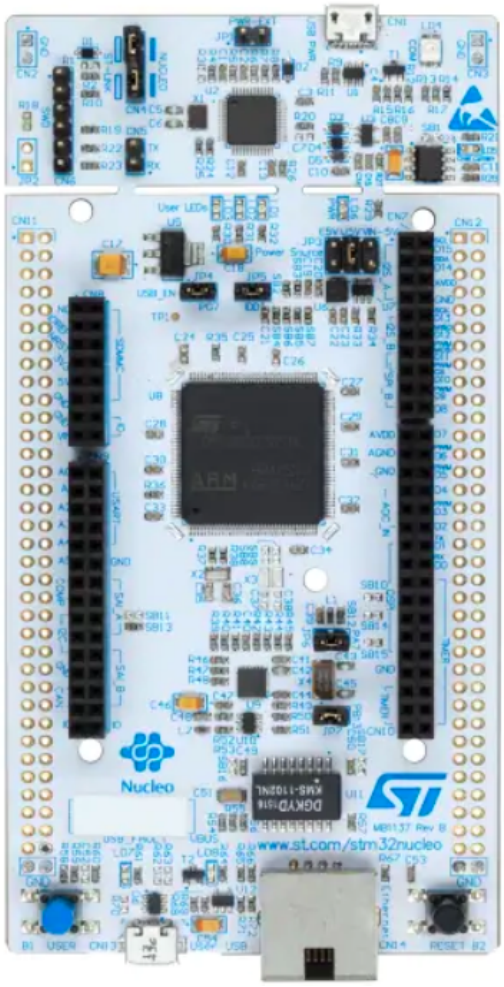


Figura 3.1: Stm32F439ZI Nucleo-144, vista anteriore

3.1.2 Software

Gli strumenti software impiegati nelle diverse fasi di lavoro sono elencati di seguito:

- STM32CubeIDE: piattaforma di sviluppo C/C++ avanzata con funzionalità di configurazione periferica, generazione di codice, compilazione di codice e debug per microcontrollori e microprocessori STM32 [17]. Il tool integra le funzionalità di configurazione e creazione di progetti STM32 da STM32CubeMX, uno strumento grafico che consente una configurazione semplificata di microcontrollori e microprocessori STM32, nonché la generazione del corrispondente codice C di inizializzazione per il core Arm® Cortex®-M, attraverso un processo step-by-step [18].
- Pycharm: ambiente di sviluppo integrato per il linguaggio di programmazione Python, in cui si è andato a sviluppare il codice per interfacciare la scheda con la testnet e inviare correttamente le transazioni.
- Matlab: piattaforma di programmazione e calcolo numerico, utilizzata nella fase di benchmarking, dunque per acquisire e processare i dati provenienti dalla scheda.

3.2 ARM CORTEX-M4:Scelta del Nodo Ethereum

Nei capitoli precedenti si è discusso di Ethereum come rete peer-to-peer di nodi (ovvero device) che eseguono software, che possono verificare i blocchi e i dati delle transazioni.
Per associare un dispositivo ad un nodo di Ethereum deve essere eseguita l'applicazione software, detta client[1].

Esistono diversi tipi di nodi che sfruttano i dati in maniera diversa.
I client possono eseguire tre diversi tipi di nodi:

- Full Node: Memorizza i dati completi della blockchain (sebbene siano periodicamente ridotti così che un full node non memorizzi tutti i dati dal blocco genesi), partecipa alla convalida dei blocchi e effettua la verifica di tutti i blocchi e degli stati. Tutti gli stati possono essere derivati da un nodo completo (sebbene gli stati meno recenti siano ricostruiti a partire dalle richieste effettuate ai nodi di archivio) [19]. Chiaramente, hanno anche il compito di servire la rete e fornire i dati in base alle richieste ricevute.

- Light Node: Scarica le intestazioni dei blocchi; queste contengono solo le informazioni sommarie sui contenuti di ognuno di essi, mentre ogni altra informazione richiesta dal light node viene ricevuta dal full node.
  Può, quindi, verificare solo l'inserimento delle transazioni nei blocchi approvati e consente agli utenti di partecipare alla rete Ethereum senza l'hardware potente o l'elevata larghezza di banda necessari per eseguire i nodi completi. Inoltre, i nodi leggeri non partecipano al consenso (ossia non possono essere miner/validatori), ma possono accedere alla blockchain di Ethereum con la stessa funzionalità di un nodo completo [19].

- Accounts: Sono associati a una coppia di chiavi crittografiche e a un address. Un Account node non partecipa al consenso e non può verificare le transazioni, ma può solo proporre transazioni [13].

Usare un hardware dedicato, come un dispositivo IoT, per eseguire un full node o un light node richede diversi requisiti, di cui il principale è lo spazio di memoria su disco.
Infatti, la sincronizzazione della blockchain di Ethereum è molto intensiva e richiede molto spazio. Sarebbe opportuno avere un Solid-State Disk (SSD) con centinaia di GB di spazio libero da usufruire come riserva anche dopo la sincronizzazione.

Ethereum raccomanda specifiche ben definite come[2]:

[1] Non è più possibile eseguire un client di esecuzione da solo. Dopo il merge, sia il client di esecuzione sia quello di consenso devono essere eseguiti insieme, affinché un utente possa ottenere accesso alla rete di Ethereum.
[2] `https://ethereum.org/en/developers/docs/nodes-and-clients/run-a-node/`

- CPU veloce con più di 4 core.
- Oltre 16 GB di RAM.
- SSD veloce con più di 1 TB.
- Oltre 25 Mb/s di larghezza di banda.

Dunque, sia un full node che un light node (seppur richiede una minor occupazione di memoria) non possono essere le scelte adeguate, a causa delle capacità limitate del dispositivo.

Quindi, in base alle analisi svolte, l'unico modo per coinvolgere un ARM CORTEX-M4 nel mondo Blockchain è quello di far funzionare il dispositivo come account.
In questo modo, deve solo generare, firmare e formattare le transazioni e non partecipa attivamente al meccanismo di consenso.
Le transazioni costruite saranno quindi inviate a dispositivi più potenti che eseguono un nodo (full o light), che le invierà alla rete includendole nei blocchi e prendendo in carico la costruzione e la proposta del blocco. In tal caso un account può memorizzare anche solo 40 B di dati, rispetto a 124 B, che corrispondono a 32 B di chiave segreta e 8 B di nonce. Dalla chiave segreta poi è possibile generare on-board la chiave pubblica (64 B) e il corrispondente address (20 B).

3.3 Interazione del dispositivo con la rete

In virtù dei limiti nelle risorse, di cui discusso precedentemente, Ethereum offre la possibilità di poter usufruire di nodi come servizio.

Ci sono una serie di servizi che eseguono infrastrutture di nodi ottimizzate, che permettono di concentrarsi sullo sviluppo dell'applicazione, piuttosto che sulla manutenzione dell'infrastruttura stessa.
Nello specifico, si può utilizzare un provider Application Programming Interface (API) di terze parti; tra i più rilevanti possono esserne citati alcuni come Alchemy, Infura, QuickNode, Moralis[3].
Questi eseguono una varietà di client e tipi di nodi, permettendo, ad esempio, di accedere ad un full node, oltre a metodi specifici del client in un'unica API.

Questi servizi in genere forniscono una chiave API che può essere usata per scrivere e leggere dalla blockchain. Spesso includono l'accesso alle testnet Ethereum oltre alla Mainnet.

[3] https://ethereum.org/en/developers/docs/nodes-and-clients/nodes-as-a-service/

È importante notare che i servizi che fungono da nodo non memorizzano e non devono memorizzare le chiavi o le informazioni private, offrendo così un'enorme vantaggio in termini di sicurezza.

Come provider si è optato per *Alchemy*[4], una fra le piattaforme principali per sviluppatori blockchain, che mediante accesso all'endpoint di un'API sulla rete di prova Görli (o Goerli), ha permesso di inviare transazioni e interrogare la rete stessa.

Görli è una fra le principali testnet di Ethereum che l'11 Agosto 2022, ha ufficializzato il suo passaggio al PoS. Infatti, è stata selezionata proprio per verificare come abbia influito il cambiamento del meccanismo di consenso. In quanto rete di prova, le transazioni usufruiscono di Ether "finti", ottenibili gratuitamente mediante *faucet*, ovvero semplici siti web che permettono di regalare frazioni di criptovaluta.

3.4 Generazione Address

Mediante script python si è andato a generare un account conforme ad Ethereum, seguendo questi semplici passaggi,:

1. Generazione della chiave privata, mediante ECDSA.
2. Calcolo della chiave pubblica associata.
3. Address ottenuto dagli ultimi 20 byte dell'hash della chiave pubblica, mediante Keccak256.

Dunque, attraverso questa logica viene ottenuta la chiave privata che viene memorizzata sul dispositivo e il corrispondente address, funzionale alla generazione delle transazioni.
La chiave privata non deve essere assolutamente divulgata o smarrita, altrimenti non sarebbe possibile in alcun modo risalire ad essa e la sua perdita avrebbe come conseguenza il non poter più accedere al saldo associato all'indirizzo corrispondente, anche se per il lavoro in questione non si tratta di criptovalute reali. Come analizzato nelle sezioni precedenti, la chiave pubblica non deve essere memorizzata, ma viene calcolata on-board dalla chiave privata.

Esempio di address generato:
0x30E1A3C3ccE3585C0988B85904C2ed1FDEcAd83a.

[4] `https://www.alchemy.com`

3.5 Settaggio Periferiche

Per gli obiettivi del lavoro in questione le periferiche attivate, mediante lo strumento grafico STM32CubeMX, sono di seguito elencate.

- **RNG (Random Number Generator) o True RNG**: l'RNG integrato nei prodotti STM32 fornisce numeri casuali che vengono utilizzati quando si desidera ottenere un risultato imprevedibile. Infatti, quelli utilizzati per le applicazioni crittografiche producono tipicamente sequenze di bit 0 e 1 casuali. L'attivazione della periferica ha permesso di valutare le performance del dispositivo in relazione all'algoritmo ECDSA.

  Il diagramma a blocchi semplificato in Figura 3.2 mostra i moduli della periferica funzionali e di controllo. Il generatore di numeri casuali è basato su un circuito analogico composto da diversi oscillatori ad anello le cui uscite vengono campionate e poi XORate per generare i *seeds* che alimentano un blocco di Digital post-processing in grado di produrre quattro numeri casuali a 32 bit per ogni round di calcolo. Il campionamento dei seeds analogici è regolato da un segnale di clock RNG dedicato, in modo che la qualità del numero casuale sia indipendente dalla frequenza HCLK[5]. Il contenuto del blocco di post-processing viene trasferito nel registro dati attraverso una First In First Out (FIFO) a quattro parole.
  Il Data Ready flag (DRDY) viene attivato non appena il FIFO è pieno e viene automaticamente resettato quando non è più possibile leggere altri dati dall'RNG. Parallelamente, un blocco di gestione degli errori verifica il corretto comportamento del seed e la frequenza del clock di origine dell'RNG. I bit di stato vengono impostati e viene attivato un interrupt se viene rilevata una sequenza anomala nel seed o se l'RNG frequenza è troppo bassa. Il controllo degli errori della frequenza RNG deve essere disabilitato se il clock RNG è fissato al di sotto di $AHB\_CLK/32$ (ad esempio, per motivi di qualità)[20].

[5]Clock principale della central processing unit (CPU)

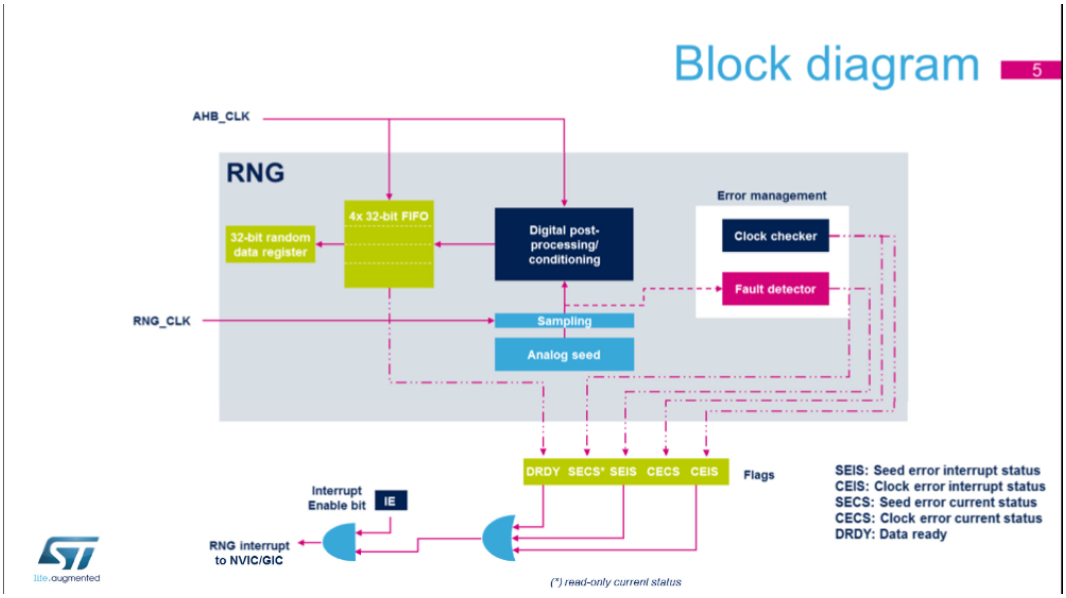


Figura 3.2: Diagramma a blocchi del RNG [20].

- **USART3 (Universal Synchronous-Asynchronous Receiver/Transmitter)**: L'interfaccia è disponibile su PD8 e PD9 dell'STM32 può essere collegata sia all'ST-LINK che al connettore morpho ST. Per impostazione predefinita, la comunicazione USART3 tra l'STM32 di destinazione e ST-LINK è abilitata, per supportare la porta COM virtuale.
  Nello specifico, la periferica è stata responsabile della comunicazione seriale tra device e PC, impostata in modalità asincrona, sia nella fase di testing tramite Matlab, sia nella fase di trasmissione delle transazioni attraverso python.

- **DWT (Data Watchpoint and Trace)**: È un'unità di debug opzionale che fornisce punti di controllo, tracciamento dei dati e profilazione del sistema per il processore.
  L'unità contiene quattro contatori, tra cui è stato abilitato quello per il clock cycle ($CYCCNT$),mediante istruzioni fornite da riga di codice.
  La sua funzione è quella di un contatore che viene incrementato ad ogni ciclo di clock del processore. In caso di overflow, CYCCNT viene azzerato [21].
  In questo modo sono state ottenute le misure del numero dei cicli di clock per valutare le performance del dispositivo in relazione ad ECDSA e alla generazione delle transazioni, di cui si discuterà nel capito successivo.

Capitolo 4

BENCHMARKING

Introdotti gli strumenti utili alla realizzazione del progetto, si passa all'analisi delle performance del dispositivo, considerando il numero di cicli di clock misurati in relazione al numero di esecuzioni delle funzioni valutate.
La prima sezione si concentra sull'implementazione e sulla valutazione delle operazioni congrue all'algoritmo ECDSA; mentre nelle ultime due sezioni, con il medesimo iter, ma considerando alcuni passaggi algoritmici differenti, l'analisi volge alla generazione delle transazioni valutando, prima, quelle Legacy per proseguire con quelle definite dallo standard EIP-1559.

4.1 Benchmarking ECDSA: uECC

Nel paragrafo 2.1 è stato descritto ECDSA, schema crittografico che è alla base delle transazioni Ethereum. Dunque, si è valutata la risposta del dispositivo relativamente alle operazioni che l'algoritmo include.

Nel lavoro presente in [22], sono state studiate le performance del dispositivo in relazione alle librerie *X-Cube-Crypto* [23] e *micro-ecc* [24], confrontando i risultati ottenuti.

La libreria *XCube-Cripto*, messa a disposizione dalla STMicroelectronics, include tutti i principali algoritmi di sicurezza per la crittografia, l'hashing, l'autenticazione dei messaggi e la firma digitale. Al contrario, *micro-ecc* è una libreria open-source, creata per dispostivi embedded, definita come una veloce implementazione Ellipticcurve Diffie–Hellman (ECDH) ed ECDSA per processori a 8 bit, 32 bit e 64 bit. Inoltre, impiega una protezione di base contro la maggior parte degli attacchi sidechannel (come il timing o power analysis), adattandosi bene all'approccio di cercare di implementare una soluzione utilizzabile, leggera e per lo più non intrusiva.
Dal confronto tra le due è emerso che la libreria micro-ecc fornisce risultati migliori. Per tale ragione, al fine di non replicare test ridondanti, l'analisi dell'attuale lavoro si focalizza sull'uso di tale libreria.

Il file di progetto impiegato in questa prima fase, è "uECC-Benchmark", che include la cartella corrispondente alla libreria micro-ecc, in cui sono contenuti il file header "uECC.h" e il corrispondente source file "uECC.c".

Il testing è stato eseguito considerando 1000 iterazioni in cui si sono state valutate le funzioni:

- **SHA-256**: Generazione del digest del messaggio da firmare.
  La funzione SHA-256 viene implementata mediante una funzione inclusa nalla libreria HAL [25] del firmware, cioè la *HAL_HASHEx_SHA256_Start*, previa

abilitazione tramite il tool STM32CubeMX della periferica *HASH* impostata in SHA256 mode.

- **KeyGen**: Generazione della coppia di chiavi privata-pubblica.
  Viene implementata mediante la funzione *uECC_make_key*, che viene chiamata dopo la funzione *uECC_set_rng*, utilizzata per generare byte casuali. Entrambe si trovano all'interno della libreria micro-ecc.

- **Sign**: Generazione della firma ECDSA del digest, utilizzando la chiave privata, mediante la funzione *uECC_sign*.

- **Verify**: Verifica della firma ECDSA, utilizzando il digest e la chiave pubblica, tramite la funzione *uECC_verify*.

Per ognuna delle esecuzioni viene generato un messaggio random da firmare e delle chiavi private random, a partire dei valori casuali generati dall'RNG di bordo.

4.1.1 Codice e Modalità di Esecuzione

Le parti di codice funzionali nella raccolta delle misurazioni appartengono allo script main.c del file di progetto in questione. Per semplicità sono state riportate solo le parti salienti del codice, ai fini della spiegazione dei risultati ottenuti, omettendo le parti relative alla configurazione delle periferiche, inizializzazione delle variabili e tutte le funzioni, corrispondenti ai moduli attivati mediante tool STM32CubeMX, che di default sono presenti nel codice.

Avviata la modalità debbugging, nell'istante in cui l'esecuzione del programma giunge in corrispondenza del test loop, viene avviato lo script Matlab che permette di raccogliere le misurazioni, grazie alla lettura tramite porta seriale e contemporaneamente viene ripresa l'esecuzione del programma principale.

Come si può osservare dalle righe di codice sottostanti,grazie all'attivazione del registro DWT del device e del relativo contatore CYCCNT, come già descritto precedentemente, per ognuna delle funzioni valutate, il numero dei cicli eseguiti durante un'operazione viene ricavato attraverso la differenza tra il numero di cicli compiuti prima e dopo la suddetta operazione (deltaC = c1 - c0), infine il risultato viene trasmesso tramite UART e processato in Matlab.

```

int main(void)
....
....

while (1)
...
```

```
  for (int i=0; i<1000; i++) {
      testDone = uECC_test3(uECC_secp256k1(), &myRNG_func); /** Test 3
     - random message, random privK **/
    }
...

int uECC_test3(uECC_Curve curve, uECC_RNG_Function rng_function) {
  uint16_t deltaC_TX[2]; //HAL_UART_Transmit needs u8 or u16
  unsigned message_size = 32;
  uint8_t message[message_size];
  int hash_size = 32;
  uint8_t message_hash[hash_size];
  int pubK_size = uECC_curve_public_key_size(curve); //64 bytes for
    secp256k1
  int privK_size = uECC_curve_private_key_size(curve); //32 bytes for
    secp256k1
  uint8_t pubK[pubK_size];
  uint8_t privK[privK_size];
  int messageGen = 0;
  int keygen = 0;
  int status = 0;
  uint8_t signature[64]; //64 bytes for secp256k1
  int sign = 0;
  int verify = 0;

  /** Test 3 - random message, random privK **/
  messageGen = rng_function(message, message_size);

  /* Generate key pair */
  uECC_set_rng(rng_function);
  c0 = DWT->CYCCNT;
  keygen = uECC_make_key(pubK, privK, curve);
  c1 = DWT->CYCCNT;
  deltaC = c1 - c0;
  deltaC_TX[0] = deltaC >> 16;
  deltaC_TX[1] = deltaC;
  HAL_UART_Transmit(&huart3, deltaC_TX, sizeof(deltaC_TX), 10);


  /* Generate SHA-256 digest */
  c0 = DWT->CYCCNT;
  status = HAL_HASHEx_SHA256_Start(&hhash, &message, sizeof(message),
    &message_hash, 10000); //Timeout 10s
  c1 = DWT->CYCCNT;
  deltaC = c1 - c0;
  deltaC_TX[0] = deltaC >> 16;
  deltaC_TX[1] = deltaC;
  HAL_UART_Transmit(&huart3, deltaC_TX, sizeof(deltaC_TX), 10);

  /* ECDSA signing */
  c0 = DWT->CYCCNT;
```

```
55  sign = uECC_sign(&privK, &message_hash, hash_size, &signature, curve
      );
56  c1 = DWT->CYCCNT;
57  deltaC = c1 - c0;
58  deltaC_TX[0] = deltaC >> 16;
59  deltaC_TX[1] = deltaC;
60  HAL_UART_Transmit(&huart3, deltaC_TX, sizeof(deltaC_TX), 10);
61
62  /* Verify ECDSA signature */
63  /* (Recomputation of SHA-256 digest is for benchmarking purposes)
     */
65  status = HAL_HASHEx_SHA256_Start(&hhash, &message, sizeof(message),
      &message_hash, 10000); //Timeout 10s
66  c0 = DWT->CYCCNT;
67  verify = uECC_verify(&pubK, &message_hash, hash_size, &signature,
      curve);
68  c1 = DWT->CYCCNT;
69  deltaC = c1 - c0;
70  deltaC_TX[0] = deltaC >> 16;
71  deltaC_TX[1] = deltaC;
72  HAL_UART_Transmit(&huart3, deltaC_TX, sizeof(deltaC_TX), 10);
73
74
75  if ((messageGen && keygen && !status && sign && verify) == 1) //
      successful sign&verify
76
77    return 0;
78  }
79  else
80  {
81    return 2;
82  }
83 ...
```

Listato 4.1: Codice Semplificato del Test: Messaggio e Chiave privata Random

4.1.2 Risultati Testing: Messaggio e Chiave Privata Random

Dalle 1000 esecuzioni per ogni funzione, l'STM32 genera 4000 valori uint32 di misurazioni. A partire da questi valori, Matlab andrà a leggere 8000 valori uint16 dalla porta seriale.
Ogni misurazione identifica un numero di cicli di clock.

Per compiere le giuste valutazioni in merito alla performance del device, vengono generati i plot, mediante script Matlab, inerenti al numero di cicli di clock, in relazione al numero di esecuzioni delle operazioni d'interesse.
Inoltre, ai fini di un'analisi completa sono stati riportati i parametri statistici calcolati, quali media, valore massimo, valore minimo e deviazione standard nella Tabella 4.1.

Il plot in Figura 4.1 permette di confrontare le misure dei cicli di clock di tutte le funzioni, questo consente di mettere in evidenza gli ordini di grandezza in gioco. Infatti, si può osservare che le operazioni valutate impiegano un numero di cicli proporzionali a 10^7, fatta eccezione della funzione *SHA256* che impiega un numero di cicli di clock circa quattro volte inferiore, permanendo nell'ordine di 10^3 cicli di clock, dunque risulta l'operazione più rapida tra tutte.

Continuando ad esaminare la funzione *SHA256* in Figura 4.2, i messaggi in input sono di volta in volta differenti, grazie alla randomicità conferita dal TRNG di bordo. Questo si potrebbe tradurre in un andamento variabile del numero di cicli di clock, al contrario vi è, invece, un andamento pressoché costante, confermato anche dalla deviazione standard contenuta e pari a 4.7298, ad eccezione di sporadici spike dovuti a processi interni al device.

Per quanto riguarda i plot corrispondenti alla Figura 4.3 e Figura 4.4 si può dedurre di come, in entrambi i casi, l'ordine di grandezza nel numero di cicli di clock arrivi a 10^7 e la deviazione standard in entrambi i casi si dimostra limitata (Figura 4.1). Questo permette di affermare che, indipendentemente dall'input, che potrebbe essere un messaggio o una chiave fissi o generati casualmente, se l'andamento del numero di cicli di clock risulta costante (confermato dai parametri statistici), le funzioni di *KeyGen* e di *Sign* presentano ottimi requisiti di tempo costante.
Questo requisito è fondamentale per ottenere un sistema solido contro gli attacchi basati sul tempo di esecuzione, ovvero il *Timing Attack*.

L'ultima operazione valutata, ovvero la verifica della firma, *verify*, presenta invece una deviazione standard meno contenuta rispetto alle altre funzioni pari a 397021.1. Tuttavia, questo algoritmo non usufruisce di parametri segreti, come ad esempio la chiave privata, dunque, non necessita di rispettare la caratteristica di tempo costante. Inoltre, bisogna specificare che è l'unica funzione che non viene implementata da un account, sebbene sia comunque stata analizzata per fornire uno scenario esauriente

dell'implementazione ECDSA.

Nel complesso, il testing ha permesso di confermare l'utilizzo della libreria *micro-ecc* e dunque della fattibilità dell'esecuzione dell'algoritmo ECDSA sul dispositivo in questione.

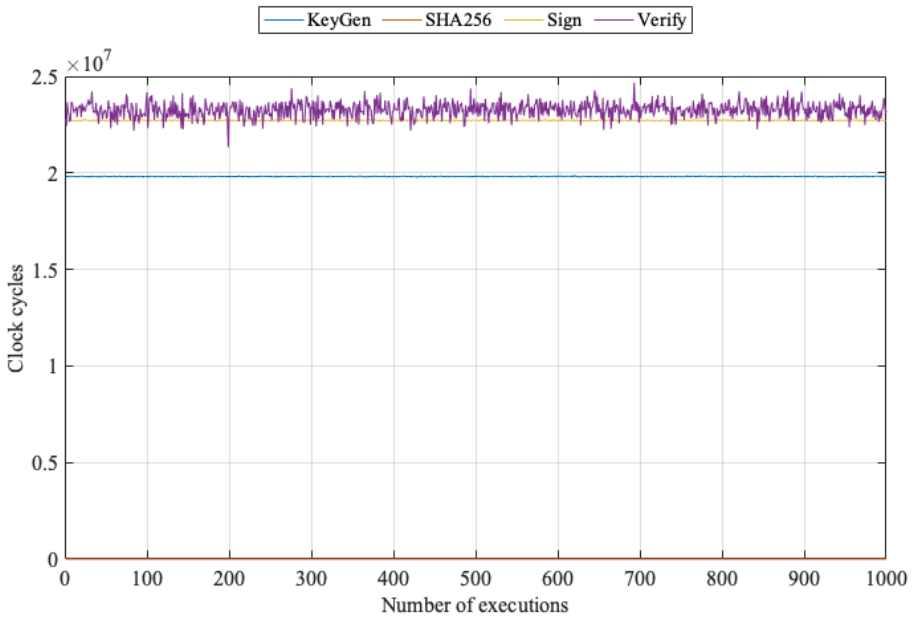


Figura 4.1: Rappresentazione dei cicli di clock per 1000 esecuzioni delle funzioni (Sha3, keygen, Sign, Verify). Per ogni esecuzione vi sono chiave e messaggi random.

Tabella 4.1: Media, Valore Massimo, Valore Minimo, e Deviazione Standard per ogni funzione, ricavati da 1000 esecuzioni.

Funzione	Media	Valore Massimo	Valore Minimo	Deviazione Standard
SHA256	1035.264	1125	1035	4.7298
KeyGen	19814677.216	19856509	19770903	12213.547
Sign	22717078.682	22778171	22663848	15586.485
Verify	23259510.440	24674984	21323746	397021.1

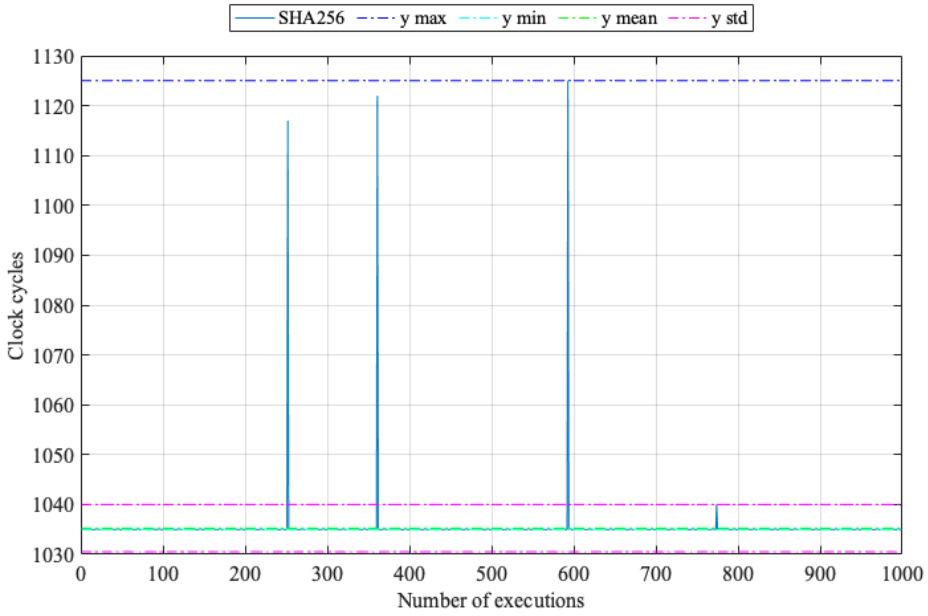


Figura 4.2: Rappresentazione dei cicli di clock per 1000 esecuzioni della funzione Sha256. In ogni esecuzione viene generato un digest di un messaggio random.

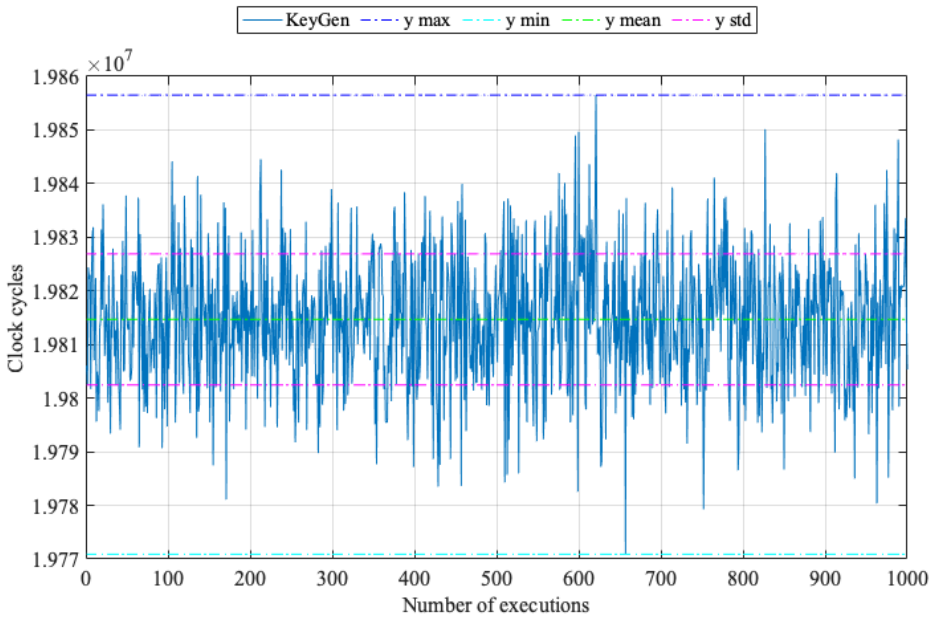


Figura 4.3: Rappresentazione dei cicli di clock per 1000 esecuzioni della funzione KeyGen. In ogni esecuzione viene generato una coppia di chiavi pubblica-privata differente.

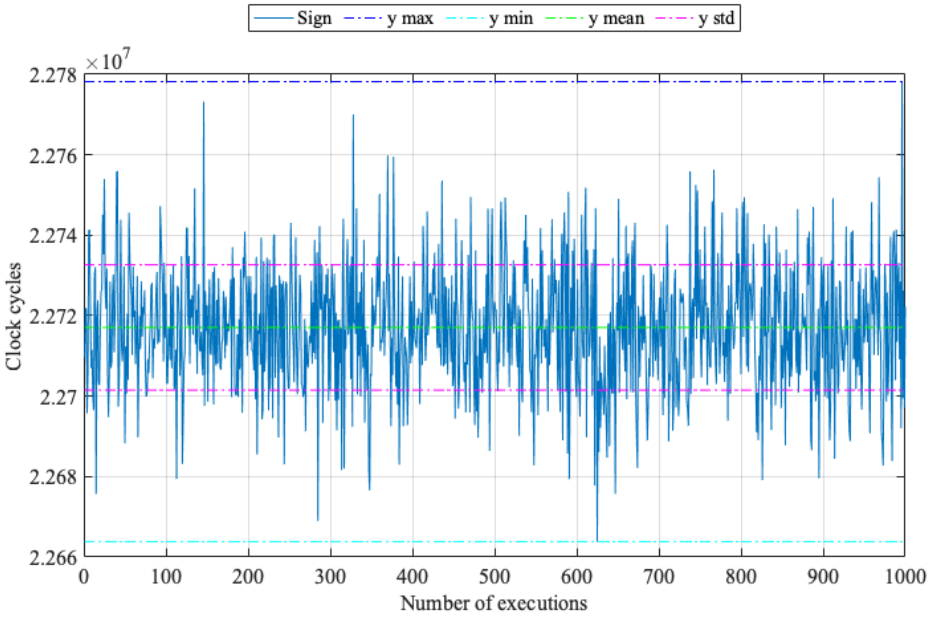


Figura 4.4: Rappresentazione dei cicli di clock per 1000 esecuzioni della funzione Sign. In ogni esecuzione messaggi random vengono firmati mediante chiavi random.

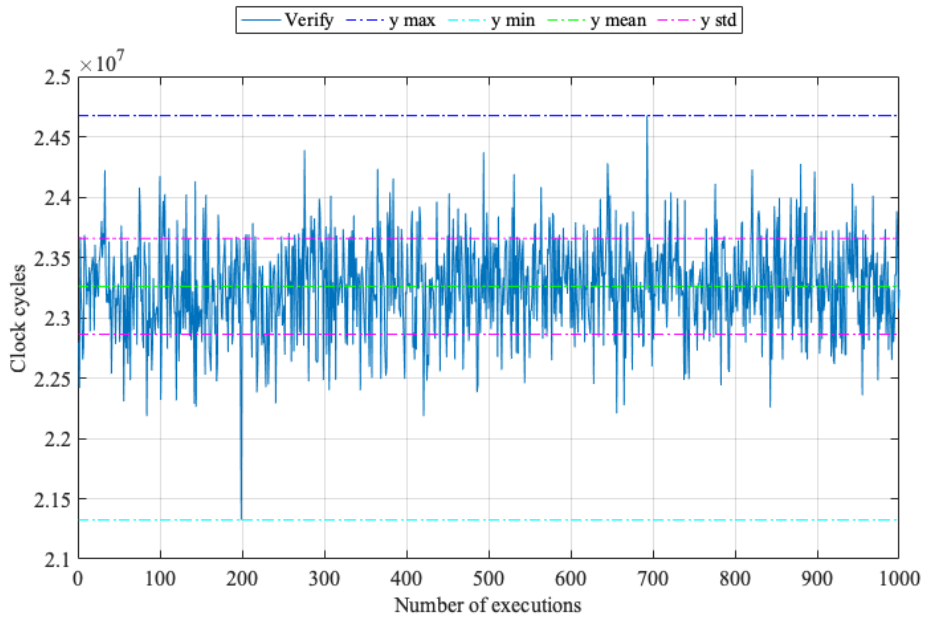


Figura 4.5: Rappresentazione dei cicli di clock per 1000 esecuzioni della funzione Verify. In ogni esecuzione viene verificata la firma usando la corrispondente chiave pubblica.

4.2 Benchmarking Legacy Transactions

Dopo aver confermato i risultati relativi all'utilizzo della libreria micro-ecc, sono state valutate le performance del device in relazione al numero dei cicli di clock e ai tempi necessari per la generazione di 1000 raw transactions generate dalla scheda stessa.

In questa fase si è usufruito degli script del progetto inerente a "GoerliTx_Legacy". In cui attraverso la funzione "createTx_benchmark" vengono esaminati i passaggi algoritmici:

1. ***Init.***: Inizializzazione delle variabili presenti nella funzione
2. ***uTxRLP***: Serializzazione dei parametri utili per generare la transazione non firmata attraverso la codifica RLP.
   In tal caso sono stati utilizzati i file.c e i file.h della libreria *Ethereum* RLP [26], il cui scopo è di serializzare oggetti in byte grezzi.
3. ***Keccak256***: Calcolo dell'hash dei dati necessari a produrre la transazione (non firmata) tramite Keccak256.
   L'implementazione di questa funzione deriva dalla libreria *ethers* [27].
4. ***rawTxRLP***: Serializzazione tramite una seconda codifica RLP dei parametri utili alla produzione della transazione completa di firma. In tal caso, è stata adoperata la stessa libreria citata nel punto 2.

Nel paragrafo 2.3.1 è stata presentata la struttura dati di una transazione che segue lo standard EIP-155 (o legacy), pertanto, rispettando le specifiche, nella Tabella 4.2 vengono presentati i parametri di cui si è usufruito per generare e successivamente inviare le transazioni verso la rete.

Tabella 4.2: Parametri settati per la generazione delle transazioni nella fase di benchmarking

Dati Tx	Valore in Esadecimale
Chain ID	"5"
Gas Price	"15540BE400" (Wei)
Gas Limit	"29040"
Destination Address	"30E1A3C3ccE3585C0988B85904C2ed1FDEcAd83a"
Value	"2386F26FC10" (Wei)
Data	"44617469205478"

Tutti i valori che si che si osservano in Tabella 4.2, che vengono definiti nello script del file di progetto in questione, sono in rappresentazione esadecimale.
Altro parametro che di diritto compone la transazione è il *Nonce*, che però non è

presente tra i dati mostrati. Quest'ultimo, che tiene conto della storia delle transazioni precedentemente inviate da quell'address mittente, viene direttamente inviato al device, mediante porta seriale, dalla rete di prova Goerli, grazie all'API provider dedicato definito nel codice Python[1].
Ogni rete Ethereum presenta un proprio identificatore univoco, che può essere ricercato nella cosiddetta *Chainlist* [28]; in queste circostanze l'ID pari a 5 è riferito alla rete di prova Goerli.

Per quanto riguarda il *Gas Price* è stato impostato ad un valore tale per cui si potesse avere la certezza che la transazione fosse inclusa nel blocco in tempi rapidi, nonostante questa sia una fase di analisi sulla generazione delle transazioni, che non prevede la trasmissione delle stesse.
Dunque, il valore in questione risulta essere pari a 9990164678180 Wei (ottenuto previa conversione in decimale), che corrispondono a 0.00000999016467818 ETH per unità di gas.

Il *Gas Limit* minimo necessario per l'invio di una transazione da un EOA ad è 21000, tuttavia dipende anche dai dati della transazione, ad esempio per ogni byte di dati pari a zero vi è un costo aggiuntivo di 4 gas e per ogni byte non nullo di 68 gas. Quindi, si è preferito impostarlo pari a 168000, considerando la conversione in decimale.

Il *Destination Address*, corrisponde all'account creato opportunamente per *Goerli*, come spiegato nel paragrafo 3.4, a cui viene rimosso lo "0x" iniziale.

Value, quantità che si vuole inviare al *Destination Address*, è stato impostato pari a 0.00000244140625 ETH e potrebbe risultare relativamente elevato, ma come chiarito poche righe fa in questa fase le transazioni vengono unicamente prodotte per il benchmark.
Successivamente, questo parametro verrà posto pari ad una quantità nulla.

Infine *Data*, quantità opzionale, contiene il byte array "0x44617469205478", scelto arbitrariamente, che corrisponde alla stringa codifica ASCII (American Standard Code for Information Interchange) dei caratteri "Dati Tx".

[1] Il listato relativo al codice in questione viene presentato nel quinto capitolo, nella sezione 5.1

4.2.1 Parametri di firma: r, s, v

Seguendo i passaggi algoritmi illustrati nella sezione precedente, per l'esecuzione del punto 2., in conformità con lo standard EIP-155, i parametri di firma vengono impostati considerando:

v = *ChainID*, r = 0, s =0

Il gruppo di byte che compone la transazione {r, s, v} viene aggiornato quando viene calcolata la firma ECDSA dell'hah digest della raw transaction non firmata.

In particolare, il parametro v, come definisce lo standard EIP-155, viene esplicitato dalla seguente formulazione:

$$v = b + ChainID * 2 + 35 \tag{3}$$

dove b può assumere solo due valori : {0,1}.

Per determinare univocamente v, è necessario ribadire, con ulteriori dettagli, con quale metodologia viene generata la firma della raw transaction.

Ottenuto, tramite *Keccak256* il digest della transazione non firmata, il passo successivo consta di generare il parametro k tramite concatenazione della chiave privata e dell'hash digest, ovvero:

k = (hash(unsignedTx)||privKey).

Questo procedimento ha lo scopo di evitare che, a partire da un k generato casualmente, si possa estrarre una chiave privata ECDSA da due messaggi differenti, firmati con lo stesso valore k [29].

Appurato ciò, bisogna considerare che sulla curva ellittica ogni coordinata x possiede due punti opposti y e -y. Per tale ragione, quando al parametro r viene assegnata la coordinata x (si veda il punto 2. del paragrafo 2.1.3) a quest'ultima sarà associata una coordinata y. Dalla valutazione della parità o disparità della coordinata y del punto, ne consegue l'assunzione del valore di b:

- Coordinata y pari : $b = 0$
- Coordinata y dispari : $b = 1$

Nel calcolo della quantità s (si veda il punto 4. del paragrafo 2.1.3), invece, dobbiamo considerare che se s risulta minore rispetto alla metà dell'ordine n della curva ellittica, allora b resta il medesimo, rispetto alle valutazioni della coordinata y. Se dal calcolo emerge che s è maggiore della metà dell'ordine n della curva, allora s diventa pari al suo opposto *(-s)* e b varia rispetto al valore assunto precedentemente.

Seguendo questo iter si ottiene il risultato definitivo del parametro v.

4.2.2 Codice e modalità di esecuzione

L'analisi delle misurazioni ottenute dalla generazioni di 1000 raw transactions segue le stesse modalità operative descritte nella sezione 4.1.1, ma considerando che, in tal caso, le operazioni valutate sono differenti.

Le parti di codice relative a queste fase, appartengono agli script "main.c" e "transaction.c".

Come si evince dal flusso di esecuzione del programma principale (Listato 4.2), dopo aver ricevuto il *nonce*, tramite UART, grazie all'interazione con la rete mediante opportune istruzioni fornite da script Python, si passa all'esecuzione del test loop e alla trasmissione dei valori, sempre attraverso porta seriale, verso Matlab.
Il *Nonce* viene incrementato di 1 in ogni iterazione.

La funzione che si occupa della generazione delle raw transaction è *createTx_benchmark*, in cui come si osserva nel Listato 4.3 è stato suddiviso in cinque intervalli di misura adiacenti.

```
int main(void)
{
...
  while (1)
  {

...
      uint32_t nonceCount = 0;
      ReceiveNonce(&nonceCount);
      for (iterationCount=0; iterationCount<1000; iterationCount++) {
        UpdateNonce(nonceCount, nonce);
        BenchmarkTx(nonce,rawTx, &rawTx_charLength);
        nonceCount++;
      }
....

    }
...
  }
```

Listato 4.2: Codice semplificato del benchmarking: main.c

4.3

```
..
void BenchmarkTx(char *nonce, char *rawTx, uint16_t *rawTx_charLength)
    {
  createTx_benchmark(rawTx, rawTx_charLength, Go_chainID, Go_privK,
    Go_to, Go_value, Go_gas_limit, Go_gas_price, nonce, Go_data);
}
...
```

```
6
7  int createTx_benchmark(char *rawTx,
8        uint16_t *rawTx_charLength,
9        uint32_t chainID,
10       char *privK_char,
11       char *to,
12       char *value,
13       char *gas_limit,
14       char *gas_price,
15       char *nonce_char,
16       char *data) {
17    /*----- Benchmark Initialization BEGIN -----*/
18    c0 = DWT->CYCCNT;
19    uECC_Curve curve = uECC_secp256k1();
20    uECC_RNG_Function rng_function = &RNG_func;
21    uint8_t hash_size = 32;
22    uint8_t payload_size = 128;
23    uint8_t data_bytesLength = strlen(data) + 1;
24    uint16_t Tx_size = 256 + data_bytesLength;
25    uint16_t uTx_charLength = 0;
26      uint8_t pubK_size = uECC_curve_public_key_size(curve); //64 bytes
      for secp256k1
27      uint8_t privK_size = uECC_curve_private_key_size(curve); //32
      bytes for secp256k1
28 //   uint8_t pubK[pubK_size];
29    uint8_t privK[privK_size];
30    uint8_t k_seed[Tx_size + privK_size];
31    uint8_t k_uint8[hash_size];
32    uint32_t k[hash_size / 4];
33    uint8_t payload_hash[hash_size];
34    uint8_t rawSignature[pubK_size]; //64 bytes for secp256k1
35    uint8_t uTx[Tx_size];
36    char uTx_char[Tx_size];
37      char *r = 0;
38      char r_char[65];
39      char *s = 0;
40      char s_char[65];
41      uint32_t v = 0;
42      uint32_t recID = 0;
43    uint8_t sign = 0;
44    c1 = DWT->CYCCNT;
45    deltaC = c1 - c0;
46    deltaC_TX[0] = deltaC >> 16;
47    deltaC_TX[1] = deltaC;
48    HAL_UART_Transmit(&huart3, deltaC_TX, sizeof(deltaC_TX), 10);
49    /*----- Benchmark Initialization END -----*/
50
51
52    /*----- Benchmark uTxRLP BEGIN -----*/
53    c0 = DWT->CYCCNT;
54    /** Encode unsigned transaction */
```

```
55  /* This is the payload to hash. Signing this hash will give the r,s,
      v
56   * parameters needed for encoding the final signed transaction.
57   */
58  uTx_charLength = gen_transaction(uTx_char, Tx_size, nonce_char,
      gas_price, gas_limit, to, value, data, r, s, chainID);
59  ///////Fix bug in RLP encoding
60  if(uTx_char[2]==48){
61    if(uTx_char[3]==48){
62      uTx_char[2]=56;
63    }
64  }
65  c1 = DWT->CYCCNT;
66  deltaC = c1 - c0;
67  deltaC_TX[0] = deltaC >> 16;
68  deltaC_TX[1] = deltaC;
69  HAL_UART_Transmit(&huart3, deltaC_TX, sizeof(deltaC_TX), 10);
70  /*----- Benchmark uTxRLP END -----*/
71
72  /*----- Benchmark Keccak256 BEGIN -----*/
73  c0 = DWT->CYCCNT;
74  /** Hash & Sign */
75  payload_size = (uTx_charLength - 1) / 2;
76  hex2byte_arr(uTx_char, (payload_size + 1)*2, uTx, payload_size); //
      Convert payload from char to uint8
77  hex2byte_arr(privK_char, (privK_size + 1)*2, privK, privK_size); //
      Convert privK from char to uint8
78  /*** To avoid ECDSA nonce reuse exploit - BEGIN
79   * "#Singing a second message using the same K value [...] is what
      opens ECDSA to attack"
80   * (from https://github.com/Marsh61/ECDSA-Nonce-Reuse-Exploit-
      Example/blob/master/Attack-Main.py , line 70)
81   * To avoid this exploit, k is not randomly generated. It is
      obtained instead as
82   *     k = keccak256(uTx || privK)
83   * reducing the chanches of using the same k when signing different
      messages.
84   * */
85  for (int i=0; i<payload_size; i++) {
86    k_seed[i] = uTx[i];
87  }
88  for (int i=0; i<privK_size; i++) {
89    k_seed[payload_size + i] = privK[i];
90  }
91  keccak256(k_seed, payload_size + privK_size, k_uint8); //Compute k
      from seed
92  for (int i=0; i<8; i++) {
93    k[i] = k_uint8[i*4 + 3] + (k_uint8[i*4 + 2] << 8) + (k_uint8[i*4 +
       1] << 16) + (k_uint8[i*4] << 24);
94  }
95  /*** Exploit avoidance - END */
```

```
96  keccak256(uTx, payload_size, payload_hash); //Compute digest from
      payload
97  c1 = DWT->CYCCNT;
98  deltaC = c1 - c0;
99  deltaC_TX[0] = deltaC >> 16;
100 deltaC_TX[1] = deltaC;
101 HAL_UART_Transmit(&huart3, deltaC_TX, sizeof(deltaC_TX), 10);
102 /*----- Benchmark Keccak256 END -----*/
103
104 /*----- Benchmark uECCsign BEGIN -----*/
105 c0 = DWT->CYCCNT;
106 uECC_set_rng(rng_function); //DON'T REMOVE! Needed for uECC_sign
107 uint32_t k_final[hash_size / 4];
108 sign = uECC_sign_deterministic_custom(&privK, &payload_hash,
      hash_size, k, &rawSignature, &v, k_final, curve); //Sign digest
109
110 /** Compute recovery ID */
111 recID = v + (chainID << 1) + 35; /* From https://ethereum.
      stackexchange.com/a/62769 */
112 c1 = DWT->CYCCNT;
113 deltaC = c1 - c0;
114 deltaC_TX[0] = deltaC >> 16;
115 deltaC_TX[1] = deltaC;
116 HAL_UART_Transmit(&huart3, deltaC_TX, sizeof(deltaC_TX), 10);
117 /*----- Benchmark uECCsign END -----*/
118
119 /*----- Benchmark finalRLP BEGIN -----*/
120 c0 = DWT->CYCCNT;
121 /** Generate final signed transaction */
122 int8_to_char(&rawSignature, privK_size, r_char);
123 int8_to_char(&rawSignature[privK_size], privK_size, s_char);
124
125 *rawTx_charLength = gen_transaction(rawTx, Tx_size, nonce_char,
      gas_price, gas_limit, to, value, data, r_char, s_char, recID);
126 ///////Fix bug in RLP encoding
127 if(rawTx[4]==48){
128   if(rawTx[5]==48){
129     rawTx[4]=56;
130   }
131 }
132 c1 = DWT->CYCCNT;
133 deltaC = c1 - c0;
134 deltaC_TX[0] = deltaC >> 16;
135 deltaC_TX[1] = deltaC;
136 HAL_UART_Transmit(&huart3, deltaC_TX, sizeof(deltaC_TX), 10);
137 /*----- Benchmark finalRLP END -----*/
138
139
140 if (sign == 1){
141
142   return 0;
```

```
143   }
144   else {
145
146     return 2;
147   }
148 }
```

Listato 4.3: Codice semplificato del benchmarking: transaction.c

4.2.3 Risultati ottenuti dalla generazione di transazioni legacy

Dalle misure ottenute del numero di cicli di clock in relazione al numero di esecuzioni, pari a 1000, viene valutato il tempo impiegato dal device a compire le operazioni d'interesse t_e, rapportando il numero di cicli (n_{ck}) alla frequenza massima di lavoro del dispositivo (f_d), che da specifiche risulta essere pari a 180 MHz, ovvero:

$$t_e = \frac{n_{ck}}{f_d} \tag{4}$$

I dati vengono processati in Matlab e conseguentemente generati i plot e i corrispondenti parametri statistici per ognuno dei cinque slot di misura, riportati nella Tabella 4.3.

Per quanto riguarda l'operazione di inizializzazione, *Init.* in Figura 4.6 si può affermare che nel complesso vi è un numero costante del numero di cicli di clock misurati, in quanto le operazioni sono identiche in ogni iterazione. Gli spike sporadici, come anticipato precedentemente nella sezione 4.1.2, vengono attribuiti a funzionalità interne eseguite dal device, che non influiscono con i parametri calcolati. I valori minimi e massimi presentano una differenza irrisoria e la deviazione standard è contenuta.

I testing coinvolgono un'address mittente cui valore del *Nonce* risulta essere pari a 865[2].
In virtù di questo, considerando la prima serializzazione dei dati, mediante codifica RLP, *uTxRLP*, in Figura 4.7, si osserva un andamento del numero di cicli che ricade in un range di variabilità definita. Se all'address mittente fosse stato attribuito un *Nonce* pari a 0, considerando che le raw transactions generate sono in un numero pari a 1000, nel grafico si osserverebbe un andamento crescente. La ragione di ciò è attribuita al modo in cui l'algoritmo effettuata la codifica, in quanto per input di byte crescenti (maggiori di 127) vengono aggiunti prefissi all'output, come esposto nella sezione 2.4.

[2] L'address mittente parte da un Nonce non nullo, in quanto prima del corrispondente testing sono state eseguite alcune prove per sperimentare un primo approccio con l'ambiente Blockchain. Quanto scritto è vero anche per la generazione delle transazioni del nuovo formato, di cui si tratterà in eseguito.

L'operazione di generazione dell'hash digest *Keccak256*, in Figura 4.8, è influenzata dalla codifica del passo precedente e per tale ragione vale il medesimo ragionamento di poc'anzi, arrivando ad un numero di cicli di clock nell'ordine di 10^5.

I digest, indipendentemente dalla lunghezza dell'input, hanno tutti lunghezza fissa e pari a 32 byte e dunque l'operazione successiva di firma non viene influenzata dalla lunghezza in byte della serializzazione. Come si evince dalla Figura 4.9, l'operazione di generazione della firma richiede un numero di cicli di clock più elevato, rispetto alle altre operazioni, nell'ordine di 10^7.

Infine, la procedura termina con un'ultima codifica che comprende i valori di firma aggiornati. Questi ultimi differiscono per ogni transazione, infatti in Figura 4.10 si riscontra un andamento simile al plot della prima codifica (Figura 4.7), ma con una variabilità al clock cycle maggiore, considerando i parametri aggiornati.

In base ai valori medi relativi alle tempistiche valutate per ogni operazione, escludendo la sezione dedicata all'inizializzazione, in Tabella 4.3, è possibile affermare che la generazione di una legacy raw transaction richiede in media circa 129.65 ms.

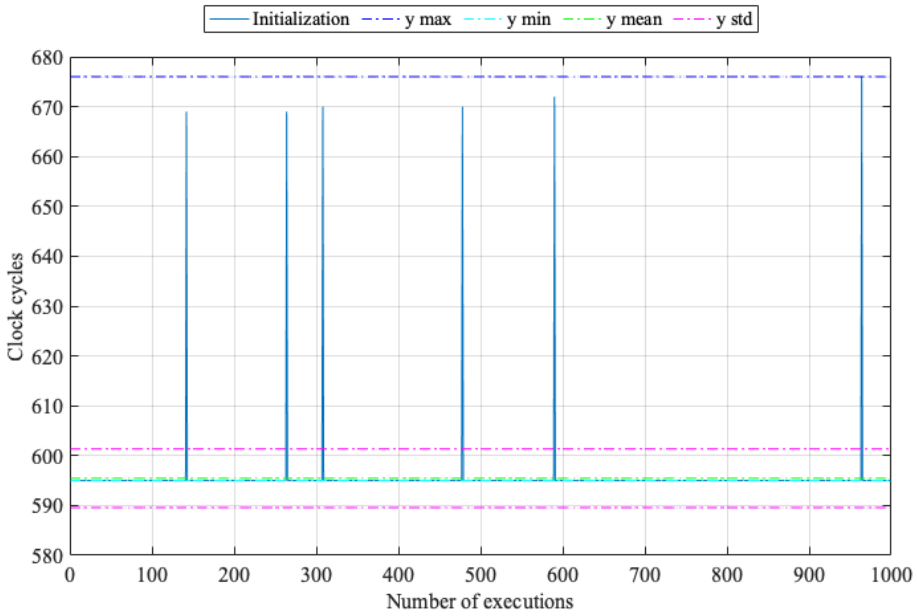


Figura 4.6: Cicli di clock impiegati per le operazioni di inizializzazione, misurati durante la generazione di 1000 transazioni verso Goerli.

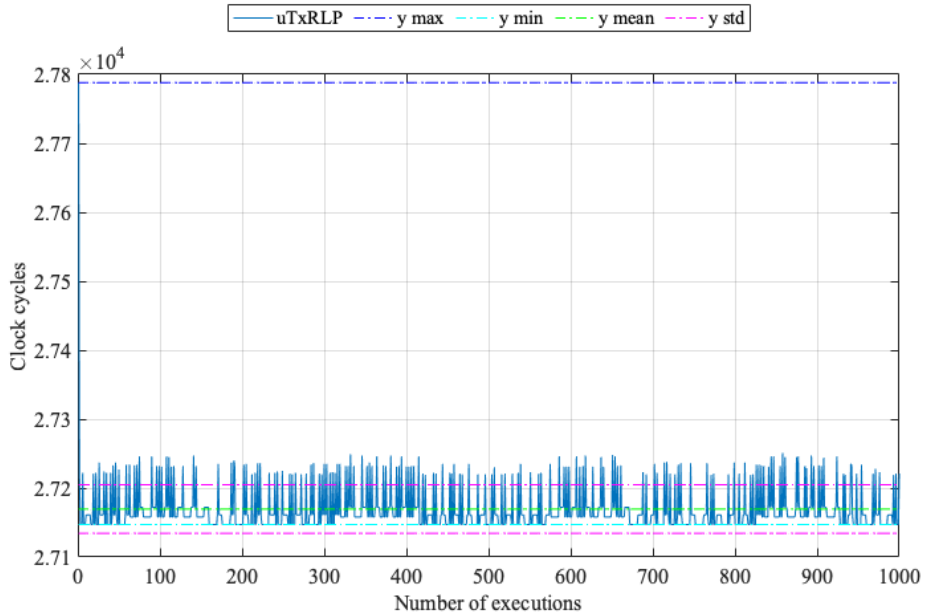


Figura 4.7: Cicli di clock impiegati per le operazioni di serializzazione della transazione (non firmata)), misurati durante la generazione di 1000 transazioni verso Goerli.

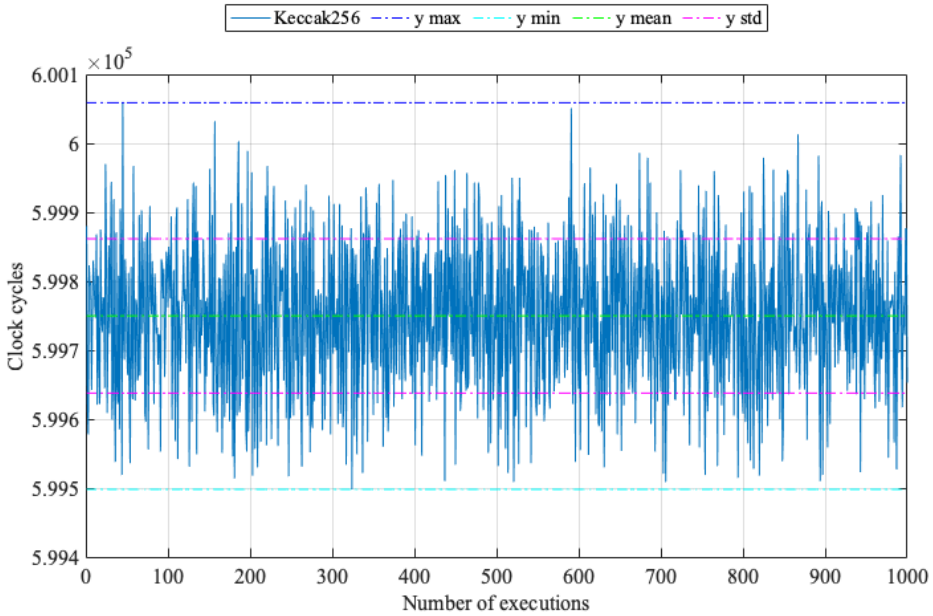


Figura 4.8: Cicli di clock impiegati per il calcolo dell'hash digest da firmare, misurati durante la generazione di 1000 transazioni verso Goerli.

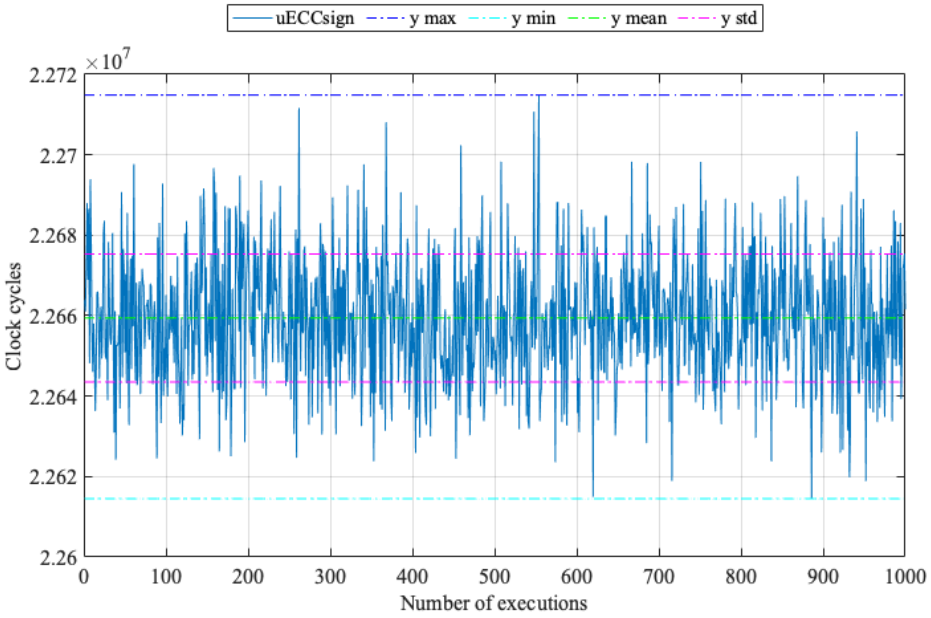


Figura 4.9: Cicli di clock impiegati per firmare una transazione, misurati durante la generazione di 1000 transazioni per Goerli.

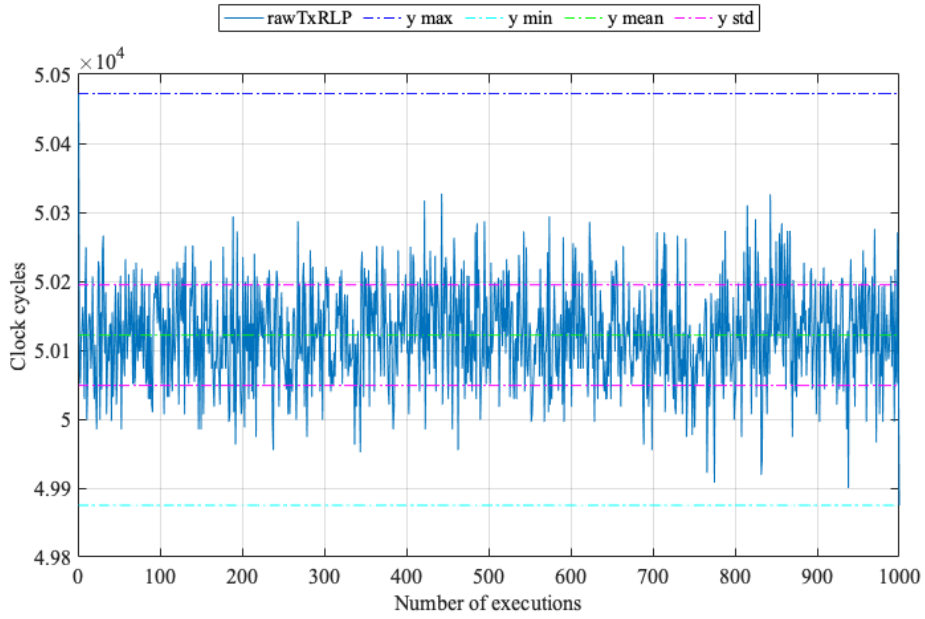


Figura 4.10: Cicli di clock impiegati per codificare una transazione firmata, misurati durante la generazione di 1000 transazioni per Goerli.

Tabella 4.3: Parametri ricavati dalle misure dei cicli di clock trascorsi, per ciascuna operazione, durante la generazione di 1000 transazioni Legacy per Goerli.

Operazione		Media	Valore Massimo	Valore Minimo	Deviazione Standard
Init.	clock cycles	595.4560	694	595	5.8753
	Time[ms]	3.31×10^{-3}	3.85×10^{-3}	3.30×10^{-3}	3.26×10^{-5}
uTxRLP	clock cycles	27169.531	27788	27147	35.1949
	Time[ms]	0.15	0.15	0.14	1.95×10^{-4}
Keccak256	clock cycles	599749.7190	600059	599498	111.9077
	Time[ms]	3.33	3.33	3.32	6.22×10^{-4}
uECCSign	clock cycles	22659419.879	22714836	22614496	15915.110
	Time[ms]	125.89	126.19	125.64	0.09
rawTxRLP	clock cycles	50121.759	50472	49875	72.854
	Time[ms]	0.28	0.28	0.27	1.58×10^{-5}

4.3 Benchmarking New Transactions

Per ricavare il numero di cicli di clock e i tempi legati alle operazioni per il nuovo formato di transazione, che segue lo standard EIP-1559, si è usufruito del file di progetto "GoerliTx_New", in cui sono presenti alcune differenze rispetto al file per la generazione delle transazioni Legacy.

Nonostante la successione dei passaggi algoritmici analizzati sia la medesima, come indicato nella sezione 4.2, vengono, in prima istanza, aggiunti i nuovi parametri che compongono la struttura dati della transazione. Infatti, in Tabella 4.4 , si evince la mancanza del *Gas Price*, sostituito dal *Max Fee per Gas* e *Max Priority Fee per Gas*, che corrispondono rispettivamente a 20 gwei (0.00000002 ETH) e 5 gwei (0.000000005 ETH).
Il *ChainID* in questo caso deve essere interpretato come una costante di tipo char che è parte integrante della transazione; mentre per il formato legacy il *ChainID* viene inizializzato come un intero, in quanto viene incluso nel valore della firma v.
Altra novità è rappresentata dall'*Access List*, impostato come lista vuota ([]).

Le restanti quantità presenti sono equivalenti a quelle indicate nella Tabella 4.2.

Tabella 4.4: Parametri settati per la generazione delle transazioni EIP-1559 nella fase di benchmarking

Dati Tx	Valore in Esadecimale
Chain ID	"5"
Max Fee per Gas	"4A817C800" (Wei)
Max Priority Fee per Gas	"12A05F200" (Wei)
Gas Limit	"29040"
Dastination Address	"92Bdd4b0eC6f1b9A13f6959EC9859dDdcc2DE39B"
Value	"2386F26FC10" (Wei)
Data	"44617469205478"
Access List	""

Prima di proseguire nello sviluppo della discussione circa l'impostazione dei parametri di firma, è opportuno spiegare in che modo è possibile distinguere i due formati di transazione.

Con l'introduzione del nuovo standard è nata la necessità di assicurare che fosse retrocompatibile con il formato standard e che non potesse essere differenziato solo in base al payload codificato.
Per tale ragione è stato introdotto il *TransactionType*[3] che rappresenta univocamente, attraverso un valore numerico, il formato specifico. Nel caso delle transazioni EIP-1559 questo valore è pari a 2.
Quindi, una transazione che aderisce a questo standard viene registrata sulla rete nella seguente forma:

$$TransactionType||TransactionPayload \tag{5}$$

Dove TransactionPayload rappresenta il contenuto vero e proprio della transazione, che attraverso un'operazione di concatenazione byte/byte array viene associato al TransactionType.
Riportando in termini espliciti l'espressione 5:

- 0x02 || rlp([ChainID, Nonce, Max Priority Fee per Gas, Max Fee per Gas, Gas Limit, destination address, Value, Data, Access List, v, r, s])

[3] https://eips.ethereum.org/EIPS/eip-2718

4.3.1 Parametri di firma: r, s, v

Secondo lo standard che definisce questo nuovo formato r, s, v non compongono la raw transaction non firmata, come invece descritto per il formato legacy e come già anticipato, in queste circostanze, il *ChainID* ha solo il ruolo di identificare la blockchain di destinazione, dunque non è associato al parametro v, anche noto come *recovery ID*.
Infatti l'hash del payload della transazione non firmata è determinato nel seguente modo:

- Keccak256(0x02 || rlp([ChainID, Nonce, Max Priority Fee per Gas, Max Fee per Gas, Gas Limit, destination address, Value, Data, Access List])

Il passaggio successivo, dunque, è il calcolo della firma; questo implica che i componenti della firma ECDSA siano estratti e determinati in conformità del nuovo standard.
I parametri r e s vengono generati seguendo la stessa procedura descritta per le transazioni legacy, mentre per quanto concerne il *recovery ID*, in tal caso, è rappresentato solo da un bit di parità che può essere 0 o 1, equivalente al parametro b presente in (3) nella sezione 4.2.1:

$v = \{0,1\}$

La parità è determinata dal valore y del punto dei curva, per cui r è il valore x associato nel processo di firma secp256k1, la cui procedura viene descritta in esposta nelle sezioni 2.1.3 e 4.2.1.

4.3.2 Risultati ottenuti dalla generazione di transazioni EIP-1559

Anche per la valutazione delle misurazioni relative alla generazione del nuovo formato di transazione, il codice e le modalità operative sono affini a quelle descritte nel paragrafo 4.2.2.
Le uniche differenze sono legate all'introduzione di alcune righe di codice associate all'aggiunta del prefisso "02", che specifica il tipo della transazione e della funzione che realizza la prima serializzazione mediante codifica RLP, che non include l'inizializzazione dei parametri di firma r, s, v.

I grafici in Figura 4.11, Figura 4.12, Figura 4.13, Figura 4.15 derivano dal processing dei dati, trasmessi mediante porta seriale dal device verso Matlab. Come nelle sezioni precedenti, in Tabella 4.5 sono riportati i parametri statistici relativi ad ognuno degli slot di misura.

Si consideri che il *Nonce* relativo all'address mittente è pari a 136, dunque per le regole imposte dalla codifica RLP, essendo un valore maggiore di 127, vale quanto

esplicitato in 4.2.3 per entrambi i passaggi legati alla serializzazione dei parametri della transazione.

Nel complesso è possibile affermare che generare raw transactions conformi allo standard EIP-1559 richiede un tempo medio pari a 129.66 ms.

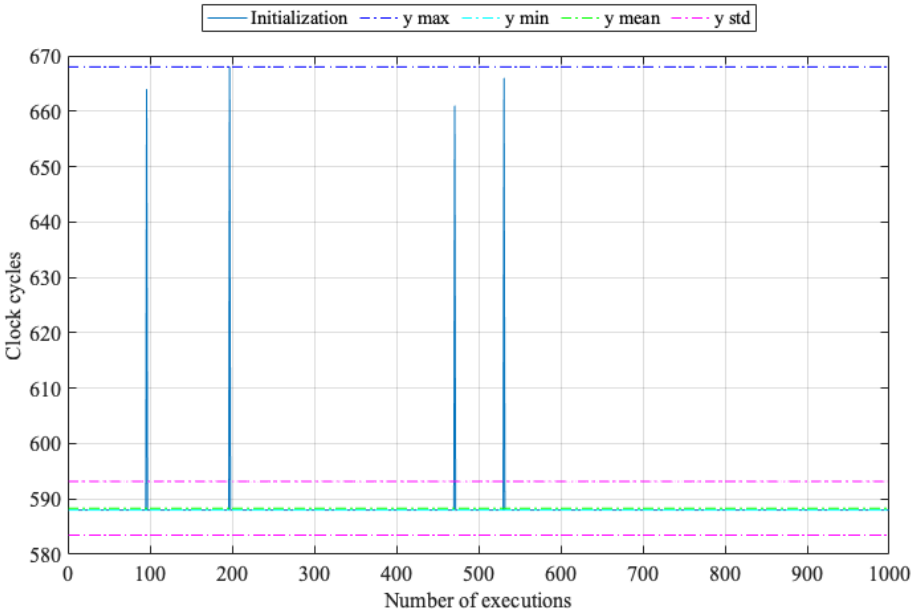


Figura 4.11: Cicli di clock impiegati per le operazioni di inizializzazione, misurati durante la generazione di 1000 transazioni verso Goerli.

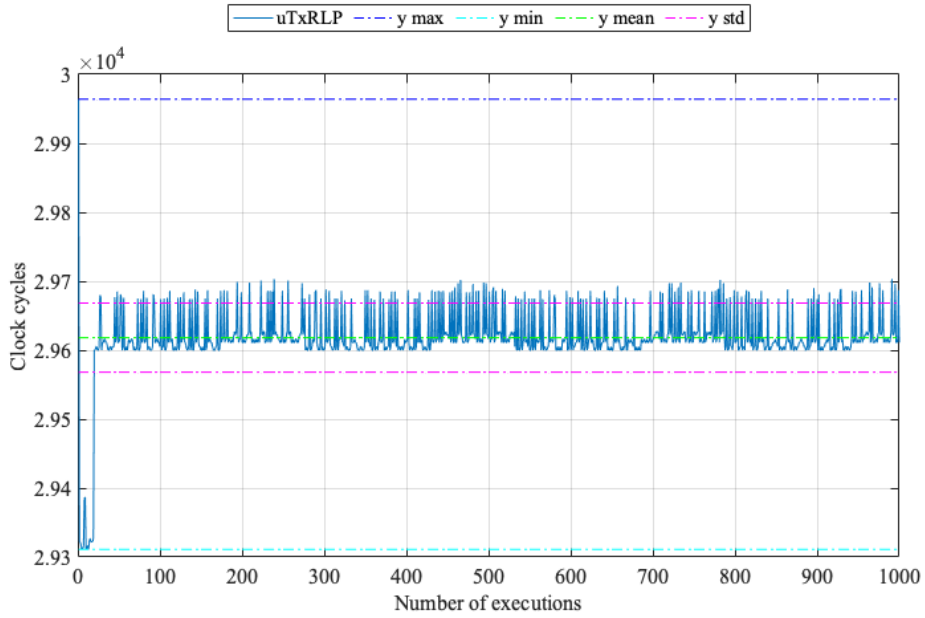


Figura 4.12: Cicli di clock impiegati per le operazioni di serializzazione della transazione (non firmata)), misurati durante la generazione di 1000 transazioni verso Goerli.

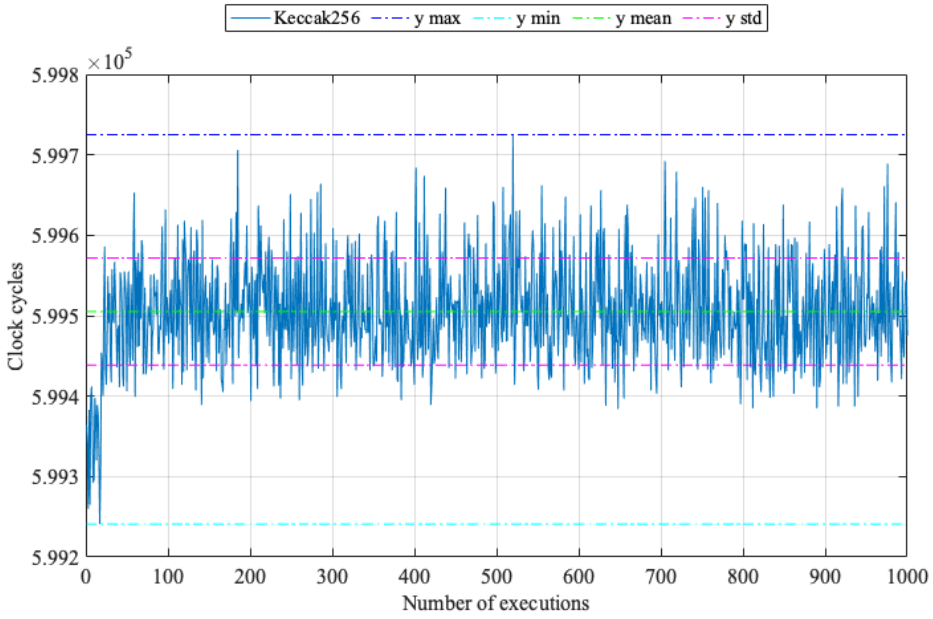


Figura 4.13: Cicli di clock impiegati per il calcolo dell'hash digest da firmare, misurati durante la generazione di 1000 transazioni verso Goerli.

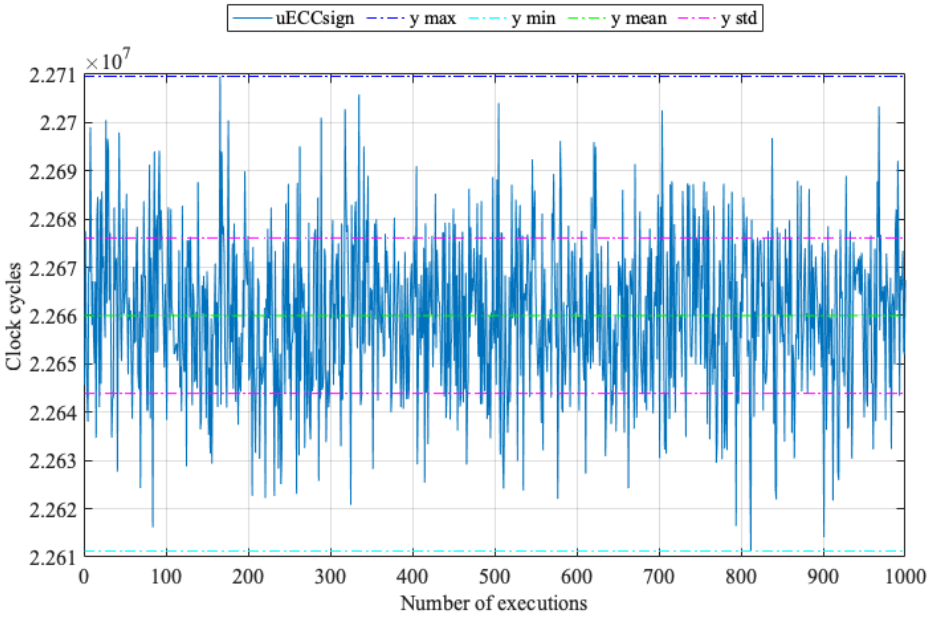


Figura 4.14: Cicli di clock impiegati per firmare una transazione, misurati durante la generazione di 1000 transazioni per Goerli.

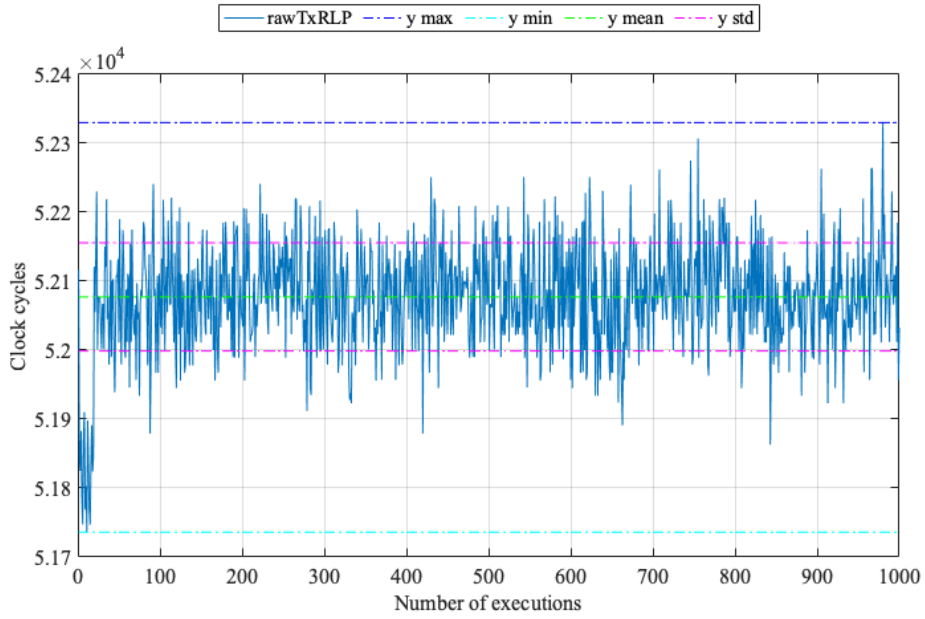


Figura 4.15: Cicli di clock impiegati per codificare una transazione firmata, misurati durante la generazione di 1000 transazioni per Goerli.

Tabella 4.5: Parametri ricavati dalle misure dei cicli di clock trascorsi, per ciascuna operazione, durante la generazione di 1000 transazioni EIP-1559 per Goerli.

Operazione		Media	Valore Massimo	Valore Minimo	Deviazione Standard
Init.	clock cycles	588.3070	668	588	4.8496
	Timems]	3.27×10^{-3}	3.71×10^{-3}	3.27×10^{-3}	2.69×10^{-5}
uTxRLP	clock cycles	29618.064	29964	29311	50.1368
	Time[ms]	0.16	0.17	0.16	2.79×10^{-4}
Keccak256	clock cycles	599505.109	599725	599241	66.4876
	Time[ms]	3.32	3.33	3.32	3.69×10^{-4}
uECCSign	clock cycles	22659986.893	22709603	22611234	16106.856
	Time[ms]	125.89	126.16	125.62	0.09
rawTxRLP	clock cycles	52077.236	52330	51736	78.502
	Time[ms]	0.29	0.29	0.28	4.36×10^{-4}

4.4 Confronto delle misure ottenute dalla generazione delle transazioni: Legacy vs EIP-1559

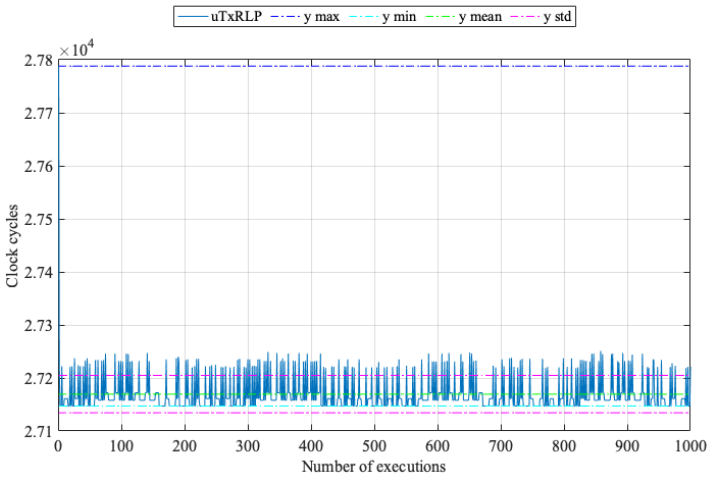
Questa sezione ha lo scopo di comparare i risultati ottenuti nella fase di benchmarking per la generazione dei due formati di transazione.

I risultati dei paragrafi precedenti vengono sintetizzati nella Tabella 4.6, in cui vengono riportati il valor medio e la deviazione standard in termini di cicli di clock e il tempo medio richiesto per ogni operazione valutata.
Per completezza vengono riportati anche i plot delle due sperimentazioni a confronto.

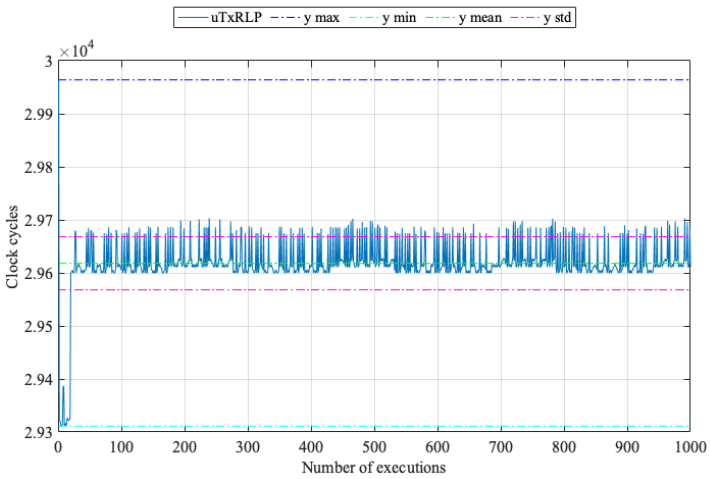
Il codice che genera le transazioni di entrambi i tipi, come già illustrato, si può considerare a grandi linee lo stesso, in quanto solo alcune nuove istruzioni ne fanno la differenza, ragion per cui è lecito attendersi performance analoghe.
Inoltre in ognuna delle 1000 iterazioni i parametri che compongono le transazioni

sono sempre gli stessi, fatta eccezione per il *Nonce* che viene incrementato in ogni esecuzione e considerando address mittenti eterogenei tra le due prove.

Osservando i grafici in Figura 4.16a e Figura 4.16b relativi alla prima codifica *uTxRLP*, si può constatare di come nel caso legacy la deviazione standard sia inferiore al caso EIP-1559 di 14.9419 cicli di clock. Anche i valori medi presentano delle differenze, infatti, nel caso legacy vi è un numero medio di cicli di clock pari a 27169.531 contro i 29618.064 del formato EIP-1559. Questo potrebbe essere attribuito alla diversa lunghezza in byte dei parametri che compongono la transazione legacy e dalla lunghezza della stringa viene soddisfatta una determinata regola definita dal processo di serializzazione, come indicato nella sezione 2.4. Nonostante ciò, i tempi medi valutati in entrambi i casi si possono considerare affini.



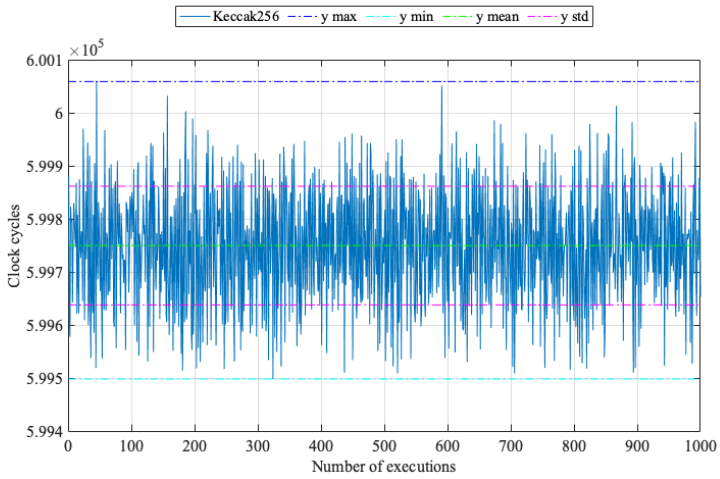
(a)



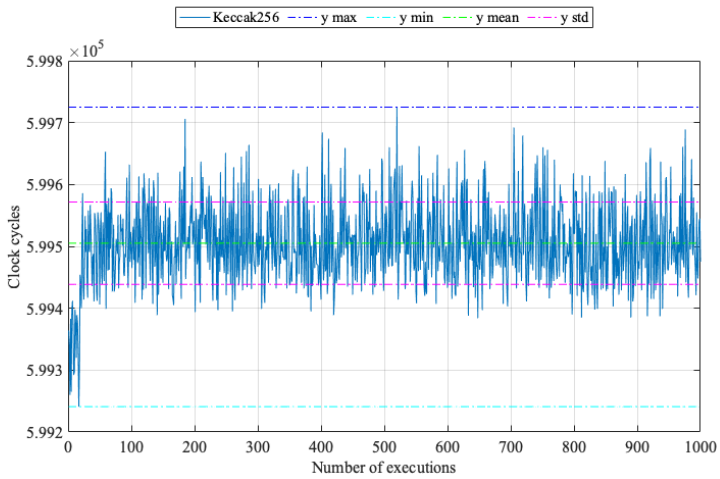
(b)

Figura 4.16: Cicli di clock della procedura di serializzazione *uTxRLP*, per la generazione di transazioni sia *Legacy* (a) che *EIP-1559*(b), misurati in 1000 esecuzioni.

Come già anticipato, l'operazione di serializzazione influenza la generazione dell'hash digest e osservando la deviazione standard di legacy si può essere assunta maggiore di 45.4201 cicli di clock. Dunque, vi è una variabilità maggiore del numero di cicli di clock nel caso legacy rispetto a quello EIP-1559, Figura 4.17. Seppur, anche in tal caso, non vi sia una differenza netta tra i tempi medi, ma solo uno scarto di 0.01 ms.



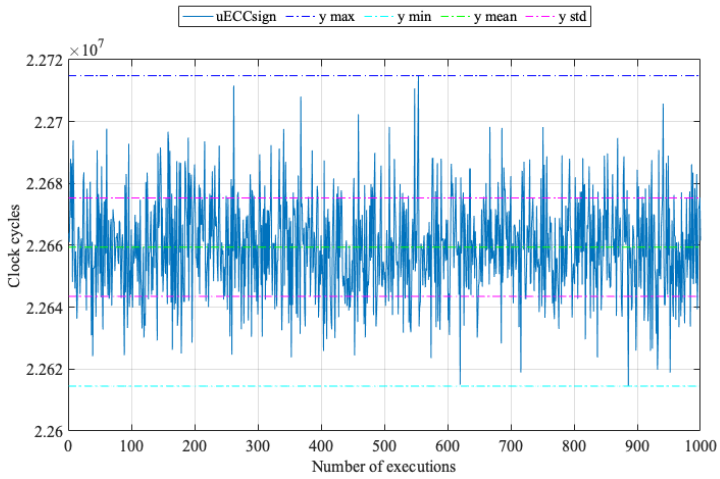
(a)



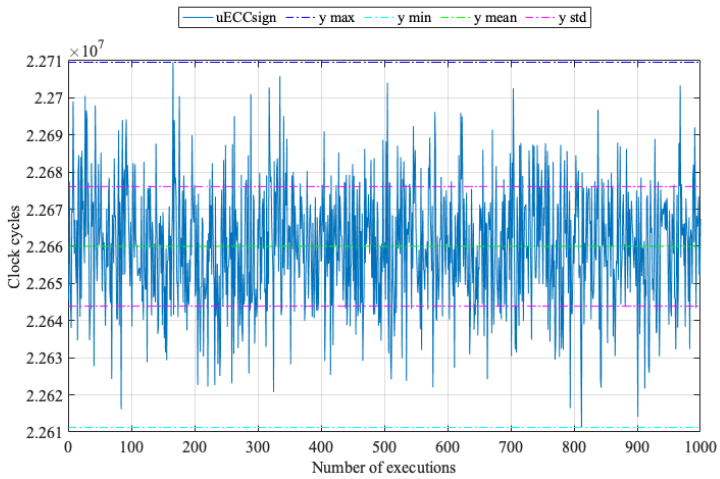
(b)

Figura 4.17: Cicli di clock della procedura *Keccak256*, per la generazione di transazioni sia *Legacy*(a) che *EIP-1559*(b), misurati in 1000 esecuzioni.

Per l'operazione di *uECCSign* in Figura 4.18, in entrambi i casi, si presentano i medesimi valori di deviazione standard. Quanto riscontrato non è un "evento" casuale, in quanto l'operazione algoritmica precedente non influisce sulla generazione della firma, l'input che viene fornito alla funzione di firma è sempre pari a 32 byte.



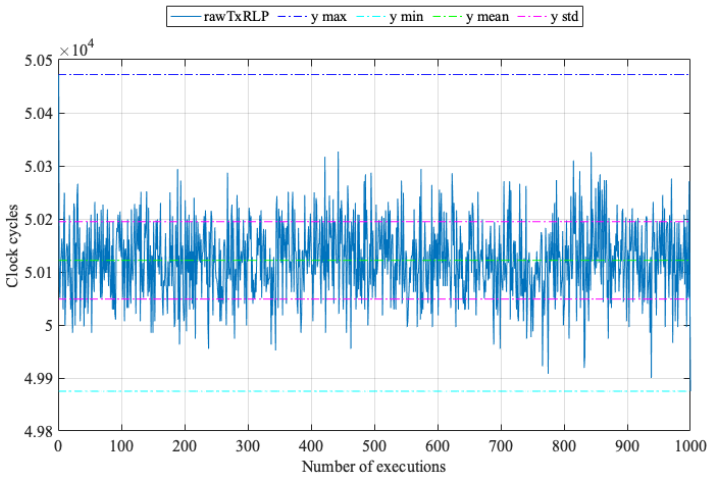
(a)



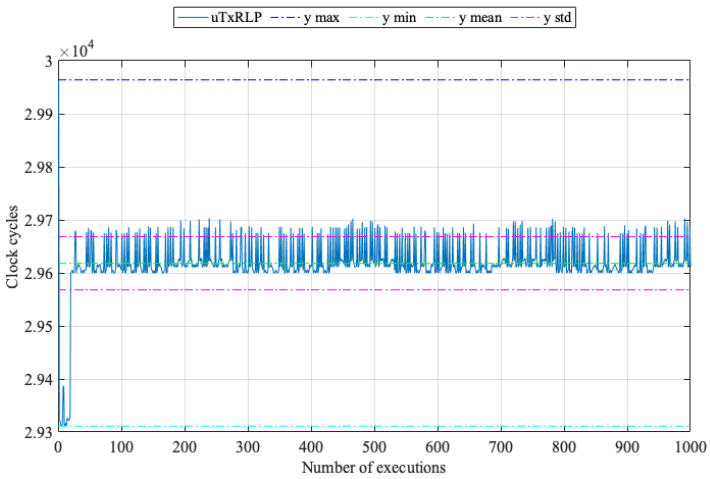
(b)

Figura 4.18: Cicli di clock della procedura di firma *uECCSign*, per la generazione di transazioni sia *Legacy*(a) che *EIP-1559*(b), misurati in 1000 esecuzioni.

Infine, per la valutazione dell'ultima operazione *rawTxRLP*, cui plot a confronto sono in Figura 4.19, le considerazioni sono similari a quelle definite poc'anzi rispetto *uTxRLP*. Infatti, dalla generazione delle transazioni legacy si misura una deviazione standard più contenuta e inferiore di 5.65 cicli di clock del caso EIP-1559. Discorso simile alla deviazione standard si può riproporre per il numero medio di cicli di clock, di fatto nel caso legacy si osserva una quantità più piccola di cicli di clock pari a 1955.477 rispetto al nuovo formato.
Nel complesso i tempi medi presentano uno scarto che può essere definito trascurabile.



(a)



(b)

Figura 4.19: Cicli di clock della procedura di codifica *rawTxRLP*, per la generazione di transazioni sia *Legacy*(a) che *EIP-1559*(b), misurati in 1000 esecuzioni.

Tabella 4.6: Confronto tra numero medio di cicli di clock richiesti, quantità di tempo e deviazione standard per ogni fase dell'algoritmo di generazione delle transazioni per i due formati di transazione.

Operazione	Formato Transazione	Tempo Medio (ms)	Numero Medio Cicli di Clock	Deviazione Standard
uTxRLP	Legacy	0.15	27169.531	35.1949
	EIP-1559	0.16	29618.064	50.1368
Keccak256	Legacy	3.33	599749.719	111.9077
	EIP-1559	3.32	599505.109	66.4876
uECCSign	Legacy	125.89	22659419.879	15915.110
	EIP-1559	125.89	22659986.893	16106.856
rawTxRLP	Legacy	0.28	50121.759	72.854
	EIP-1559	0.29	52077.236	78.502

Capitolo 5

TRASMISSIONE DI TRANSAZIONI VERSO GOERLI

Dopo aver valutato le performance del device rispetto alla generazione di transazioni conformi con Ethereum Beacon Chain, nell'ultima fase del lavoro viene eseguito il benchmarking dell'intero framework per analizzare le tempistiche riferite alla trasmissione delle transazioni legacy e secondo lo standard EIP-1559. In particolare, in questo capitolo vengono riportati:

- Tempi medi inerenti alla comunicazione seriale tra il device e il PC per la trasmissione delle raw transactions.
- Tempi medi inerenti alla comunicazione con la rete di prova Goerli.

5.1 Benchmarking Completo: Codice e Procedura

La trattazione che segue deve essere intesa come valida per entrambi i formati di transazione.

I testing prevedono la generazione di 100 raw transactions, partendo dalla richiesta del *Nonce*, da parte della scheda, dalla rete Goerli. Il *Nonce* viene fornito dalla rete tramite script Python (Listato 5.2) e successivamente incrementato di 1, stesso iter di quanto già approfondito nella sezione 4.2.
Ogni transazione generata viene inviata alla rete e affinché l'STM possa generarne di nuove, sino al limite di iterazioni impostate, attende un *ack*, come conferma che la procedura sia stata eseguita correttamente.
Questo è fondamentale per garantire una corretta sincronizzazione tra la board e la blockchain, altrimenti potrebbero verificarsi alcune inesattezze. Infatti, si è potuto constatare che eseguire i test per la trasmissione di raw transactions in tempi ravvicinati generi l'errore corrispondente a "Nonce too low" e conseguentemente le nuove transazioni non vengono inviate verso la rete.
Per comprendere tale errore bisogna considerare il *pending*, ovvero lo stato della transazione, che identifica un tempo di attesa di modo che la stessa sia accettata e inclusa nel blocco.
Dunque, è necessario attendere un un certo intervallo di tempo affinché il *Nonce* si aggiorni e possano essere trasmesse nuove transazioni.

Quanto scritto viene sintetizzato nelle seguenti righe di codice, presenti nel Listato 5.1, che compongono il loop di esecuzione principale dello script main.c, che è equivalente per i due file di progetto "GoerliTx_Legacy" e "GoerliTx_New", gli stessi di cui si è usufruito nella prima fase di benchmarking, citati nel Capitolo 4.

```
1
2
3 int main(void)
4 {
5 ...
```

```
 while (1)
  {
....

      uint32_t nonceCount = 0;
      ReceiveNonce(&nonceCount);
      for (iterationCount=0; iterationCount<100; iterationCount++) {
        UpdateNonce(nonceCount, nonce);
        PrepareTx_goerli(nonce, rawTx, &rawTx_charLength);
        TransmitTx(rawTx, rawTx_charLength);
        WaitAck();
        nonceCount++;
      }

  ....
  }
  ....
}
```

Listato 5.1: Codice relativo al programma d'esecuzione principale per la trasmissione delle transazioni verso Goerli: main.c

```


from web3 import Web3
import serial
import time

web3 = Web3(Web3.HTTPProvider('https://eth-goerli.g.alchemy.com/v2/<API>'))
from_address = '0x92Bdd4b0eC6f1b9A13f6959EC9859dDdcc2DE39B'
nonce = web3.eth.getTransactionCount(from_address)

print("Address: " + from_address)
print("Current nonce value is " + str(nonce))


# Initialize serial port connection
serialPort = serial.Serial(port="/dev/tty.usbmodem14103", baudrate=115200)
print (serialPort)

# First send number of chars to represent nonce
nonce_bytes = str(nonce).encode()
nonce_size = str(len(nonce_bytes)).encode();
serialString = serialPort.write(nonce_size)

serialString = serialPort.write(nonce_bytes)

serialString = ""  # Used to hold data coming over UART

time_serial = 0;
```

```
time_goerli = 0;

num_tx = 0
start_board = time.time()

while (num_tx < 100):

    # Wait until there is data waiting in the serial buffer
    if (serialPort.in_waiting > 0):
        # Read data out of the buffer until a carraige return / new
    line is found
        serialString = serialPort.readline()

        end_board = time.time();

        time_serial += (end_board - start_board);
        # rawTx = str(serialString.decode('Ascii'))
        # Print the contents of the serial data
        # print(serialString);
        # print(serialString.decode('Ascii'))
        serialString_Ascii = serialString.decode('Ascii');
        print(serialString_Ascii)
        rawTx = "0x";
        i = 0;
        if len(serialString_Ascii) > 0:
            if serialString_Ascii[i] == '\x00':
                i = 1;
            else:
                i = 0;
            while serialString_Ascii[i] != '\x00':
                rawTx = rawTx + serialString_Ascii[i];

                i = i + 1;
            num_tx += 1;

            # Print rawTx
            nonce = nonce + 1;
             print("Tx # " + str(nonce) + ", RawTx = " + rawTx)


            start_goerli = time.time();
            #print(web3.utils.isAddress(from_address));
            a = web3.eth.send_raw_transaction(rawTx);



            end_goerli = time.time();

            time_goerli += end_goerli- start_goerli;
            # Tell the device connected over the serial port that we
    recevied the data!
```

```
78              ack_str = "0";
79              serialString = serialPort.write(ack_str.encode())
80              # The b at the beginning is used to indicate bytes!
81              print("Transaction accepted, receipt is ")
82              print(a.hex())
83              print("Time to receive rawTx = " + str(end_goerli -
    start_goerli) + " s");
84              print("Time to send to Goerli = " + str(end_goerli -
    start_goerli) + " s");
85              print("-----------------------------------------------");
86     start_board = time.time()
87
88 print("Average SerialComm time = " + str(time_serial / 100));
89 print("Average Goerli time = " + str(time_goerli / 100));
```

Listato 5.2: Script Python per l'acquisizione e trasmissione delle transazioni verso Goerli

5.2 Misura dei tempi per la trasmissione delle transazioni legacy

Nelle prove sono stati utilizzati account differenti da cui far partire le transazioni. Dalle 100 trasmissioni è stato possibile ricavare, usufruendo delle righe di codice presenti nel Listato 5.2, sia i tempi medi relativi alla ricezione delle transazioni generate dalla board e inviate al PC che i tempi medi relativi all'interazione con la rete.

Si precisa che il parametro *Value* che compone la transazione è stato modificato, rispetto a quanto indicato nel capitolo precedente, ad una quantità nulla per evitare costi onerosi, anche se, essendo una rete di prova, la criptovaluta ad essa legata non ha alcun valore economico.

- Tempo medio per la ricezione, da parte del PC, di ogni nuova raw transaction: ≈ 0.020 s.
- Tempo medio per l'invio di ogni nuova raw transaction verso la rete di prova Goerli: ≈ 0.367 s.

Si evince che il tempo medio per la comunicazione seriale è inferiore rispetto al tempo medio relativo all'interazione con Goerli. Questo permette di affermare che l'implementazione del lavoro sulla configurazione scelta rispetta il giusto compresso temporale tra la preparazione/trasmissione delle transazioni e il riscontro da parte della rete.

5.3 Misura dei tempi per la trasmissione delle transazioni EIP-1559

Per la trasmissione di 100 transazioni relative al nuovo formato, note anche come di tipo 2, valgono le medesime considerazione di cui sopra.

In tal caso, sono stati eseguiti due testing differenti, andando a modificare il parametro di *Max Priority Fee per Gas*, che rappresenta la mancia per i validatori.

Trasmissione di transazioni impostando una *Max Priority Fee per Gas* pari a 5 Gwei:

- Tempo medio per la ricezione, da parte del PC, di ogni nuova raw transaction: ≈ 0.020 s.
- Tempo medio per l'invio di ogni nuova raw transaction verso la rete di prova Goerli: ≈ 0.260 s.

Trasmissione di transazioni impostando una *Max Priority Fee per Gas* pari a 2 Gwei:

- Tempo medio per la ricezione, da parte del PC, di ogni nuova raw transaction: ≈ 0.020 s.
- Tempo medio per l'invio di ogni nuova raw transaction verso la rete di prova Goerli: ≈ 0.265 s.

Le considerazioni sulle tempistiche in questione sono analoghe a quelle relative alle transazioni legacy ed è lecito che il tempo per la comunicazione seriale sia equivalente per entrambi i formati.
Confrontando i tempi medi relativi all'invio delle due tipologie di transazione si può assumere che nel caso EIP-1559 vi sia un vantaggio di circa 0.047 s.

Un vantaggio rilevante delle transazioni che seguono lo standard EIP-1559 consiste in una quantità deifnita come *Txs saving*. Quest'ultima è possibile visionarla quando si interroga la rete Goerli[1], inserendo l'address di interesse, dalla dashboard relativa alla transazione registrata.
Per definizione *Txs saving* corrisponde alle commissioni totali risparmiate rispetto all'importo che l'utente è disposto a pagare per una determinata transazione.

In particolare, viene calcolata come segue:

$$Txs\ saving = (MaxFeeperGas - MaxPriorityFeeperGas - BaseFee) \times GasUsed$$

[1] https://goerli.etherscan.io

Dunque, nonostante dalle tempistiche valutate non si possa affermare che vi sia un netto miglioramento passando da un formato di transazione all'altro, la digressione appena conclusa vuole mettere in evidenza che dall'innovativo tipo di transazione possono essere ricavati alcuni benefici in termini di costi.

CONCLUSIONI

Il lavoro di tesi proposto si compone di varie fasi con lo scopo di sviluppare funzionalità conformi con Ethereum Beacon Chain.
In prima analisi, sono stati esaminati i cambiamenti presenti sulla rete Ethereum a valle del cambiamento del protocollo di consenso, evidenziando alcune variazioni in termini di numero di blocchi, tempo medio per validare i blocchi e una diversa dimensione media del blocco. Considerando questa trasformazione, si è valutato se l'annessione di un diverso protocollo di consenso potesse avere degli effetti sull'implementazione proposta. Inoltre, la rete non ha subito dei cambiamenti unicamente dal punto di vista del protocollo, ma anche per quanto riguarda i formati di transazione. La transazioni standard, definite legacy, hanno fatto spazio ad un nuovo formato, che è stato sviluppato e messo a punto dallo standard EIP-1559. Questo nuovo formato è stato introdotto principalmente per far fronte alle problematiche scaturite dal meccanismo ad asta, alla base delle transazioni legacy. Seppur questo nuovo formato sia stato introdotto prima della cosiddetto *merge* della Beacon Chain con Ethereum, è parte integrante di un lungo processo di sviluppo, che si assume possa essere definitivo entro un anno.

Il progetto presentato coinvolge il dispositivo basato su core ARM CORTEX-M4 e diversi script in linguaggio C, Matlab e Python.
Le sperimentazioni iniziali hanno permesso di studiare le performance del dispositivo considerando la libreria micro-ecc. Usufruendo della libreria in questione, sono state valutate le funzioni che compongono l'algoritmo ECDSA, che è alla base delle transazioni Ethereum. Questo ha permesso di ottenere delle prove a conferma che l'implementazione progettata sia effettivamente fattibile sulla piattaforma scelta. Stabilito ciò, la fase di benchmarking successiva è stata caratterizzata dalla generazione delle transazioni per il formato legacy e il formato EIP-1559. I due formati differiscono per i parametri che compongono la transazione, per l'aggiunta del prefisso "02" che permette alla rete di distinguere in maniera univoca il formato EIP-1559 da quello legacy e per la diversa impostazione e calcolo del parametro di firma v. La generazione delle transazioni ha conseguentemente portato alla trasmissione delle stesse, verso la rete di prova Goerli, alla valutazione delle misurazioni dei tempi medi relativi all'invio delle transazioni generate dalla board verso il PC e ai tempi medi relativi all'invio delle transazioni verso la rete. Considerando che, il codice che ha consentito la generazione e trasmissione delle transazioni si può definire affine per entrambi i formati, a meno di alcune modifiche in linea con le differenze messe in evidenza poc'anzi, non è stato inaspettato osservare che le tempistiche legate alla comunicazione seriale fossero analoghe per entrambi i tipi di transazione. Per quanto riguarda, invece, la trasmissione delle transazioni verso Goerli, tra i tempi confrontati non vi è uno scarto tale da poter affermare quale delle due configurazioni sia più confacente in termini di throughput. Tuttavia, spostando l'analisi dal punto dal punto di vista dei costi, si può assumere che se viene trasmessa una transazione conforme EIP-1559 è possibile osservare, lato rete, una quantità che identifica un termine di

risparmio, rispetto alla quantità di fee che l'utente avrebbe speso altrimenti.

Questo però, non implica che i costi del gas, conseguentemente al *merge*, siano più economici, perché questi dipendono dalla domanda, infatti "Le tariffe del gas sono un prodotto della domanda di rete rispetto alla capacità della rete".

Il cambiamento del meccanismo di consenso ha posto le basi per gli aggiornamenti del prossimo futuro, come lo Sharding, grazie al quale si potrà assistere ad un maggior capacità e scalabilità della rete, comportando una riduzione non indifferente dei costi in gioco.

Sviluppi ulteriori al lavoro di tesi in questione potrebbero interessare l'ottimizzazione dell'implementazione presentata, in relazione al nuovo aggiornamento che permetterà di sfruttare, in maniera concreta, dei benefici introdotti dai nuovi formati di transazione e dalla nuova logica del consenso.

Bibliografia e Sitografia

[1] URL: https://www.oracle.com/internet-of-things/what-is-iot/.

[2] Sarvesh Tanwar et al. *Next Generation IoT and Blockchain Integration*. 2022. URL: https://www.hindawi.com/journals/js/2022/9077348/#abstract.

[3] Joseph Yiu. *Cortex-M Processors and the Internet of Things (IoT) Background of the Cortex-M processors*. 2013.

[4] *Bitcoin*. URL: https://it.wikipedia.org/wiki/Bitcoin.

[5] Yong Wang et al. "Blockchain data structure consisting of three blocks." In: (2021). URL: https://plos.figshare.com/articles/figure/Blockchain_data_structure_consisting_of_three_blocks_/13628185.

[6] *PROOF OF WORK*. URL: https://ethereum.org/it/developers/docs/consensus-mechanisms/pow/.

[7] URL: https://www.it-impresa.it/blog/ambiti-applicativi-blockchain/.

[8] *Standards for Efficient Cryptography*. 2010. URL: https://www.secg.org/sec2-v2.pdf.

[9] *ECDSA*. URL: https://web.archive.org/web/20170921160141/http://cs.ucsb.edu/~koc/ccs130h/notes/ecdsa-cert.pdf.

[10] Zhiqiang Ouyang, Jie Shao, and Yifeng Zeng. "PoW and PoS and Related Applications". In: (2021), pp. 59–62. DOI: 10.1109/EIECS53707.2021.9588080.

[11] *The Beacon Chain*. URL: https://ethereum.org/en/upgrades/beacon-chain/#content.

[12] URL: https://www.coindesk.com/learn/top-questions-about-proof-of-stake-and-staking-answered/.

[13] Giulia Rafaiani et al. "Implementation of Ethereum Accounts and Transactions on Embedded IoT Devices". In: *2022 IEEE International Conference on Omni-Layer Intelligent Systems, COINS 2022* (2022). DOI: 10.1109/COINS54846.2022.9855005.

[14] Berlin Version. *Ethereum: Yellow Paper*. URL: https://ethereum.github.io/yellowpaper/paper.pdf.

[15] Daniël Reijsbergen et al. "Transaction Fees on a Honeymoon: Ethereum's EIP-1559 One Month Later". In: *2021 IEEE International Conference on Blockchain (Blockchain)*. 2021, pp. 196–204. DOI: 10.1109/Blockchain53845.2021.00034.

[16] URL: https://medium.com/@markodayansa/a-comprehensive-guide-to-rlp-encoding-in-ethereum-6bd75c126de0.

[17] *STM32CubeIDE*. URL: https://www.st.com/en/development-tools/stm32cubeide.html.

[18] *STM32CubeMX*. URL: https://www.st.com/en/development-tools/stm32cubemx.html.

[19] *NODES AND CLIENTS*. URL: https://ethereum.org/en/developers/docs/nodes-and-clients/.

[20] URL: https://www.st.com/content/ccc/resource/training/technical/product_training/group0/79/d3/43/58/03/b4/44/be/STM32MP1-Security-Random_Number_Generator_RNG/files/STM32MP1-Security-Random_Number_Generator_RNG.pdf/jcr:content/translations/en.STM32MP1-Security-Random_Number_Generator_RNG.pdf.

[21] "ARMv7-M Architecture Reference Manual". In: (). URL: https://developer.arm.com/documentation/ddi0403/d/Debug-Architecture/ARMv7-M-Debug/The-Data-Watchpoint-and-Trace-unit/Cycle-Count-register--DWT-CYCCNT?lang=en.

[22] Andrea Gilibertii. *Blockchain transactions implementation on ARM Cortex-M4 platform*.

[23] *X-CUBE-CRYPTOLIB*. URL: https://www.st.com/en/embedded-software/x-cube-cryptolib.html.

[24] *micro-ecc*. URL: https://github.com/kmackay/micro-ecc.

[25] *Introduction Description of STM32F4 HAL and low-layer drivers UM1725 User manual*. URL: www.st.com.

[26] *Ethereum-RLP*. URL: https://github.com/KingHodor/Ethereum-RLP.

[27] *sha3*. URL: https://github.com/firefly/wallet/tree/master/source/libs/ethers/src.

[28] URL: https://chainlist.org.

[29] URL: https://github.com/Marsh61/ECDSA-Nonce-Reuse-Exploit-Example.

RINGRAZIAMENTI

Un ringraziamento particolare al Professor Marco Baldi per avermi seguita in questi mesi senza mai farmi mancare il suo sostegno e i suoi preziosi suggerimenti, aiutandomi e accompagnandomi nella realizzazione della tesi. La mia decisione in merito alla scelta del percorso di studi, dopo la triennale, la devo anche e sopratutto al forte interesse che avevo nell'interfacciarmi con il suo corso. Ad oggi posso dire, grazie ai suoi insegnamenti, di voler continuare ad approfondire e mettere in pratica quel che ho appreso da lei durante tutti i mesi di lezione.

Un ringraziamento speciale al Dott. Paolo Santini e alla Dott.ssa Giulia Rafaiani che in questo lavoro hanno avuto un ruolo fondamentale: mi hanno accompagnata per tutto il percorso, insegnandomi e chiarendomi dubbi e incertezze ogni qualvolta ne ho avuto bisogno. Senza il vostro prezioso aiuto non sarei giunta alla compimento di questo lavoro, grazie davvero.

Ai miei genitori, che sono il mio punto di riferimento, che mi hanno sostenuta con amore e dedizione e che mi hanno permesso di percorrere e concludere questo cammino. Lo ritengo un privilegio inestimabile.
Ai miei piccoli (ormai non più) fratelli, Alessia e Tommaso e al mio fratello "acquisito" Daniele. Grazie perché senza di voi non sarei mai arrivata fino in fondo a questo difficile, lungo e tortuoso cammino. Questa tesi la dedico a voi che siete la mia famiglia, il mio più grande sostegno e la mia guida.

Grazie alla mia cara e super nonna, per la fortuna che ho nell'averla nella mia vita, per l'amore che mi ha saputo donare e per l'appoggio che non mi ha mai fatto mancare.
Ringrazio anche i miei zii, Donato e Oriana, per il loro interesse e affetto avuto nei miei confronti. Li ringrazio perché hanno partecipato anche loro in questo percorso continuando a starmi vicina.

Voglio ringraziare una persona unica e speciale, Francesco, la persona che più di tutte, con il suo amore, è stata capace di capirmi e di sostenermi nei momenti difficili. Grazie a te ho avuto il coraggio di mettermi in gioco e di capire che, in fondo, gli ostacoli esistono per essere superati. Hai affrontato insieme a me questo cammino, giorno dopo giorno, festeggiando insieme ogni vittoria e affrontando ogni sconfitta. In questi anni ci siamo sempre sostenuti e incoraggiati l'un l'altra.

Un grazie che non avrà mai fine a Sara, la migliore amica, la miglior compagna di avventure e di mille risate che si possa desiderare. In questi anni è sempre stata presente in ogni momento, anche a chilometri di distanza, mi ha saputa apprezzare per come sono, con i miei pregi e soprattutto i miei difetti; rispettando i miei silenzi e sopportando le mie "fisse" di ogni genere. Un'amica sincera e leale che mi ha fatto riscoprire la bellezza di avere una persona a cui poter confidare senza "riserve" i miei

pensieri e le mie emozioni.