Università Politecnica delle Marche
Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Elettronica

Implementazione di transazioni blockchain Ethereum su processori embedded ARM Cortex-M4

Implementation of Ethereum blockchain transactions on ARM Cortex-M4 embedded processors

Relatore:
Chiar.mo Prof. Marco Baldi

Tesi di Laurea di:
Daniela Voltattorni

Correlatori:
Dott. Paolo Santini
Dott.ssa Giulia Rafaiani

A.A. 2020/2021

Indice

Introduzione

L'avvento della digitalizzazione negli ultimi anni ha avuto un grosso impatto in diversi settori produttivi, in particolare in quello industriale; è proprio in questo contesto che nascono le tecnologie legate all'Industria 4.0 e all'Industrial Internet of Things (IIoT), introducendo un nuovo modello produttivo.
L'Industria 4.0, il cui nome fa riferimento alla quarta rivoluzione industriale, pone le sue basi sul concetto di smart factories, ossia sistemi indipendenti ed automatici capaci di interagire tra di loro e con altre fabbriche; l'obiettivo di questa nuova modalità di produzione è quello di favorire una comunicazione macchina-macchina (M2M), promuovendo lo sviluppo di macchinari intelligenti, di tecniche di elaborazione di grandi quantità di dati (big data) e di dispositivi IoT (Internet of Things) e, dall'altro lato, di limitare il lavoro manuale a favore della robotica. La teoria alla base dell'IIoT è incentrata sul fatto che i macchinari e dispositivi smart interconnessi tra di loro siano effettivamente più accurati, efficienti ed economici rispetto alla stessa mole di lavoro svolta da un essere umano [1]. Questa modernizzazione del sistema produttivo porta a numerosi vantaggi, primo tra tutti una riduzione degli errori e delle spese del personale grazie alla diffusione del machine learning, un insieme di tecniche che permettono ad una macchina intelligente di apprendere in modo autonomo e individuare e correggere eventuali malfunzionamenti di sistema; in secondo luogo, l'utilizzo di questi metodi permette alle imprese di aumentare la produzione, di riorganizzarla e adattarla in real time alle esigenze dei clienti; infatti i "big data", ossia il fenomeno di gestire e analizzare grandi quantità di dati provenienti da dispositivi IoT e smart, sono usati per raccogliere informazioni sui consumatori e sui macchinari delle linee di produzione per decidere le strategie aziendali e di marketing da perseguire.
Lo sviluppo dell'Industria 4.0 conduce dunque alla realizzazione di ambienti produttivi flessibili e altamente efficienti grazie all'implementazione di soluzioni tecnologiche con lo scopo di incrementare l'attività di produzione automatizzata e, contemporaneamente, ridurre l'intervento umano, anche nei processi decisionali e logistici.
Uno dei pilastri portanti dell'Industria 4.0 sono le piattaforme Internet of Things, ossia l'insieme di sensori e sistemi di elaborazione in grado di interconnettere e garantire la circolazione di dati tra macchina e l'uomo; l'introduzione di dispositivi IoT in ambito industriale, tuttavia, presenta alcuni problemi:

- i dati raccolti dai sensori vengono memorizzati in determinati data center che sono esposti a rischi di attacco e divulgazione di informazioni e a guasti che possono corrompere i dati stessi;
- durante la comunicazione tra due o più dispositivi IoT i dati possono essere intercettati.

Per ovviare a tali inconvenienti che coinvolgono sia la fase di memorizzazione che la fase di trasmissione di dati sensibili, è necessario implementare una tecnica che assicuri sicurezza nelle transazioni e un controllo degli accessi e dei permessi; tutti i dispositivi connessi a Internet, infatti, richiedono un sistema di protezione sicuro, distribuito e leggero.
A fronte di questi bisogni in ambito industriale si può far uso della blockchain [2];

questa nuova tecnologia è costituita da un registro immutabile, distribuito e decentralizzato in cui i dati, una volta trascritti, non possono più essere modificati o eliminati. La blockchain pone dunque le basi per rivoluzionare quella che è l'interazione uomo-macchina e lo fa garantendo da un lato l'integrità e una gestione efficiente dei dati, dall'altro assicurando la condivisione informativa tra i soggetti autorizzati all'interno di una stessa rete distribuita, tutelando al tempo stesso la privacy delle identità di rete.
In questo contesto si sviluppa il lavoro di tesi qui presentato, che si pone come obiettivo quello di creare un nodo blockchain su un dispositivo IoT, nello specifico un microcontrollore. Questa piattaforma embedded, applicata ad esempio in un ambiente industriale, può ricevere diversi pacchetti di dati da uno o più sensori ad esso collegati; da qui l'esigenza di archiviare tutte le informazioni ricevute in un unico ledger, che non consenta modifiche e che sia accessibile ad altri nodi in base ad alcuni parametri di autorizzazione, che dipendono dalla tipologia di blockchain scelta. Il nodo implementato sul dispositivo IoT deve essere in grado di eseguire alcune operazioni elementari che sono alla base delle operazioni crittografiche della blockchain, come la generazione di una coppia di chiavi, la creazione di un payload, la firma digitale dei dati con relativa verifica e infine l'implementazione di una transazione vera e propria. La transazione generata da questo processo verrà poi inviata a un altro nodo della rete che, tramite il processo di mining, ne verificherà la validità e correttezza, aggiungendola al registro distribuito in caso affermativo.

La struttura dell'elaborato è articolata nel seguente modo:

- Nel primo capitolo viene fornita una panoramica generale sul funzionamento e sulle principali caratteristiche della blockchain; vengono inoltre descritti gli utilizzi e i vantaggi che questa tecnologia può apportare nel settore industriale e manifatturiero.

- Nel secondo capitolo si passa a esaminare Ethereum, la piattaforma blockchain scelta per l'implementazione, e vengono approfonditi alcuni aspetti tecnici alla base del funzionamento del sistema.

- Nel terzo capitolo vengono illustrate le componenti software e hardware su cui è stato implementato il nodo della rete Ethereum.

- Nel quarto capitolo viene esposto e descritto il codice C realizzato per la creazione del nodo e la generazione di transazioni Ethereum; nel dettaglio, si pone l'attenzione sugli algoritmi per la generazione della coppia di chiavi, per la firma digitale e la verifica.

- Infine, il quinto capitolo illustra le prestazioni del microcontrollore utilizzato sulla base delle funzioni che vengono impiegate per la generazione di transazioni; viene inoltre mostrato un confronto, a livello di prestazioni, tra l'implementazione di due diverse librerie, X-CUBE CRYPTO e MicroECC.

1 Aspetti generali della tecnologia Blockchain

Per poter meglio comprendere il processo di implementazione di un nodo su microcontrollore, oggetto di questa tesi, è necessario fornire una panoramica generale sulla tecnologia blockchain e sulle sue caratteristiche tecniche. In particolare, in questo capitolo viene data una descrizione dei fondamenti della tecnologia, delle diverse infrastrutture di rete e del processo per la validazione delle transazioni, per proseguire poi con una breve parentesi sui vantaggi apportati da questo sistema innovativo nel mondo industriale.

1.1 Caratteristiche generali

La tecnologia blockchain, proposta per la prima volta nel 2008 e implementata l'anno successivo, venne presentata da Satoshi Nakamoto nel suo white paper come la digitalizzazione di un libro mastro per l'allora nuova criptovaluta Bitcoin; l'obiettivo di Satoshi era quello di creare un sistema di pagamento elettronico basato su una prova crittografica anziché sulla fiducia, permettendo a due enti di negoziare direttamente tra loro senza la necessità di intermediari come banche o istituti finanziari [3].
La blockchain supera dunque il concetto di centralizzazione di un'autorità terza mediatrice e si pone come un database distribuito, strutturato come una catena di transazioni marcate temporalmente che non possono essere cancellate o modificate. Rispetto ad un sistema centrale, la blockchain è un registro pubblico distribuito: questo vuol dire che l'accesso è consentito a più nodi di una stessa rete e che questi ultimi detengono una copia dell'intero registro delle transazioni; le caratteristiche innovative di tale tecnologia decentralizzata sono le seguenti:

- trasparenza dei dati e del processo di aggiornamento di essi; le transazioni effettuate infatti sono visibili a tutti i nodi della rete blockchain;
- integrità delle informazioni, che vengono protette da modifiche non autorizzate in modo da garantire attendibilità e solidità dei dati [4];
- tracciabilità delle transazioni per cui, data una catena di blocchi, è possibile consultare la "storia" di ogni transazione e risalire alla prima transazione generata;
- non ripudiabilità dei dati; questa caratteristica deriva dal fatto che tutte le informazioni che vengono registrate all'interno del ledger, essendo inalterabili, sono autentiche;
- immutabilità, in quanto le transazioni non sono modificabili, essendo tutti i blocchi collegati tra loro; come conseguenza questo aspetto garantisce le proprietà di integrità dei dati e di non ripudiabilità.

Queste caratteristiche permettono non solo la possibilità di archiviare informazioni, come transazioni e smart contract, senza passare per un'istituzione centrale ma anche di poterlo fare in sicurezza, essendo il sistema incorruttibile.
La blockchain si presenta come una catena di blocchi sequenziali, in ciascuno dei

quali vengono registrate un insieme di transazioni. Come si osserva dalla figura 1, ogni blocco risulta inscindibilmente legato al precedente da una stringa generata attraverso una funzione crittografica di hash, usata per rendere sicura la trascrizione e il trasferimento di criptovalute e dati sensibili. In questo modo, a partire dal blocco corrente è possibile risalire fino al blocco genesi, ossia al primissimo blocco generato e visualizzare l'intera storia delle transazioni della blockchain. Ciascun blocco contiene, oltre a un numero ben definito di transazioni, un intestazione (*header*) che include un hash del blocco corrente, l'hash del blocco precedente, un riferimento all'istante temporale in cui è stata inserita nel blocco l'ultima transazione (*timestamp*) e un *nonce*, ossia un numero pseudo casuale che viene utilizzato come contatore nel processo di mining [5]. E' proprio la presenza dell' hash del blocco precedente che rende immutabile la blockchain e impedisce la modifica o l'eliminazione dei blocchi già convalidati da parte di un utente malevolo.

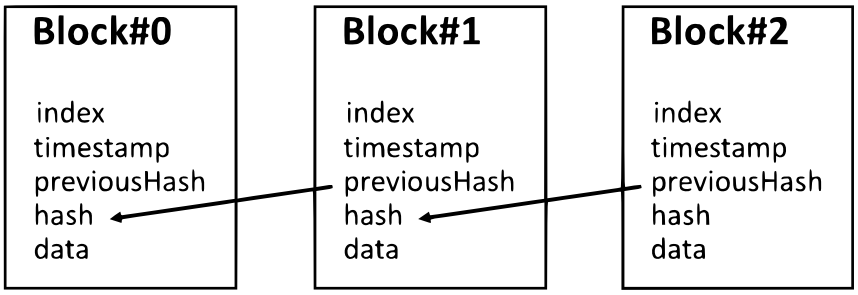


Figura 1: *Struttura e legame dei blocchi di una blockchain*

Per poter inviare o ricevere una transazione, ogni utente della blockchain deve disporre di un *wallet*, ossia un account in cui vengono memorizzate la chiave privata e pubblica e il corrispondente indirizzo associato a quel wallet. L'utente che intende inviare dati o valute a un altro nodo deve firmare la transazione con la propria chiave privata e conoscere la chiave pubblica del destinatario; prima di essere registrata nel ledger, tuttavia, la transazione deve essere sottoposta a una fase di validazione da parte della rete, processo che assicura da un lato che la transazione sia stata messa sulla rete dall'effettivo proprietario dell'account e, dall'altro, che la stessa transazione dal medesimo mittente non avvenga più volte. Solo il raggiungimento del consenso distribuito porta all'inserimento all'interno della catena delle transazioni, raggruppate in blocchi, e alla propagazione delle stesse attraverso la rete.

1.2 Architettura di rete

Come è già stato anticipato, la blockchain è un *ledger* digitale decentralizzato e strutturato in modo tale che tutti gli utenti (detti anche *nodi*) appartenenti alla rete possano accedere al sistema e compiere operazioni di lettura, scrittura, convalida e invio di transazioni verso altri utenti. Tutti i nodi della rete, dunque, sono interconnessi tra loro in modo da formare un sistema paritario peer-to-peer e privo di gerarchie di tipo client-server in cui tutti i partecipanti possiedono una copia dell'intera blockchain. A tal proposito è utile fare una distinzione tra le due diverse tipologie di nodo che è possibile implementare sulla singola macchina connessa alla rete blockchain: i *full nodes* e i *light nodes*. I *full nodes*, o nodi completi, sono nodi

del tutto indipendenti che detengono localmente una copia completa del libro mastro e che sono quindi in grado di visualizzare tutte le transazioni a partire dal blocco genesi; tutte le operazioni di mining e di verifica circa la correttezza delle transazioni e dei blocchi vengono svolte dai full nodes e, per quanto siano dispendiosi e richiedano una potenza di calcolo elevata per poter scaricare i dati dell'intera blockchain, essi rappresentano una garanzia di sicurezza grazie alla loro capacità di non doversi appoggiare a terze parti per le operazioni di controllo e verifica. Dall'altro lato troviamo i *light nodes* o nodi leggeri, ossia nodi che non memorizzano l'intera cronologia delle transazioni ma scaricano solamente le intestazioni dei blocchi (*header*) di interesse, appoggiandosi a un nodo completo per la richiesta di queste informazioni; in sostanza, per potersi collegare alla rete i light nodes devono necessariamente riporre la fiducia in un altro nodo, senza il quale non sarebbero capaci di verificare in maniera indipendente la validità delle transazioni e dei blocchi. Se da una parte questa tipologia di nodo è del tutto dipendente da terze parti e quindi poco sicuro, dall'altra rappresenta una soluzione molto comoda ed economica se si desidera eseguire un numero limitato di operazioni, in quanto i nodi leggeri contengono ed elaborano meno dati rispetto ai nodi completi e richiedono una minore potenza computazionale per essere implementati; per tale ragione i light node sono molto spesso dei *wallet* che vengono installati su dispositivi mobili o piattaforme embedded con spazio di archiviazione ridotto [6].

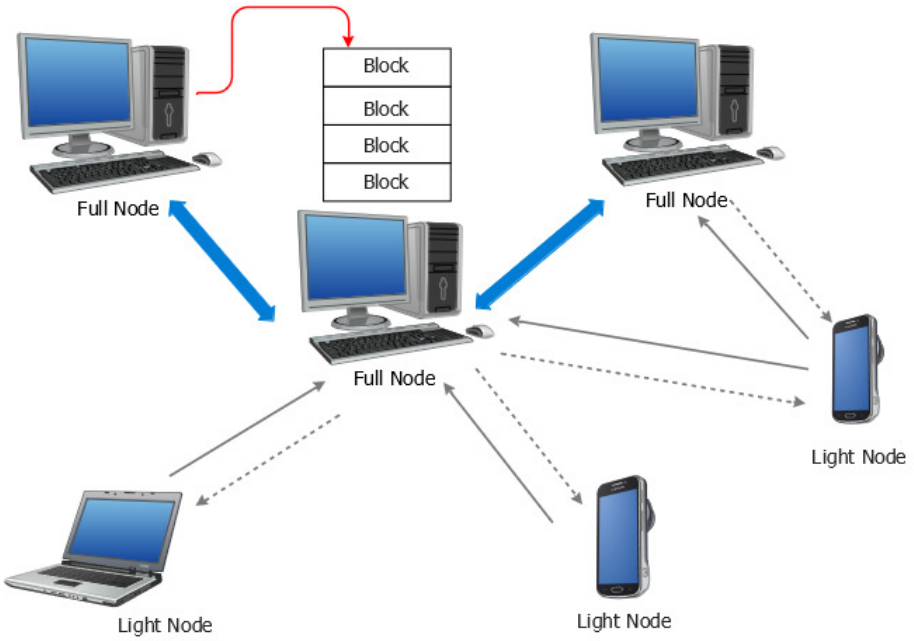


Figura 2: Architettura di una rete blockchain che include full nodes e light nodes

In una qualsiasi rete blockchain sono presenti in genere entrambe le tipologie di nodo: i full nodes come garanti di sicurezza del sistema e per lo svolgimento dei processi di mining e di convalida delle transazioni secondo le regole del consenso, mentre i light nodes per l'esecuzione di operazioni più limitate che non richiedono la visualizzazione dell'intera catena. Come si osserva dalla figura 2, i full node interagiscono tra loro per controllare continuamente e aggiornare in modo univoco e sicuro, una volta raggiunto il consenso, il registro delle transazioni. A loro volta i

full nodes possono essere interconnessi con uno o più nodi light ai quali trasmettono le informazioni di interesse come il timestamp e il numero identificativo (nonce) di un singolo blocco; d'altra parte i light nodes possono creare e inviare transazioni grezze ai nodi completi, i quali le convalidano e verificano prima di inserirle nella blockchain.
I nodi di diverse reti possono avere autorizzazioni e differenziarsi per la gestione del meccanismo di consenso; infatti, a seconda dell'attribuzione dei permessi e dell'autorità, esistono tre principali modelli di blockchain: blockchain pubblica, privata e consortium.

- Blockchain pubblica (*permissionless*): è un modello di blockchain totalmente decentralizzata in cui qualsiasi nodo può accedere, inviare e validare transazioni e partecipare al processo di consenso; essendo una rete aperta a chiunque e priva di autorizzazioni, per garantire la sicurezza di questo sistema peer-to-peer vengono conferiti incentivi economici ai miner per ripagarli dall'enorme quantità di potenza di calcolo messa a disposizione per la validazione dei blocchi [7]; le blockchain pubbliche più note sono Bitcoin ed Ethereum in cui i blocchi e le transazioni sono visibili a tutti senza restrizioni;

- consortium (*public permissioned*): è un modello ibrido tra la blockchain pubblica e privata in cui viene sacrificata la completa decentralizzazione dell'architettura a favore di un controllo sugli accessi, sui permessi e visibilità dei nodi da parte di un ristretto numero di utenti. In genere la visualizzazione del registro e la trasmissione di transazioni è consentito a chiunque mentre il mining è soggetto ad autorizzazione, per cui solo pochi nodi "eletti" possono partecipare alla validazione dei blocchi; alcuni esempi di blockchain pubbliche ma permissionate sono Hyperledger Fabric, Hyperledger Besu e Ripple;

- blockchain privata (*private permissioned*): sono delle reti private e non visibili a cui è possibile accedere solo mediante autorizzazione. Questa blockchain è caratterizzata dalla presenza di un'organizzazione centrale che controlla tutti i permessi relativi all'accesso, alla lettura e scrittura dei dati e al mining dei blocchi; Multichain rappresenta un esempio di blockchain privata.

La scelta della tipologia di blockchain dipende dall'utilizzo che se ne vuole fare: per lo scambio di criptovalute e di dati non sensibili in modo anonimo si possono impiegare blockchain pubbliche, garanti di un maggior grado di sicurezza delle informazioni contenute nel registro grazie all' elevato numero di utenti che partecipano al processo di verifica e validazione delle transazioni; per piccole realtà aziendali, in cui la priorità è quella di avere uno stretto controllo sui partecipanti del network e limitare la diffusione dei dati a una ristretta cerchia di utenti, è invece preferibile adottare modelli di blockchain private, che garantiscono una maggior privacy delle informazioni e una maggior velocità di convalida delle transazioni grazie al numero di nodi limitato e noto a priori. Infine, le blockchain consortium vengono implementate in situazioni in cui si desidera mantenere il controllo sulle figure che partecipano al mining e, al tempo stesso, rendere accessibile a chiunque la lettura del registro distribuito.

1.3 Mining e algoritmi di consenso

Quando un nodo invia una certa quantità di criptovaluta o di dati a un altro nodo dello stesso network, la transazione risultante non viene subito inserita nel ledger ma subisce una serie di controlli per verificare l'autenticità dei dati, l'identità del mittente e la correttezza della transazione. Il mining è proprio il processo che consente a specifici nodi nella rete, detti *miners*, di convalidare le transazioni, aggregarle in blocchi e aggiungerle al registro in modo tale da raggiungere il consenso distribuito del network; i full nodes e i miners stessi controllano la validità di transazioni e blocchi generati per cui, se ad esempio un miner dovesse produrre un blocco non valido, questo verrebbe rifiutato da tutti gli altri nodi partecipanti e non verrebbe aggiunto alla catena. Gli algoritmi più utilizzati per il raggiungimento del consenso sono due: la Proof of Work (PoW) e la Proof of Stake (PoS).
La Proof of Work, largamente utilizzata in Bitcoin, consiste nella risoluzione di una sfida crittografica tramite hash, ossia una funzione che converte dei dati arbitrari in un numero casuale di lunghezza fissa; la procedura di convalida dei blocchi in un sistema Proof of Work è la seguente:

1. ciascun miner sceglie le transazioni da inserire in un blocco, dando solitamente priorità alle transazioni che presentano spese di commissione maggiori;

2. ogni miner inizia a calcolare la funzione hash dell'intestazione del blocco scelto e il risultato deve soddisfare determinati parametri, ad esempio deve iniziare con un numero di zeri ben definito; la funzione hash usata in Bitcoin è SHA256 mentre in Ethereum è l'algoritmo Ethash;

3. se la soluzione della funzione hash non è valida, il miner va a cambiare il valore del *nonce* (contenuto, come è già stato detto, nell'header della transazione) e ricalcola l'hash;

4. se la soluzione dell'hash soddisfa i parametri stabiliti, il miner trasmette il blocco a tutti i nodi della rete, i quali ne vanno poi a verificare l'effettiva validità;

5. se il nuovo blocco viene ritenuto valido allora viene aggiunto alla catena.

La risoluzione del problema crittografico mediante hashing richiede ai miners un gran numero di tentativi e soprattutto un'elevata potenza computazionale, che si traduce in un consumo energetico non trascurabile. Per incentivare i miners a svolgere questa prova del lavoro sono previste delle ricompense per ogni nuovo blocco minato, oltre alle commissioni delle transazioni incluse nel blocco (*fee*); l'attività di mining dunque, oltre ad essere uno strumento che garantisce fiducia e sicurezza nella blockchain, serve anche a immettere in rete nuove criptovalute, che vengono create con ogni nuovo blocco. Il concatenamento dei blocchi rende impossibile modificare le transazioni incluse in un blocco senza prima modificare tutti i blocchi successivi; di conseguenza, il costo necessario per modificare un blocco aumenta ogni volta che viene aggiunto un nuovo blocco alla catena [8].
L'algoritmo Proof of Work, oltre ad essere dispendioso in termini di potenza di calcolo ed energia, rende la blockchain che lo implementa difficile da scalare a causa del basso throughput, della lentezza delle transazioni e delle commissioni elevate e vulnerabile all'attacco del 51% (majority attack), secondo cui un attaccante che

detiene il 51% della potenza di calcolo del sistema può generare e approvare più velocemente i suoi blocchi rispetto a tutti gli altri miners, prendendo così il controllo sulla validazione delle transazioni.
Un protocollo di consenso più efficiente che risolve i problemi introdotti dal PoW è la Proof of Stake: i partecipanti al processo di verifica, detti validatori, vengono selezionati in base allo stake, ossia alla quantità di criptovalute che possiedono, e ricevono una ricompensa per ogni blocco validato (esattamente come nel Proof of Work); dunque, maggiore è lo stake posseduto da un partecipante della rete e maggiore è la probabilità che questo sia scelto per validare i blocchi di transazioni. Ad oggi, grazie ai numerosi vantaggi apportati in termini di scalabilità, di prestazioni e di costo, sono sempre di più le reti blockchain che stanno cercando di rimpiazzare l'algoritmo Proof of Work a favore del Proof of Stake; tra queste, in particolare, occorre citare la rete Ethereum che, grazie all'aggiornamento Ethereum 2.0 partito nel dicembre 2020 e ancora in fase di test [9], introduce una Beacon Chain che implementa l'algoritmo Proof of Stake in modo da avere come risultato finale una piattaforma più sostenibile, sicura e scalabile.

1.4 Vantaggi e utilizzi della blockchain nell'Industria 4.0

Si è ampliamente parlato di come la blockchain rappresenti una nuova tecnologia che sta prendendo sempre più piede nell'ambito dell'Industria 4.0 in quanto introduce numerosi vantaggi, primo tra tutti quello di creare un ambiente distribuito che non necessita la presenza di un'autorità centrale o di intermediari. Oltre alla garanzia di security e privacy durante lo scambio e nella memorizzazione di dati tra dispositivi IIoT, l'applicazione della blockchain nel settore industriale e manifatturiero presenta altri vantaggi e funzionalità, ad esempio quelli derivanti dall'uso di smart contract: ogni azienda detiene dei registri in cui viene annotato il capitale corrente della stessa e può risultare complicato aggiornare continuamente questo elenco ad ogni passaggio di proprietà da un'impresa ad un'altra; un metodo flessibile e automatico con cui ciò può avvenire è per mezzo degli smart contract, un programma informatico, salvato all'interno della blockchain, che definisce delle regole e che è in grado di eseguire automaticamente delle azioni specifiche quando vengono soddisfatte le condizioni del contratto; in sintesi, si può sfruttare lo smart contract per trasferire e controllare le attività e le valute scambiate tra due enti senza l'intervento di terze parti e ulteriori verifiche [10].
Centrale il ruolo della blockchain anche per quanto riguarda la tracciabilità dei prodotti: l'interesse di un'azienda è rivolta soprattutto al monitoraggio di tutto il ciclo di produzione, a partire dal recapito delle materie prime per l'assemblaggio fino alla vendita del prodotto finale al cliente; la blockchain può in tal senso essere utile per organizzare e gestire in modo sicuro la supply chain, registrando la lista di tutti i prodotti, macchinari e costi, monitorando le procedure di produzione e tracciando tutte le modifiche dei processi produttivi avvenuti nel tempo; il risultato è quello di avere un sistema che garantisce l'affidabilità dei prodotti e del ciclo produttivo, realizzabile grazie alla proprietà di integrità dei dati della blockchain: le transazioni sono raggruppate in blocchi e questi ultimi vengono aggiunti alla blockchain in modo sequenziale; i blocchi sono collegati tra loro mediante una funzione crittografica di hash che assicura la sequenzialità con il blocco precedente, per cui, una volta formatasi la catena di blocchi, risulta impossibile modificare o cancellare le transazioni

in un blocco senza che l'hash del blocco successivo cambi. Infine, l'adozione della blockchain nell'ambito dell'Industria 4.0 favorisce il pagamento digitale e automatico peer-to-peer con una riduzione del fee e dei tempi di transazione rispetto al classico pagamento elettronico con carta di credito.

2 La blockchain Ethereum

Ethereum è una piattaforma blockchain pubblica distribuita, usata come rete di pagamento e di archiviazione dati decentralizzata ma soprattutto per creare e attivare contratti intelligenti senza tempi di inattività, frodi e controlli da parte di terze parti; l'idea alla base di Ethereum è infatti quella di costruire una blockchain in grado di eseguire applicazioni e programmi generici (Dapps).
La valuta nativa di Ethereum, l'Ether (ETH), viene generata dalla piattaforma stessa come ricompensa per i miner, grazie all'algoritmo di Proof of Work (che, come è stato detto, si sta cercando di rimuovere a favore del Proof of Stake), e può essere scambiata tra i vari nodi della rete attraverso le transazioni.
In questo capitolo viene presentato questo modello di piattaforma blockchain, ponendo in particolare l'attenzione verso le funzioni crittografiche alla base della creazione delle transazioni Ethereum, la cui implementazione verrà illustrata nei successivi capitoli.

2.1 Generazione di chiavi

In Ethereum esistono due tipi di account: gli account EOA (externally owned accounts) e gli account associati a smart contract; i primi sono account di proprietà esterna, controllati da una chiave privata con la quale è possibile accedere al proprio conto e inviare e ricevere transazioni, mentre i secondi sono controllati da un codice di contratto interno che si attiva quando l'indirizzo ad esso collegato riceve una transazione e che agisce eseguendo quanto scritto nel codice [11].
Per quanto riguarda gli account EOA, Ethereum utilizza ECC (Elliptic Curve Cryptography) per la crittografia a chiave pubblica, ossia per la generazione di una chiave pubblica a partire da una certa chiave privata, e l'algoritmo ECDSA (Elliptic Curve Digital Signature Algorithm) per la firma digitale delle transazioni; l'obiettivo è quello di creare una firma digitale che permetta la verifica di autenticità dell'utente che invia la transazione, garantendo anche l'integrità dei dati e il non ripudio da parte del mittente.
Una transazione Ethereum, per poter essere inclusa nella blockchain, necessita di una firma digitale valida; la chiave privata viene usata per creare la firma necessaria per spendere o inviare fondi o informazioni dimostrando la proprietà dell'account da cui è partita la transazione; un utente che intende trasmettere o condividere dei dati deve infatti firmare la transazione tramite la propria chiave privata, che non deve essere in alcun modo trasmessa o archiviata sulla rete: chiunque in possesso della chiave privata di un certo wallet sarà in grado di avere il controllo di quel corrispondente account, potendo accedere alle risorse (e quindi agli Ether) di quel portafoglio e potendo inviare transazioni ad altri account.
In Ethereum la chiave privata viene generata mediante un generatore di numeri pseudo-casuali come CSPRNG e quello che produce è una stringa di lettere e numeri che identifica in maniera univoca un utente e che deve essere tenuta segreta dal proprietario; questa chiave è lunga 256 bit e viene mostrata in formato esadecimale con 64 caratteri, dal momento che un carattere esadecimale è rappresentabile come 4 bit, per cui il numero delle possibili combinazioni è 2^{256} , un numero sufficientemente grande tale da rendere la chiave sicura.
A questo punto, seguendo la teoria della crittografia asimmetrica, viene generata la

chiave pubblica a partire da quella privata, utilizzando la curva ellittica secp256k1, mostrata in Figura 3, la cui equazione associata ha la forma $y^2 = x^3 + 7$. Si va a

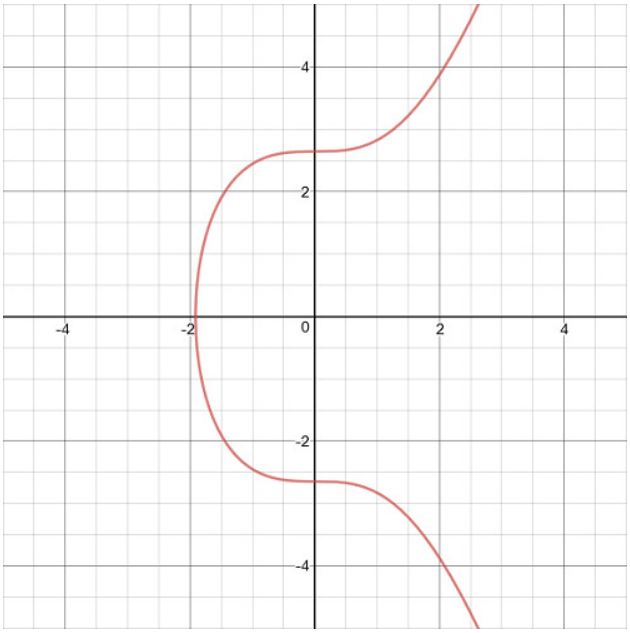


Figura 3: *Grafico della curva secp256k1, usata sia in Ethereum che in Bitcoin*

scegliere un punto casuale G(x,y) sulla curva chiamato punto generatore e, indicando con d_a la chiave privata, si ricava la chiave pubblica:
$Q_a = G(x, y) * d_a$.

Si osserva che la chiave ottenuta è un punto sulla curva ellittica [12] di coordinate (x,y) ed è lunga 512 bit, quindi 128 caratteri in formato esadecimale. Noti Q_a e G e data l'operazione di moltiplicazione della curva ellittica, risulta impossibile risalire al valore di d_a, in quanto ciò richiederebbe il calcolo della funzione inversa e dunque la risoluzione del problema del logaritmo discreto (ECC non definisce le operazioni di divisione e sottrazione); in altre parole, quella della curva ellittica è una funzione trapdoor in quanto è computazionalmente semplice calcolare la chiave pubblica a partire da quella privata mentre è impossibile ricavare la chiave privata da quella pubblica, sebbene le due chiavi siano strettamente legate; l'unidirezionalità di questo processo è alla base della sicurezza di questo sistema a chiave pubblica.
L'indirizzo pubblico del wallet Ethereum, una stringa di 40 caratteri esadecimali (160 bit), deriva dalla funzione hash della chiave pubblica; una funzione crittografica di hash è una funzione unidirezionale che prende in input dati di dimensioni arbitrarie e produce in output una stringa di bit di dimensioni fissa. Essendo una funzione "one-way", se si conosce solo l'hash in uscita non è possibile dal punto di vista computazionale risalire ai dati di ingresso, a meno di condurre un attacco a forza bruta, calcolando l'hash di ogni singolo input e verificando la corrispondenza con il risultato in esame.
La funzione hash usata da Ethereum è Keccak-256, una variante di SHA-3, che riceve in ingresso la chiave pubblica di 512 bit e genera in output una stringa di 256 bit; di questi si vanno a considerare gli ultimi 160 bit, ossia gli ultimi 40 caratteri, per rappresentare l'indirizzo associato a quella specifica chiave (vedi Figura 4). Il troncamento dell'hash che si ottiene dalla chiave pubblica è finalizzato al consegui-

mento di indirizzi più corti possibili. Una volta prelevati questi 40 caratteri dalla chiave pubblica si va ad aggiungere il prefisso 0x, dal momento in cui gli indirizzi Ethereum sono in formato esadecimale; questi saranno dunque composti solo da numeri dallo 0 al 9 e dalle lettere dalla A alla F, senza distinzione tra maiuscole e minuscole (la presenza di maiuscole in un indirizzo ethereum viene usata unicamente come checksum, cioè per provare che la digitazione di un indirizzo sia avvenuta in maniera corretta) [13].

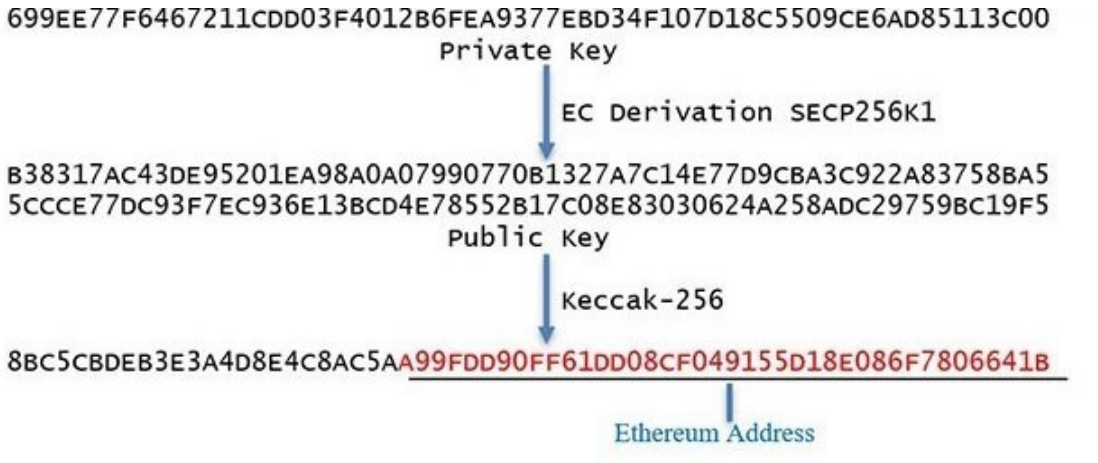


Figura 4: *Rappresentazione del processo di generazione di un indirizzo Ethereum a partire dalla chiave privata*

Con questa procedura vengono creati tutti gli indirizzi della piattaforma Ethereum, ciascuno dei quali è associato all'identità di un unico utente della rete grazie al legame con la chiave pubblica, derivata a sua volta dalla corrispondente chiave privata. Il vantaggio di questo tipo di crittografia risiede nella generazione di chiavi più corte rispetto ad altri algoritmi di crittografia asimmetrica come RSA (una chiave ECC a 256 bit equivale a una chiave RSA a 3072 bit) e nel minor consumo di potenza di calcolo e di utilizzo della memoria [14].

2.2 Transazioni Ethereum

Una transazione è un pacchetto di dati firmati che permette uno scambio di token, informazioni e messaggi tra due account appartenenti alla stessa rete; come si può osservare dalla figura 5, le transazioni sono strutturate nel seguente modo [15]:

- *Nonce*: rappresenta il numero progressivo delle transazioni passate inviate dall'account; il nonce, il cui valore aumenta di uno ogni volta che l'account invia una transazione, ha lo scopo di evitare il cosiddetto replay attack, ossia la duplicazione di una stessa transazione.
- *Gas price*: prezzo (in wei) che si paga per unità di gas.
- *Gas limit*: quantità massima di gas che può essere consumata in una transazione; su Ethereum si consiglia di non superare il limite di 21 000 gas.

- L'indirizzo di destinazione, che può essere relativo all'account EOA o all'account del contratto; entrambi i tipi di account (così come quello del mittente) sono identificati da un indirizzo di 20 byte.
- Valore: quantità di Ether da inviare al destinatario (facoltativo).
- Dati: payload, che comprende smart contract e dati (facoltativo).
- I tre valori (r,s,v) della firma digitale ECDSA, che identifica il mittente.

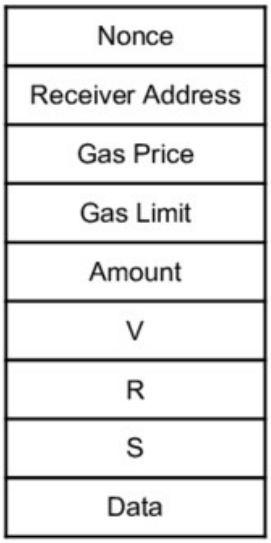


Figura 5: *Struttura di una transazione Ethereum*

Il Gas è il prezzo che i mittenti delle transazioni devono pagare per ogni operazione effettuata sulla rete Ethereum; ciò è dovuto al fatto che ogni transazione, sia essa un semplice trasferimento di dati, di Ether o una funzione di attivazione di uno smart contract, comporta un costo proporzionale alla complessità computazionale; questa potenza di calcolo richiesta per l'esecuzione di operazioni viene pagata in Ether o in wei (1 Ether=10^{18} wei). Gli utenti della rete possono regolare il prezzo del gas (quindi il parametro Gas price) nelle transazioni che intendono inviare per ottenere una conferma più rapida: infatti maggiore è il prezzo del gas, maggiore è la probabilità che la transazione venga confermata, mentre le transazioni con un prezzo ridotto hanno priorità inferiore rispetto alle altre.
In figura 6 sono rappresentate le varie fasi che deve superare una transazione Ethereum per poter essere immessa nel network: nel caso in esempio si è fatto uso delle librerie web3.js per interagire con un nodo Ethereum e scrivere una transazione contenente tutti i parametri sopra descritti; si osserva come tutti i valori dei suddetti parametri vengono prima codificati mediante RLP (Recursive Length Prefix), quindi serializzati secondo l'ordine all'interno della transazione, dopodichè alla stringa risultante viene applicata una funzione hash keccak per ottenere l'ID della transazione; solo a questo punto la transazione viene firmata con l'opportuna chiave privata dell'utente.

Figura 6: *Descrizione del processo di serializzazione e firma di una transazione Ethereum*

2.3 La firma ECDSA

Come è stato già anticipato, ECDSA viene usato anche per l'apposizione e la verifica di firme digitali. Quando un utente intende inviare un payload ad un altro nodo della rete, è necessario che il mittente firmi la relativa transazione o messaggio tramite la propria chiave privata e invii i suddetti dati al destinatario, il quale verificherà la veridicità della firma con la chiave pubblica del mittente. Indicando con d_a e Q_a rispettivamente la chiave privata e pubblica del mittente, la procedura per la firma digitale è la seguente [16]:

1. L'utente che intende inviare una transazione effettua per prima cosa una codifica RLP dei dati della transazione, dopodiché calcola l'hash del pacchetto z=hash(m) tramite la funzione keccak-256 in modo da adattare la lunghezza del messaggio alle proprietà della curva ellittica.

2. L'algoritmo genera un numero grande k intero nell'insieme [0,. . . ,p-1], dove p è un numero primo che rappresenta l'ordine della curva ellittica; il numero k rappresenta una chiave privata temporanea a cui è associata una chiave pubblica E=(x_1,y_1)=k*G , con G punto generatore.

3. La coordinata x_1 della chiave E rappresenta la prima parte della firma digitale e la si indica con $r \equiv x_1$ modulo p; se r=0, allora si genera un altro valore di k e si riesegue la procedura.

4. A partire da r si ricava la seconda parte della firma: $s \equiv k^{-1}(z + r * d_a)$ modulo p; se s=0 si risceglie un nuovo valore di k e si rieseguono i calcoli.

Alla fine si ottiene la coppia (r,s) che rappresenta i valori della firma digitale, che viene inviata assieme al payload al destinatario, il quale procede nel seguente modo per la verifica della firma:

1. Esattamente come il mittente, calcola l'hash keccak-256 della transazione z=hash(m).
2. Va a calcolare $w \equiv s^{-1}$ modulo p.
3. Calcola $u_1 \equiv (w * z)$ modulo p e $u_2 \equiv (w * r)$ modulo p.
4. Si ricava il punto $(x_1 y_1) = u_1 * G + u_2 * Q_a$.
5. Se $r \equiv x_1$ modulo p, allora la firma può essere ritenuta valida.

In una transazione, insieme alla coppia (r,s) è presente un terzo valore indicato con v che serve per il recupero della chiave pubblica del mittente; infatti nei campi di una transazione è indicato solo l'indirizzo del destinatario e non quello del mittente, per cui è necessario il recupero della chiave pubblica dalla cui conoscenza si può poi risalire facilmente all'indirizzo.
Oltre a dare una garanzia dell'autenticità del mittente, la firma digitale assicura anche l'integrità dei dati, in quanto, una volta firmata una transazione, se si andasse a modificare il payload della transazione cambierebbe anche l'hash rispetto all'originale e dunque la firma non sarebbe più valida; la firma digitale garantisce infine il non ripudio dato che il mittente non può negare di aver generato la transazione o negare il contenuto della stessa.

2.4 La codifica RLP

Si è visto nei precedenti sottocapitoli come l'operazione di serializzazione sia centrale per la corretta generazione di una transazione Ethereum; in particolare il processo di serializzazione dei dati viene applicato due volte: la prima volta quando viene creato il payload contenente i dati grezzi della transazione, di cui poi si va a calcolare l'hash; la seconda volta quando, dopo che si è firmato l'hash del payload, si vanno a inserire i parametri di firma nella transazione.
La tecnica di serializzazione implementata dalla rete Ethereum è la RLP, acronimo di Recursive Length Prefix; lo scopo della RLP è quello di serializzare e codificare array di dati binari in modo che tutti i campi che compongono la transazione risultino concatenati in un'unica stringa in formato esadecimale. La stringa in output dalla funzione RLP, oltre a contenere i diversi parametri della transazione (ovviamente in formato esadecimale), presenta alcune cifre proprio davanti ai dati che si intende codificare; queste due cifre in esadecimale servono a dare indicazioni circa il tipo e la dimensione del dato che si intende serializzare.
La funzione di codifica RLP accetta in input un elemento che può essere un byte, una stringa o una lista di elementi [17] e restituisce un output che dipende, come già detto, dal tipo di dato e dalla dimensione in byte:

- se si vuole serializzare un byte con valore compreso tra 0 e 127, allora la codifica RLP sarà uguale al valore di quel byte in formato esadecimale;

- se si vuole serializzare una stringa in formato esadecimale, allora la codifica RLP sarà pari a 0x80 a cui dovrà essere sommata la lunghezza in byte della stringa seguita dalla stringa stessa. Ad esempio, se si vuole serializzare il parametro r della firma digitale "0x39707d278d9462bbe1c4d8fc3b145f7f9e8a05bdd9e139741fd47c5eaf2dde", avente una lunghezza di 32 byte, allora il codificatore RLP restituisce "0xa039707d278d94620bbe1c4d8fc3b145f7f9e8a05bdd9e0139741fd47c5eaf2dde", ossia la stessa stringa di partenza con un numero identificativo davanti, 0xa0; questa cifra, che in decimale corrisponde a 160, deriva dalla somma tra 128 (0x80 in esadecimale) e 32 (la dimensione in byte della stringa);

- se si vuole serializzare una lista di elementi avente una lunghezza superiore ai 55 byte, allora la codifica RLP sarà pari a 0xf7 più la dimensione in byte della lista seguita dalla concatenazione delle codifiche RLP degli elementi che compongono tale lista; questo è ciò che avviene durante la serializzazione del payload, considerato come una lista di tante stringhe; la prima cifra del payload serializzato corrisponde infatti alla dimensione totale della transazione, mentre tutti i parametri interni del payload vengono serializzati come stringhe e concatenati tra loro.

3 Analisi della piattaforma embedded e della libreria utilizzata

Nel seguente capitolo viene fornita una descrizione della piattaforma hardware utilizzata per l'implementazione del nodo; dopo una breve quadro sulle caratteristiche tecniche della scheda, si passa a esaminare la parte software, con un particolare riguardo alle librerie ST impiegate per la realizzazione dell'algoritmo; infine vengono riportati i parametri della curva ellittica secp256k1, implementata dal protocollo Ethereum.

3.1 Descrizione dell'hardware e dei software utilizzati

La piattaforma che è stata utilizzata ai fini di questo lavoro di tesi è la STM32 Nucleo-144 della STMicroelectronics (Figura 7); essa è basata su un microcontrollore STM32F439ZI con un core ARM Cortex-M4 a 32 bit che opera a una frequenza di 180MHz, comprende 144 pin multifunzione ed un debugger ST-LINK che permette la programmazione della scheda tramite USB. Il dispositivo è dotato di una memoria SRAM da 256 KB e di una memoria Flash da 2 MB, all'interno della quale viene memorizzato il codice eseguibile. La scheda offre numerose funzionalità tra cui la possibilità di implementare algoritmi crittografici come AES, DES, SHA-1 e SHA-2 e relativa generazione di chiavi di cifratura e firme digitali; tra i vari algoritmi spicca la crittografia a chiave pubblica basata sulle curve ellittiche ECC (e relativo algoritmo ECDSA per la firma digitale), che sono alla base della generazione di chiavi e della firma digitale per le transazioni Ethereum [18].

Il device è dotato di diverse periferiche, ad esempio tre convertitori analogico/digitali a 12 bit, due convertitori D/A e un generatore di numeri casuali (RNG); per la trasmissione dei dati dalla scheda viene utilizzato il protocollo di comunicazione seriale UART, che converte i dati paralleli in forma seriale, li trasmette in seriale al dispositivo UART ricevente per poi riconvertirli, una volta ricevuti, in parallelo; altri protocolli di comunicazione supportati dal microcontrollore sono USART, I^2C, SPI, CAN e SAI.

Dal punto di vista software, invece, per consentire la scrittura e l'esecuzione di un codice sulla piattaforma embedded sono stati utilizzati due programmi: STM32CubeMX e Atollic TrueSTUDIO. STM32CubeMX è uno strumento grafico che garantisce una configurazione molto semplice dei microcontrollori della serie STM32, nonché la generazione di un codice C di inizializzazione che tiene conto dell'attivazione delle periferiche e delle porte desiderate; dopo aver selezionato il corretto modello del microcontrollore e della scheda di sviluppo si aprirà la schermata del progetto con una rappresentazione virtuale del microcontrollore, da cui è possibile selezionare, attivare o disattivare periferiche e funzionalità. Atollic TrueSTUDIO è invece un IDE di sviluppo e di debug basato su Eclipse con un compilatore C/C++ ottimizzato che garantisce efficienza nella progettazione di sistemi embedded. In alternativa ad Atollic è possibile usare altri tool di sviluppo come Keil MDK-ARM e STM32CubeIDE. Per l'implementazione del progetto in esame sono state inizializzate e impiegate le seguenti periferiche:

- USART3: come è già stato anticipato, il modulo USART permette la comunicazione seriale sincrona o asincrona tra diversi dispositivi. Nel caso in esame

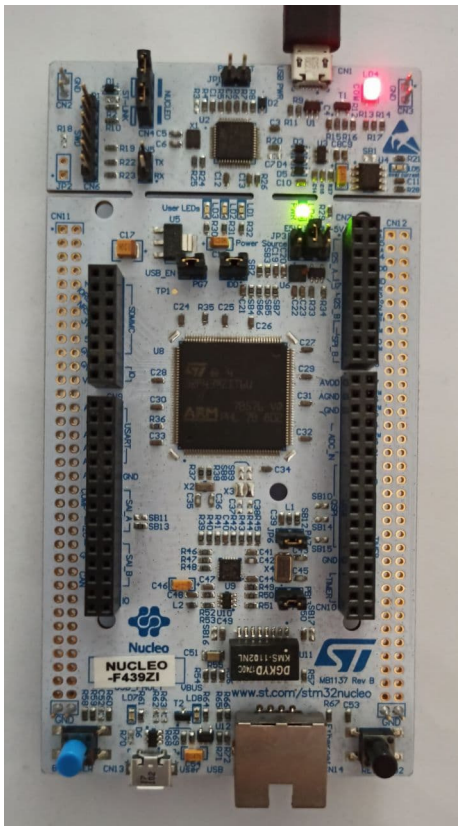


Figura 7: *Scheda STM32 Nucleo-144*

questa periferica, impostata per semplicità in modalità asincrona, consente la trasmissione di informazioni tra il microcontrollore e il calcolatore in modo da poter visualizzare e rielaborare in Matlab i dati di interesse;

- CRC (Cyclic Redundancy Check): è una tecnica utilizzata per il controllo di integrità dei dati e rilevazione degli errori in fase di trasmissione e archiviazione; per poter usufruire correttamente delle librerie crittografiche ST è richiesta l'attivazione del clock della periferica CRC;

- RNG (Random Number Generator): la maggior parte dei microcontrollori STM32 è dotato di un True RNG, un generatore che produce una casualità di numeri non deterministica dipendente da sorgenti fisiche imprevedibili e incontrollabili, come può essere ad esempio un rumore termico o meccanico, per cui risulta impossibile risalire all'algoritmo che l'ha generato. Il random number generator in esame è un circuito analogico costituito da diversi oscillatori in grado di generare un rumore continuo, che viene elaborato per produrre un numero casuale; il circuito comprende inoltre un clock a frequenza costante che ha lo scopo di sincronizzare la generazione dei numeri. In questo progetto il TRNG viene utilizzato per lo svolgimento di alcuni test, ad esempio per la generazione di un numero random lungo 32 byte da utilizzare come chiave privata o per la generazione di un messaggio random da firmare; nei capitoli successivi si approfondirà nel dettaglio questo aspetto.

L'attivazione di questi moduli avviene tramite il tool STM32CubeMX: dopo aver configurato correttamente la RCC, la periferica interna che gestisce i diversi oscillatori e le frequenze di clock, si passa all'inizializzazione di tutte le periferiche sopra descritte; il codice che viene conseguentemente generato tiene conto di tutti i parametri attivati o modificati con il software grafico e include, a seconda della preferenza impostata, le librerie HAL o le LL.
Il progetto viene aperto con Atollic TrueSTUDIO ed esso contiene, oltre a quanto sopra descritto, anche i file header, le librerie CMSIS (caratteristiche dei microprocessori ARM) e tre diverse librerie esterne da cui vengono prelevate le funzioni crittografiche per la creazione di transazioni Ethereum; nel dettaglio, queste librerie sono:

- X-CUBE-CRYPTOLIB, la libreria crittografica fornita dalla ST, che contiene funzioni firmware per i principali microcontrollori della serie STM32; essa include tutti i principali algoritmi di sicurezza per hashing, crittografia, autenticazione e firma digitale come RSA, AES, MAC, diverse funzioni della famiglia SHA e delle curve ellittiche; di questa libreria per il progetto in esame sono state utilizzate le funzioni per la generazione di numeri pseudorandom (PRNG), le funzioni per la generazione di un digest tramite hash SHA-256 e le funzioni per l'implementazione dell'algoritmo ECDSA per la generazione delle chiavi, la firma e verifica delle transazioni Ethereum (vedi Cap. 2);

- Ethereum-RLP, una libreria messa a disposizione nella repository di GitHub[1] da cui sono state prese le funzioni per la serializzazione della transazione Ethereum;

- Ethereum wallet, libreria anch'essa estratta da GitHub[2]; da questa è stata presa la funzione per l'hashing *keccak256*, utilizzata sia per la generazione dell'indirizzo Ethereum dalla chiave pubblica sia per l'hashing della transazione dopo la serializzazione del payload.

3.2 Implementazione della curva ellittica

Come è già stato ampiamente detto, nella blockchain Ethereum, così come in Bitcoin, per la generazione della chiave pubblica, per la firma digitale e per la verifica della stessa viene adottato l'algoritmo ECDSA, basato sulla crittografia a curva ellittica; tra le numerose curve appartenenti alla famiglia delle ECC, quella che viene usata dalle due principali blockchain è la *secp256k1*, definita dallo Standards for Efficient Cryptography Group (SECG) come una particolare curva di Koblitz strutturata in maniera non casuale in modo da garantire, a differenza delle curve NIST come la *secp256r1*, un calcolo particolarmente efficiente, risultando essere fino al 30% più veloce rispetto alle altre curve; inoltre, grazie proprio alla corretta scelta dei parametri a e b, questo tipo di curva non risulta essere compromessa dalla presenza di backdoor sulla curva [22] [23].
La curva *secp256k1* è definita sul campo finito $Z_{2^{256}-2^{32}-2^9-2^8-2^7-2^6-2^4-1}$ e ha equazione

[1] link alla libreria Ethereum-RLP: `https://github.com/KingHodor/Ethereum-RLP`

[2] link alla libreria Ethereum wallet: `https://github.com/firefly/wallet/tree/master/source/libs/ethers/src`

$$y^2 = x^3 + 7$$

In seguito sono riportati tutti i parametri della curva ellittica espressi in forma esadecimale, necessari per la corretta implementazione del codice in C:

- Modulo del campo p= FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F.
- Parametro a= 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000.
- Parametro b= 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007.
- Punto base o punto generatore G:

  - Coordinata x: G_x= 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798.

  - Coordinata y: G_y= 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8.

- Ordine n del punto generatore G: n = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141.

Tutti questi parametri della curva ellittica secp256k1 vengono implementati per l'esecuzione di tre operazioni: la generazione di una coppia di chiavi, la firma digitale di una transazione mediante chiave privata e la sua verifica con chiave pubblica.

4 Implementazione del codice

Nel seguente capitolo viene spiegato il funzionamento dell'algoritmo, implementato sul microcontrollore e scritto in C, per la generazione di transazioni Ethereum valide. Il nodo che viene creato è un light node e non un nodo completo in quanto la piattaforma embedded non dispone di memoria e capacità computazionali sufficienti per effettuare il mining delle transazioni; dunque, le operazioni che il microcontrollore si limita a effettuare sono quelle più basilari: generazione di una coppia di chiavi e indirizzo associato, generazione di un pacchetto dati, serializzazione dei dati, firma e verifica del pacchetto e infine creazione della transazione vera e propria.
Per l'analisi sono state create 256 transazioni, per la cui generazione sono state eseguite tutte le operazioni sopra elencate e che verranno approfondite nel seguito.

4.1 Generazione della coppia di chiavi e dell'indirizzo associato

I crittosistemi basati su curve ellittiche si avvalgono di una coppia di chiavi, una privata e una pubblica; ogni coppia di chiavi è associata a un particolare insieme di parametri del dominio definito dalle curve ellittiche. La chiave privata è una sequenza di 32 byte ottenuti applicando l'algoritmo secp256k1 per cui i valori numerici devono soddisfare delle particolari specifiche; più nel dettaglio, la chiave privata deve essere un numero compreso tra 1 e n-1, dove n è l'ordine della curva. Nell'implementazione del codice, la chiave privata è stata realizzata come un numero del tutto random generato a partire dal TRNG del microcontrollore.

```
void GenerateRandomPrivKey( int Arr_length, uint8_t* privatekey, int32_t ChiavePriv_length)  {
    uint32_t trng=0;
    uint32_t TrueRandomNum[Arr_length];
    for(int i=0; i< Arr_length; i++) {
        HAL_RNG_GenerateRandomNumber(&hrng, &trng);
        TrueRandomNum[i]=trng;
    }
    STM32_SHA256_HASH_DigestCompute((uint8_t*) TrueRandomNum, Arr_length, (uint8_t*) privatekey, &ChiavePriv_length);
}
```

La funzione *GenerateRandomPrivKey* prende in ingresso la dimensione dell'array in cui viene memorizzato il numero casuale prodotto dal TRNG, il puntatore alla chiave privata e la sua lunghezza; all'interno di questa funzione viene generato un vettore, chiamato *TrueRandomNum*, che restituisce un numero random la cui dimensione dipende dal valore di *Arr_length*. Per fare in modo che la funzione restituisca sempre e comunque una chiave privata di 256 bit, viene applicato un hashing SHA-256 sul numero random casuale e il risultato viene passato al puntatore *privatekey*.
Prima di assegnare il valore in output alla chiave privata è necessario inizializzare le strutture in cui le variabili saranno definite: per prima cosa si vanno a inserire in una struttura, chiamata *EC_st*, tutti i parametri della curva ellittica che sono stati definiti nel precedente capitolo, dopodiché si inizializza la struttura in questione; lo stesso procedimento si applica anche ad un buffer, denominato *Crypto_Buffer*, richiesto dalla libreria X-CUBE per la memorizzazione di tutti quei parametri su cui devono essere eseguite le operazioni crittografiche; infine è necessario inizializzare le strutture che contengono la chiave privata e i due punti di interesse della curva

ellittica: il punto generatore G e la chiave pubblica *PubKey*.

```
ECpoint_stt *G = NULL;          //Object that contains the coordinates of the generator point G
ECpoint_stt *PubKey = NULL;       // Object that contains the coordinates of the public key
ECCprivKey_stt *PrivKey = NULL;      // Private Key structure

ECCinitEC(&EC_st, &Crypto_Buffer);           //initialize EC parameters
ECCinitPoint(&PubKey, &EC_st, &Crypto_Buffer);        //initialize public key PubKey
ECCinitPoint(&G, &EC_st, &Crypto_Buffer);           //initialize generator point G
status=ECCinitPrivKey(&PrivKey, &EC_st, &Crypto_Buffer);         //initialize private key structure
if (status != ECC_SUCCESS) {
    printf("Error message! Something went wrong, check the initialized parameters %lu\n", status);
     return(-1);
}
   //Set the coordinates of the generator point G
 status = ECCsetPointGenerator(G, &EC_st);
 if (status != ECC_SUCCESS) {
     printf("Error! ECCsetPointGenerator returned %lu\n", status);
     return(-1);
 }
```

Come si può osservare, se la variabile *status* assume un valore diverso da quello che stabilisce la correttezza dell'algoritmo, l'esecuzione del codice si interrompe e viene rilasciato un messaggio di errore.
Dopo aver correttamente assegnato il valore prodotto dalla funzione SHA256 alla struttura associata alla chiave privata, si passa alla generazione della chiave pubblica; come è già stato detto nel precedente capitolo, la chiave pubblica altro non è che il prodotto tra la chiave privata e il punto generatore G. Indicando con PrivKey, PubKey e G i puntatori alle strutture che contengono rispettivamente la chiave privata, la chiave pubblica e il punto generatore, si implementa la funzione *ECCscalarMul* che esegue questa moltiplicazione; la funzione richiede in ingresso anche i parametri della curva ellittica e il buffer per la memorizzazione delle operazioni.

```
ECCsetPrivKeyValue(PrivKey, privatekey, priv_size);    //Set the private key object
// Multiply PrivateKey by generator point G
status=ECCscalarMul(G, PrivKey, PubKey, &EC_st, &Crypto_Buffer);
if (status != ECC_SUCCESS) {
   printf("Error! ECCscalarMul returned %lu\n",status);
   return(-1);
 }
ECCvalidatePubKey(PubKey, &EC_st, &Crypto_Buffer);   //Validate Public key
```

La validazione della chiave pubblica avviene mediante la funzione *ECCvalidatePubKey*, che va a verificare le seguenti condizioni [24]:

- che la chiave sia diversa dal punto infinito O;
- che le coordinate del punto associato alla chiave appartengano alla curva ellittica secp256k1;
- che il prodotto tra l'ordine della curva e la chiave pubblica restituisca il punto infinito O.

Una volta verificata la validità si passa alla generazione dell'indirizzo Ethereum associato alla chiave pubblica; c'è da precisare che le funzioni della libreria X-CUBE dispongono i byte nell'ordine little endian per cui, per poter ricavare il corretto indirizzo, è necessario invertire l'ordine dei byte della chiave pubblica, operazione che

può essere eseguita mediante la funzione *ECCgetPointCoordinate* che restituisce la coordinata x o y della chiave nell'ordine big endian.
La funzione hash utilizzata per generare l'indirizzo Ethereum è keccak256, implementata nella libreria di Ethereum wallet; a tal proposito, prima di procedere con l'operazione è importante che entrambe le coordinate della chiave pubblica vengano concatenate e memorizzate in un unico array di 64 byte, come richiesto dalla funzione di hashing, che deve ricevere in input anche la dimensione della chiave e un array di 32 byte in cui verrà poi inserito il digest prodotto.

```
uint8_t pubKeyX[32];                    // Buffer to keep the returned public key x coordinate
uint8_t pubKeyY[32];                   // Buffer to keep the returned public key y coordinate
int32_t Xsize;                      //size of the public key x coordinate
int32_t Ysize;                   //size of the public key y coordinate
uint8_t PublicKey[64];             //Array that contains public key
ECCgetPointCoordinate(PubKey, E_ECC_POINT_COORDINATE_X, pubKeyX, &Xsize);   //extract the x coordinate
ECCgetPointCoordinate(PubKey, E_ECC_POINT_COORDINATE_Y, pubKeyY, &Ysize);   //extract the y coordinate
for(int i=0; i<32; i++) {
   PublicKey[i]=pubKeyX[i];
   PublicKey[32+i]=pubKeyY[i];
 }
ethers_keccak256(PublicKey, sizeof(PublicKey) , hashed);     //hashing of public key (32 byte)
memcpy(address, &hashed[12], 20);     //Copy on address the last 20 byte of the generated digest
```

Quello che si ottiene in output da questa operazione è una stringa di 64 caratteri; dato che l'indirizzo è formato soltanto dagli ultimi 40 caratteri di questa stringa, con la funzione C *memcpy* vengono copiati gli ultimi 20 byte del digest nel vettore *address*, che costituisce l'indirizzo associato alla chiave pubblica precedentemente creata.

4.2 Generazione di una transazione grezza e della sua firma

Una volta prodotta una coppia di chiavi e il relativo indirizzo associato, è possibile procedere con la creazione del payload della transazione, la cui struttura è stata illustrata nel capitolo 2; per l'implementazione della codifica RLP, necessaria per la serializzazione dei parametri di cui la transazione è formata, è stata utilizzata la libreria Ethereum RLP di Github.
Prima di procedere con l'algoritmo di serializzazione è opportuno assegnare dei valori ai vari campi della transazione:

- Il nonce è stato impostato uguale a 1 e, ad ogni nuova transazione proveniente dallo stesso indirizzo, viene incrementato di 1; essendo state effettuate 256 prove, il nonce assumerà valori da 1 a 256;
- il gas price e il gas limit sono stati scelti in accordo con i parametri della rete Ethereum: il gas price a 1000 Gwei e il gas limit a 21544 unità di Gas;
- la quantità di Ether da inviare è stata impostata a 0, mentre come dato è stata scelta, per semplicità di notazione in esadecimale, la stringa "deadbeef";
- come indirizzo del destinatario è stato scelto un indirizzo associato a un nodo reale della rete: "0x0B53152B4bd2B89CAb631ca40F251E91760A73c7";
- I parametri della firma digitale r e s devono essere nulli, mentre v deve essere impostato uguale al chainID; volendo registrare la transazione su una rete Hyperledger Besu, il valore del chainID è stato settato a 1981.[25].

Tutti questi parametri, oltre a un array, denominato *rawtx*, in cui viene memorizzata la stringa serializzata, vengono forniti in ingresso alla funzione *assembleTx*; è importante che tutti i dati che compongono la transazione vengano dichiarati come tipi di dato char, come richiesto dalla funzione in questione. L'operazione successiva prevede il calcolo dell'hash di questo payload serializzato, sempre mediante keccak256; dato che quest'ultima richiede in ingresso un array di tipo uint8, mentre l'output della funzione di serializzazione è una stringa di caratteri di tipo char, è necessario effettuare una conversione, operazione che viene fatta mediante la funzione della libreria RLP Ethereum *hex2byte_arr*.

```
//Create the payload to hash
lgt=assembleTx(rawTx, tx, signature, raw_tx_bytes, nonce_char, gas_price, gas_limit, to, value, data, r, s, chainID);
uint8_t Raw_conv[256];
hex2byte_arr(rawTx, strlen(rawTx), Raw_conv, 128);    //Convert payload from char to uint8

ethers_keccak256(Raw_conv, (lgt-1)/2 , hash_transaction);         //Hash of the transaction with keccak256
```

Una volta convertiti i caratteri della stringa in byte e dimensionato opportunamente l'array in output, si procede con l'implementazione della funzione di hashing; essa restituisce in output un digest di 256 bit, che rappresenta proprio il pacchetto dati che si deve andare a firmare.
Per l'operazione di firma digitale, si deve per prima cosa inizializzare la struttura *sign* in cui verranno memorizzati i parametri r e s in output; si procede quindi creando una struttura (nel codice chiamata *signCtx*) in cui inserire tutti i parametri che devono essere utilizzati per l'implementazione della firma, come la chiave privata, i parametri della curva ellittica e un numero casuale k generato da una funzione PRNG di X-CUBE, opportunamente inizializzata. La firma dell'hash del payload, effettuata mediante la funzione *ECDSAsign*, restituisce un puntatore ai due parametri della firma r e s, disposte con un ordine dei byte little endian; anche qui, come per la chiave pubblica, la conversione in big endian viene attuata tramite la funzione *ECDSAgetSignature* che restituisce un array con la coordinata r o s della firma.

```
status = STM32_Init_RNG_for_Sign(&RNGstate, Num_Prova);    //initialize PRNG

ECDSAinitSign(&sign, &EC_st, &Crypto_Buffer);   // initialize sign structure

/* Prepare the structure for the ECDSA signature sign */
signCtx.pmEC = &EC_st;
signCtx.pmPrivKey = PrivKey;
signCtx.pmRNG = &RNGstate;

/*Sign the hash of the payload */
status = ECDSAsign(hash_transaction, CRL_SHA256_SIZE , sign, &signCtx, &Crypto_Buffer);

status=ECDSAgetSignature(sign, E_ECDSA_SIGNATURE_R_VALUE, signR, &signRsize);   //store r in signR
status=ECDSAgetSignature(sign, E_ECDSA_SIGNATURE_S_VALUE, signS, &signSsize);    //store s in signS
```

4.3 Processo di verifica della firma e creazione della transazione Ethereum

Una volta che è stata eseguita la firma della transazione Ethereum è necessario verificarne la validità, ossia occorre riscontrare che la firma sia stata effettivamente generata dalla corretta chiave privata, associata alla corrispondente chiave pubblica

con cui si effettua la verifica.
La verifica della firma viene implementata dalla funzione *ECDSAverify*, che richiede in ingresso il messaggio che è stato firmato e la sua dimensione, il puntatore ai parametri della firma e una struttura contenente la chiave pubblica e i parametri della curva secp256k1; se la firma è valida il codice prosegue con la sua esecuzione, altrimenti viene visualizzato un messaggio di errore e l'esecuzione si interrompe.

```
/* Try to verify the signature of the message */
verctx.pmEC = &EC_st;
verctx.pmPubKey = PubKey;

/* Verify the signature with the digest */
status = ECDSAverify(hash_transaction, CRL_SHA256_SIZE, sign, &verctx, &Crypto_Buffer);
```

Una volta verificata la validità della firma, si procede con la creazione della transazione vera e propria: si implementa nuovamente la funzione *assembleTx* per serializzare i parametri della transazione, che rimangono gli stessi del payload precedentemente creato. L'unica differenza è rappresentata dai parametri r, s e v della firma che si vanno a inserire: nell'implementazione del payload i parametri r e s sono stati posti uguali a zero e il parametro v uguale all'ID della rete blockchain su cui si vuole registrare la transazione [26]; in questo caso, invece, come parametri r e s vengono inseriti i corrispondenti valori ottenuti dall'algoritmo di firma, dopo averli opportunamente convertiti in stringhe di caratteri, come richiesto dalla funzione.
Per quanto riguarda invece il parametro v, bisogna seguire quanto raccomandato dall'EIP-155 per la sua implementazione [27]. Per prima cosa c'è da sottolineare l'importanza del parametro v in quanto permette al destinatario di una transazione, che conosce solo i parametri r e s della firma, di risalire al corretto valore della chiave pubblica del mittente; infatti, a causa della simmetria della curva ellittica rispetto all'asse x (vedi figura 3), al valore di una coordinata x della chiave possono corrispondere due possibili valori di y, uno sull'asse positivo e uno sull'asse negativo. Il parametro v determina proprio quali coordinate (e quindi quale chiave pubblica) scegliere tra le due possibili combinazioni; la sua determinazione dipende dalla parità dell'ultimo bit della coordinata y del punto R(x,y), quello cioè che viene generato dal prodotto tra il numero casuale k e il punto generatore G nel processo di firma.
Come già illustrato, durante la fase di generazione del payload, v va posto uguale all'ID del network su cui si sta lavorando, che nel caso in esame corrisponde a quello di Hyperledger Besu; questo network ha un chainID uguale a 1981, per cui v viene posto uguale a tale valore. Una volta effettuata la firma dell'hash del payload e aver inserito i parametri della firma nella transazione grezza, è necessario calcolare il corretto valore di v da inserire. L'EIP-155 raccomanda, in questo caso, di calcolare v come [27]:

$$v=\{0,1\}+(chainID * 2) +35$$

dove {0,1}rappresentano i due possibili valori determinati dalla verifica di parità di y, cioè sul fatto che l'ultimo bit della coordinata y di R sia pari o dispari.
Dato che l'ID del network è codificato nel parametro v della firma, non è necessario includere il chainID come parametro a se stante nella transazione finale firmata.

Purtroppo la funzione di firma della libreria X-CUBE-CRYPTOLIB qui usata non permette di risalire al numero casuale k generato ma solamente allo stato del PRNG, per cui risulta impossibile determinare con certezza la coordinata y del punto R, che viene scartata; per tale ragione, si è scelto di utilizzare di default sempre il bit 0 come risultato della verifica di parità. Ciò produce, in alcune transazioni, un'inesattezza nell'indirizzo del mittente in quanto viene considerata la coordinata y opposta rispetto a quella della chiave che si sta utilizzando.

```
int8_to_char(signR, 32, r_char);       //convert parameter r from uint8 to char
int8_to_char(signS, 32, s_char);       //convert parameter s from uint8 to char
v_final = v + (chainID *2) + 35;        //calculate v

assembleTx(completerawTx, tx, signature, completeraw_tx_bytes, nonce_char, gas_price, gas_limit, to, value, data, r_char, s_char, v_final);
HAL_UART_Transmit(&huart3, completerawTx, sizeof(completerawTx), 1000);
HAL_UART_Transmit(&huart3, Space, sizeof(Space), 10);
printf("Signature VALID\n");
```

La funzione *assembleTx* restituisce l'array *completerawTx* in cui è contenuta la transazione vera e propria, correttamente firmata con la chiave privata del mittente. Una volta effettuate tutte le operazioni descritte, compresa la visualizzazione della transazione su monitor seriale, vengono liberate tutte le strutture contenenti i valori relativi alla coppia di chiavi, al numero casuale k e parametri di firma e si procede con l'implementazione della transazione successiva.

5 Risultati del benchmark sul microcontrollore STM32F439ZI

Utilizzando la libreria X-CUBE-CRYPTOLIB, la libreria crittografica certificata messa a disposizione dalla ST, è stato possibile creare un codice con funzioni specifiche in C che prevedono:

1. la generazione del digest di un messaggio fornito in input e lungo 32 byte tramite la funzione hash SHA256;

2. la generazione di una coppia di chiavi che utilizza i parametri della curva ellittica secp256k1: in questo modo vengono create una chiave privata di 32 byte e una chiave pubblica, generata a partire da quella privata, di 64byte;

3. la firma digitale, tramite chiave privata, del digest del messaggio; la firma utilizza l'algoritmo ECDSA e viene suddivisa in due parametri, r e s, entrambi di 32 byte;

4. la verifica di tale firma tramite la corrispondente chiave pubblica.

I passaggi sopra descritti presentano tempi di esecuzione diversi tra loro che dipendono da diversi fattori come la natura del RNG utilizzato, le librerie (e quindi le relative funzioni) utilizzate e soprattutto il tipo di operazione che si va ad eseguire. Per poter fare un'analisi comparativa di questi dati si è scelto di misurare il numero di cicli della CPU che si verificano in ognuna di queste 4 operazioni principali grazie all'abilitazione del registro DWT (Data Watchpoint and Trace Unit) della scheda e del relativo counter dei cicli CYCCNT; il numero dei cicli eseguiti durante un'operazione viene ricavato facendo la differenza tra il numero di cicli eseguiti prima e dopo la suddetta operazione e il risultato viene trasmesso tramite UART e rielaborato in Matlab.
In primo luogo, i test totali realizzati sono stati suddivisi a partire dal tipo di funzione RNG utilizzata per la generazione della chiave privata:

- Funzione PRNG fornita dalla libreria X-CUBE-CRYPTOLIB.

- Chiavi random generate in Matlab e fornite al microcontrollore tramite file binario.

Per ciascuna di queste due categorie sono stati poi effettuati tre diversi tipi di test, combinando opportunamente la tipologia del messaggio in input e della chiave privata:

- **Test 1- Fixed message, random keys**: il messaggio di cui si va a fare l'hash è lo stesso per ciascuna misurazione ed è una stringa di 32 byte in cui il primo byte è uguale a 1 mentre gli altri sono tutti zeri; la chiave privata è invece random e cambia ad ogni misurazione.

- **Test 2- Random messages, fixed key**: il messaggio in input è diverso ad ogni misurazione e generato in modo random; la chiave privata invece è sempre la stessa ed è definita come un array di 32 byte in cui i byte assumono valori crescenti da 0 a 31 (ossia il primo byte vale 0, il secondo 1, il terzo 2 e così via).

- **Test 3- Random messages, random keys**: sia il messaggio sia la chiave privata sono random e diverse per ogni misurazione effettuata.

Il numero di misurazioni effettuate per queste prove è pari a 1000; per ciascuna di esse si è analizzato il comportamento delle funzioni per la generazione dell'hash, della coppia di chiavi, della firma e della verifica, andando a graficare ed elaborare in Matlab il numero di cicli necessari per l'esecuzione di ciascuna funzione e andando a estrarre i seguenti parametri:

- Valore minimo;
- Valore massimo;
- Media;
- Deviazione standard.

Come già accennato, per la misurazione del numero di cicli di clock impiegati per l'esecuzione di ogni operazione crittografica è stata utilizzata l'unità DWT, che viene abilitata eseguendo i seguenti comandi:

```
/* USER CODE BEGIN SysInit */
CoreDebug->DEMCR |= CoreDebug_DEMCR_TRCENA_Msk;          /* Enable Data Watchpoint and Trace Register (DWT) */
//  DWT->LAR = 0xC5ACCE55; //magic word "0xC5ACCE55"       /* Unlock access to DWT (ITM, etc.)registers.
DWT->CYCCNT = 0;                                         /* Clear DWT cycle counter */
DWT->CTRL |= DWT_CTRL_CYCCNTENA_Msk;                     /* Enable DWT cycle counter */
```

Il counter del DWT è il CYCCNT , che conta il numero di cicli eseguiti dalla CPU. Quindi, per l'analisi delle prestazioni della scheda è possibile usare questa unità disponibile in STM32 per contare il numero di cicli eseguiti dalla CPU durante l'esecuzione delle istruzioni di interesse.

5.1 Risultati ottenuti utilizzando la funzione PRNG fornita da X-CUBE-CRYPTOLIB

Come è già stato anticipato, per ogni prova sono state effettuate 1000 misurazioni; nei Test 1 e 3, per produrre una chiave privata diversa ad ogni iterazione è stata utilizzata una funzione della libreria X-CUBE-CRYPTOLIB per la generazione di numeri pseudo-random.

5.1.1 Test 1- Fixed message, random keys

Di seguito sono riportati i grafici Matlab e i valori numerici ottenuti dalle quattro diverse funzioni quando viene fornito in input un messaggio uguale per tutte e 1000 le misurazioni avente, come già detto, il primo byte uguale a 1 e tutti gli altri 0.

```
/* Fixed 32 byte message, random key */
unsigned MESSAGE_SIZE=32;
uint8_t InputMessage[MESSAGE_SIZE];  //Create a message of 32 byte: the first byte is 1,
InputMessage[0]=1;                   //the others 0
for(int i=1; i< MESSAGE_SIZE; i++) {
      InputMessage[i]=0;
}
```

A differenza del codice descritto nel capitolo 4 dove la chiave privata viene ogni volta generata a partire da una stringa fornita dal TRNG del microcontrollore, in questo test la chiave privata viene prodotta da una funzione della libreria X-CUBE che genera numeri pseudo-random. La funzione STM32_Init_RNG_for_Sign, infatti, a partire da un certo stato iniziale, genera dei numeri pseudo-random che vengono poi forniti ogni volta alla funzione *ECCkeyGen* per generare una chiave privata e, da essa, una chiave pubblica.

```
status = STM32_Init_RNG_for_Sign( &RNGstate, Num_Test);
if (status == ECC_SUCCESS)
{
  /* Generate ECC key pair */
   t1= DWT->CYCCNT;
   status = ECCkeyGen(PrivKey, PubKey, &RNGstate, &EC_st, &Crypto_Buffer);
   t2= DWT->CYCCNT;
   delta= t2 - t1;
   delta_TX[0] = delta >> 16;
   delta_TX[1] = delta;
   HAL_UART_Transmit(&huart3, delta_TX, sizeof(delta_TX),10);
```

Si osserva dal codice sopra riportato che, per misurare il numero dei cicli di clock impiegati dal microcontrollore per eseguire la generazione della coppia di chiavi, si esegue l'istruzione *DWT->CYCCNT* prima e dopo la funzione in modo da misurare il numero totale di cicli eseguiti; il risultato dell'operazione viene poi trasmesso tramite UART, in modo da poter essere rielaborato in Matlab per l'estrazione dei vari plot. L'operazione relativa al calcolo dei cicli di clock viene applicata a tutte le altre funzioni della libreria X-CUBE-CRYPTOLIB descritte nel capitolo 4; le funzioni coinvolte saranno dunque:

- *STM32_SHA256_HASH_DigestCompute* per il calcolo dell'hash tramite SHA256;
- *ECCkeyGen* per la generazione della coppia di chiavi a partire da un numero pseudo-random usato come chiave privata;
- *ECDSAsign* per la firma del messaggio;
- *ECDSAverify* per la verifica della firma.

In seguito sono riportati i risultati ottenuti dall'analisi del numero di cicli di clock richiesti per ciascuna operazione.
Il grafico 8 mette a confronto il numero dei cicli di esecuzione per tutte e quattro le funzioni esaminate, ossia *STM32_SHA256_HASH_DigestCompute*, *ECCkeyGen*, *ECDSAsign* e *ECDSAverify*. I grafici 9, 10, 11, 12 rappresentano invece nel dettaglio l'andamento dei cicli richiesti per ciascuna delle funzioni sopra elencate; infine, la tabella 1 mostra il valore massimo, minimo, medio e deviazione standard ottenuti per ciascuna funzione su 1000 misurazioni effettuate.

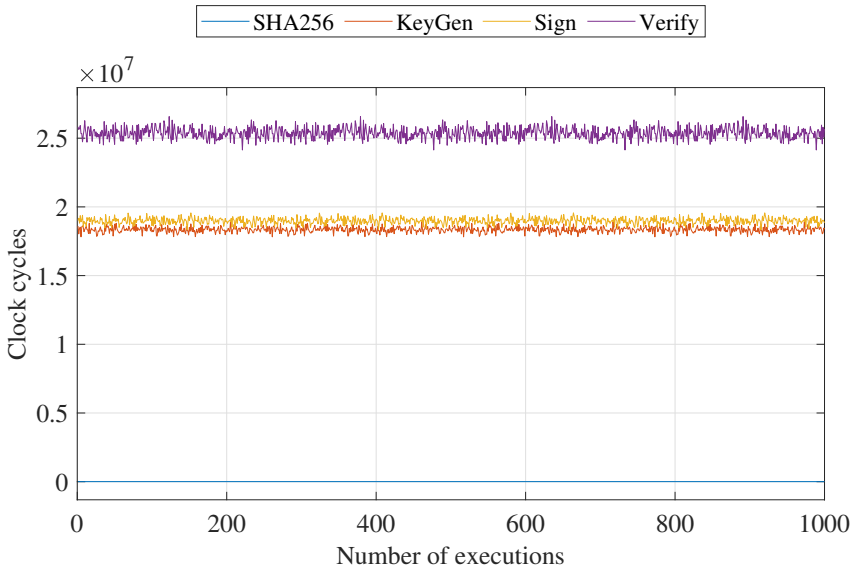


Figura 8: *Andamento del numero dei cicli di clock per tutte e 4 le operazioni*

Funzione	**Valore minimo**	**Valore massimo**	**Media**	**Deviazione standard**
SHA256	5375	5439	5377	9,9705
KeyGen	$1782 \cdot 10^4$	$1891 \cdot 10^4$	$1835 \cdot 10^4$	$21,064 \cdot 10^4$
Sign	$1833 \cdot 10^4$	$1957 \cdot 10^4$	$1898 \cdot 10^4$	$23,087 \cdot 10^4$
Verify	$2414 \cdot 10^4$	$2658 \cdot 10^4$	$2537 \cdot 10^4$	$41,853 \cdot 10^4$

Tabella 1: Valore minimo, massimo, media e deviazione standard dei cicli di clock necessari per l'esecuzione di ciascuna funzione su 1000 misurazioni effettuate.

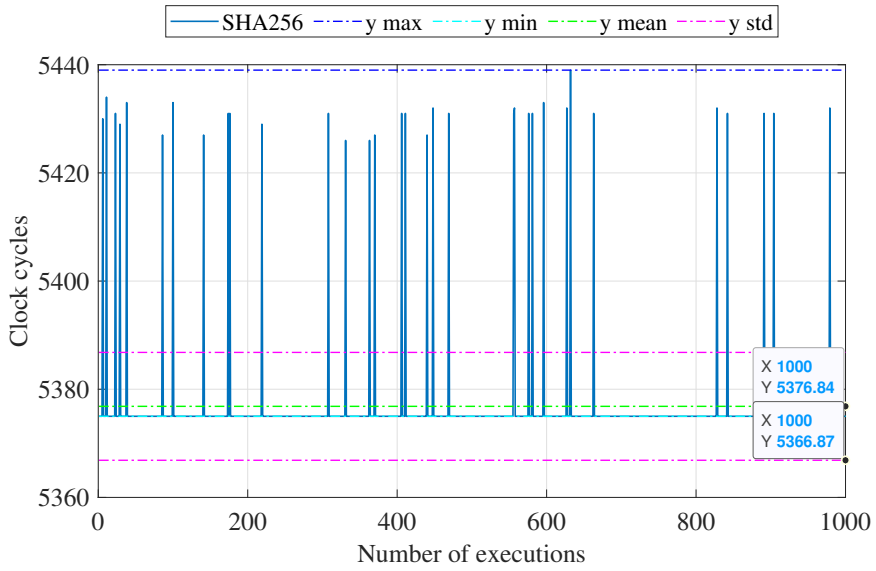


Figura 9: *Andamento del numero dei cicli di clock per l'esecuzione della funzione SHA256*

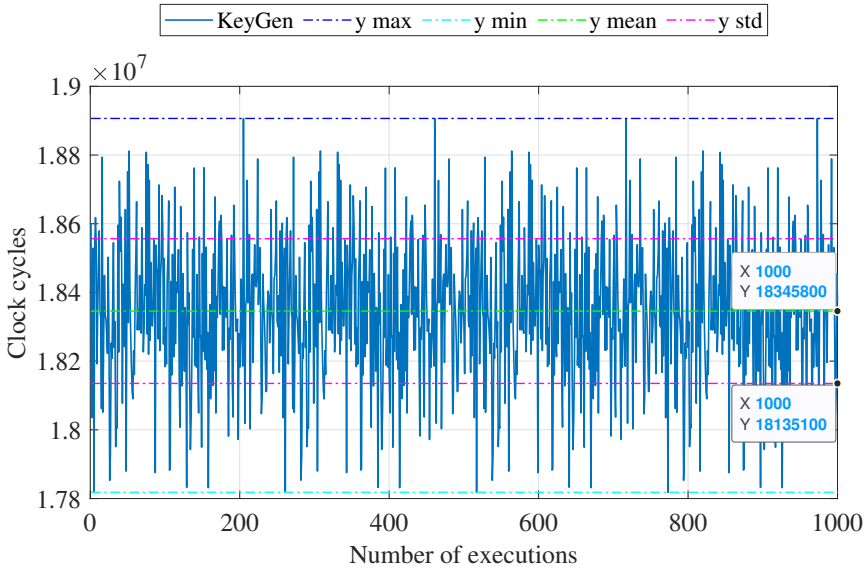


Figura 10: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la generazione della coppia di chiavi*

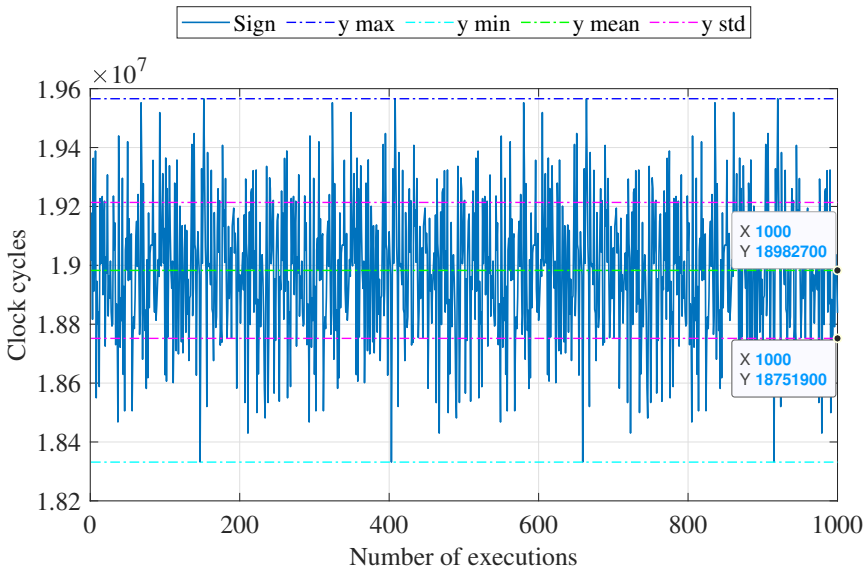


Figura 11: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la firma*

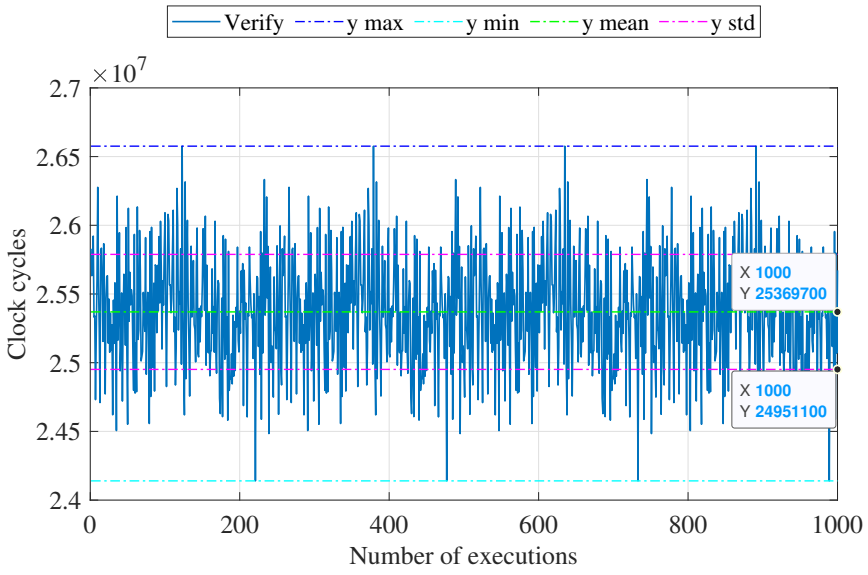


Figura 12: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la verifica*

Dai grafici e dalla tabella si osserva come la funzione per la generazione dell'hash SHA256 sia la più veloce ed efficiente in termini di esecuzione, così come in generale tutte le funzioni di hashing. La differenza del numero di cicli per SHA256 rispetto alle altre funzioni crittografiche è inferiore di ben 4 ordini di grandezza; inoltre la funzione di hashing risulta essere la più lineare e costante per tutte le esecuzioni, cioè il numero dei cicli rimane quasi sempre invariato per tutte e 1000 le misure.
Le funzioni per la generazione della coppia di chiavi e per la firma digitale del messaggio sono invece molto simili nei tempi di esecuzione e variabilità, mentre la funzione meno efficiente risulta essere quella per la verifica della firma, essendo molto variabile tra una misura e l'altra (si veda a tal proposito il valore abbastanza elevato della deviazione standard).

5.1.2 Test 2- Random messages, fixed key

Per effettuare queste 1000 misurazioni è stato utilizzato il TRNG della libreria HAL di STM per generare ogni volta un messaggio random casuale lungo 32 byte (si rivedano i capitoli precedenti per capire il funzionamento del TRNG):

```
/* Random 32 byte message, fixed key */
  unsigned MESSAGE_SIZE=32;
  uint8_t InputMessage[MESSAGE_SIZE];
  HAL_RNG_GenerateRandomNumber(&hrng, &trng);
  InputMessage[0]=trng;
  for(int i=1;i<MESSAGE_SIZE; i++) {              //Generate a 32 byte random message
      HAL_RNG_GenerateRandomNumber(&hrng, &trng);
      InputMessage[i]=trng;
       }
```

La chiave privata, come già anticipato, è un array di 32 byte di valori crescenti da 0 a 31 che rimane uguale per tutte e 1000 le misurazioni. Anche qui, come per il test precedentemente descritto, viene effettuata la stessa misurazione con il DWT per il calcolo del numero di cicli di clock. In questo caso, però, per la generazione della coppia di chiavi vengono utilizzate due diverse funzioni: *ECCsetPrivKeyValue* per settare i valore della chiave privata con quello fornito in input e *ECCscalarMul* per la generazione della chiave pubblica tramite moltiplicazione con il punto generatore.

```
/*Create a fixed private key */
   uint8_t privatekey[32];
   for(int k=0; k<32; k++) {
         privatekey[k]=k;
    }


   t1= DWT->CYCCNT;
   status= ECCsetPrivKeyValue(PrivKey, privatekey, sizeof(privatekey));
   status = ECCscalarMul(G, PrivKey, PubKey, &EC_st, &Crypto_Buffer);
   t2= DWT->CYCCNT;
   delta= t2 - t1;
   delta_TX[0] = delta >> 16;
   delta_TX[1] = delta;
   HAL_UART_Transmit(&huart3, delta_TX, sizeof(delta_TX),10);
```

Andando a plottare in Matlab le misurazioni dei cicli di clock ottenute, si ricavano i risultati mostrati nei grafici 13, 14, 15, 16, 17 e nella tabella 2.

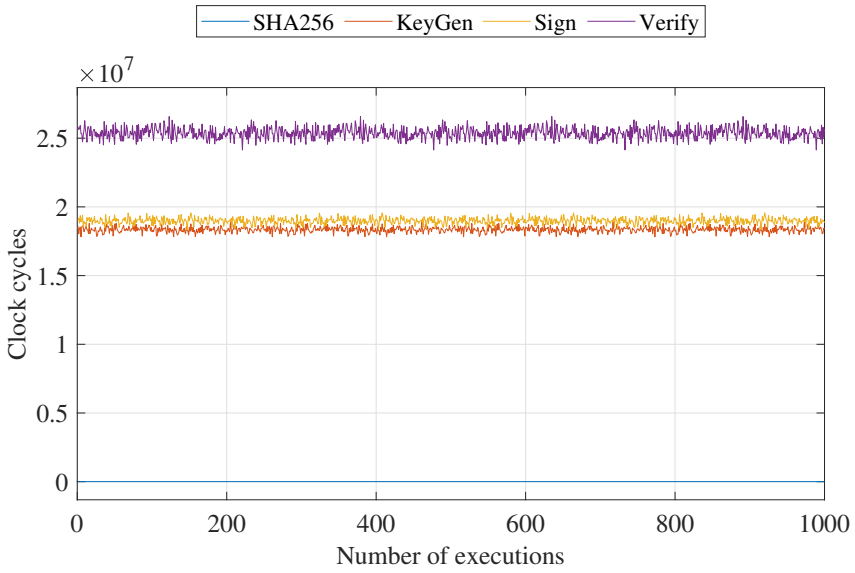


Figura 13: *Andamento del numero dei cicli di clock per tutte e 4 le operazioni*

Funzione	**Valore minimo**	**Valore massimo**	**Media**	**Deviazione standard**
SHA256	5391	5447	5392	8,014
KeyGen	$16603 \cdot 10^3$	$16604 \cdot 10^3$	$16604 \cdot 10^3$	123,151
Sign	$1814 \cdot 10^4$	$1969 \cdot 10^4$	$1894 \cdot 10^4$	$24,455 \cdot 10^4$
Verify	$2382 \cdot 10^4$	$2662 \cdot 10^4$	$2525 \cdot 10^4$	$39,229 \cdot 10^4$

Tabella 2: Valore minimo, massimo, media e deviazione standard dei cicli di clock necessari per l'esecuzione di ciascuna funzione su 1000 misurazioni effettuate.

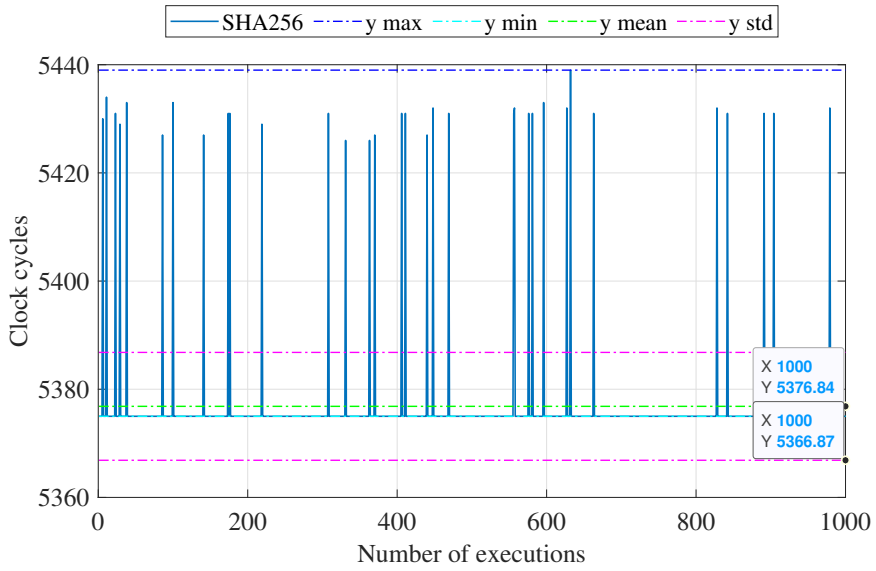


Figura 14: *Andamento del numero dei cicli di clock per l'esecuzione della funzione SHA256*

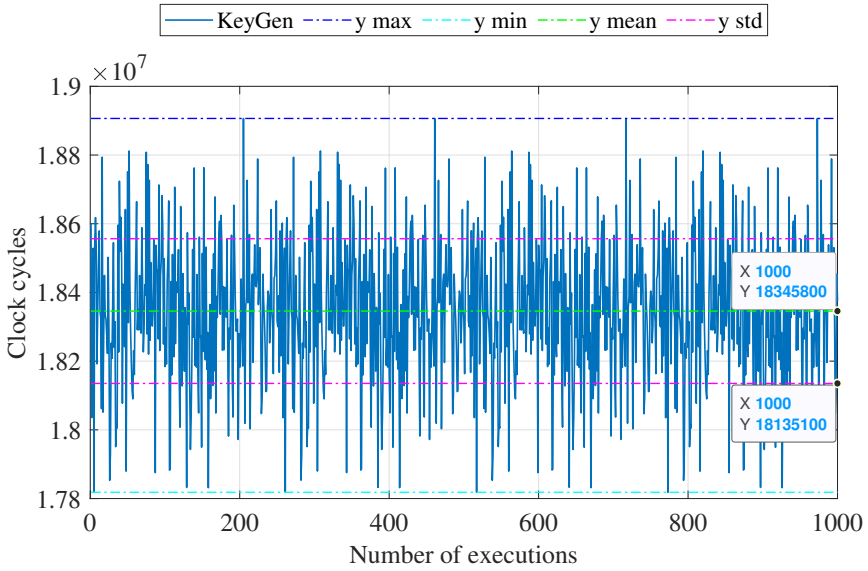


Figura 15: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la generazione della coppia di chiavi*

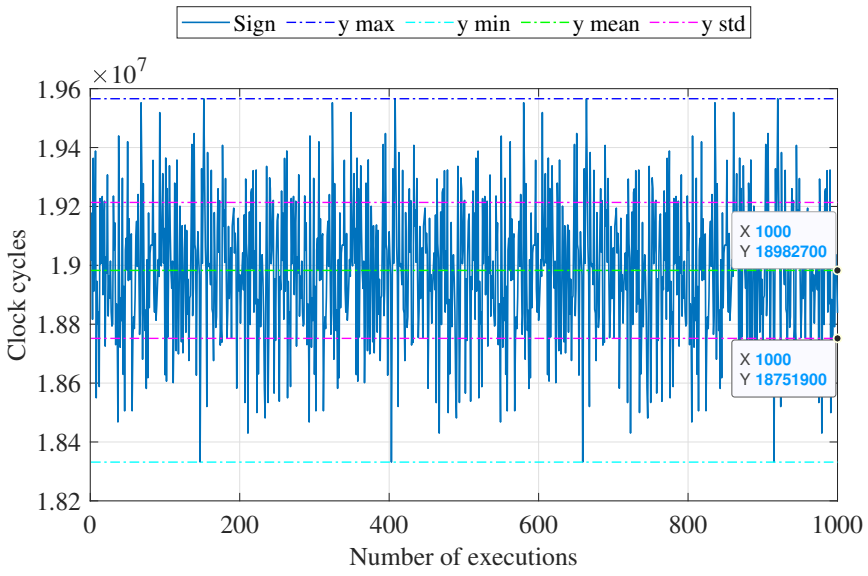


Figura 16: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la firma*

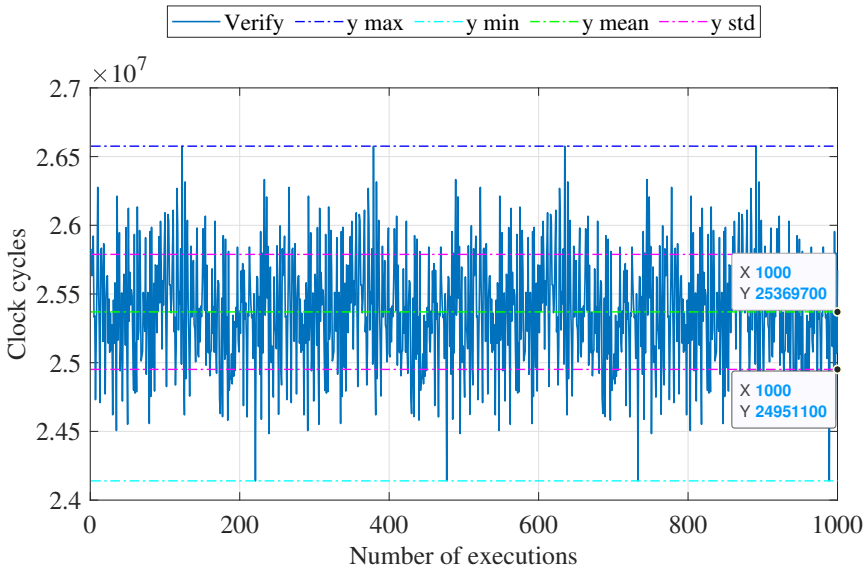


Figura 17: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la verifica*

Anche qui, come nel caso precedente, la funzione di hashing risulta essere la più efficiente e presenta un andamento e dei valori numerici molto simili a quelli che si ottenevano con il messaggio fisso e chiave variabile.
La particolarità di questo test sta nei risultati che si ottengono dalla generazione della coppia di chiavi: infatti si può notare che il numero dei cicli di clock sia più o meno costante per tutte e 1000 le misurazioni (si veda il valore della deviazione standard di circa 123) e come, dal grafico 13, il suo andamento sia più lineare rispetto al caso precedente. Anche qui, l'operazione di verifica risulta essere la meno efficiente in termini di tempo di esecuzione.

5.1.3 Test 3- Random messages, random keys

Per queste misurazioni, esattamente come per il caso precedente, è stato utilizzato il TRNG della libreria HAL per la generazione di un messaggio casuale sempre diverso mentre per la chiave privata è stata adoperata la funzione della libreria X-CUBE-CRYPTOLIB per la generazione di numeri pseudorandom. Il codice usato per la generazione del messaggio random è lo stesso descritto nel test 2, mentre quello per la generazione della coppia di chiavi è lo stesso del test 1; anche il calcolo del numero di cicli di clock mediante CYCCNT è stato implementato sulle stesse funzioni del test1.

Le figure 18, 19, 20, 21, 22 e la tabella 3 mostrano i risultati grafici e numerici ottenuti dal plot in Matlab del numero dei cicli di clock.

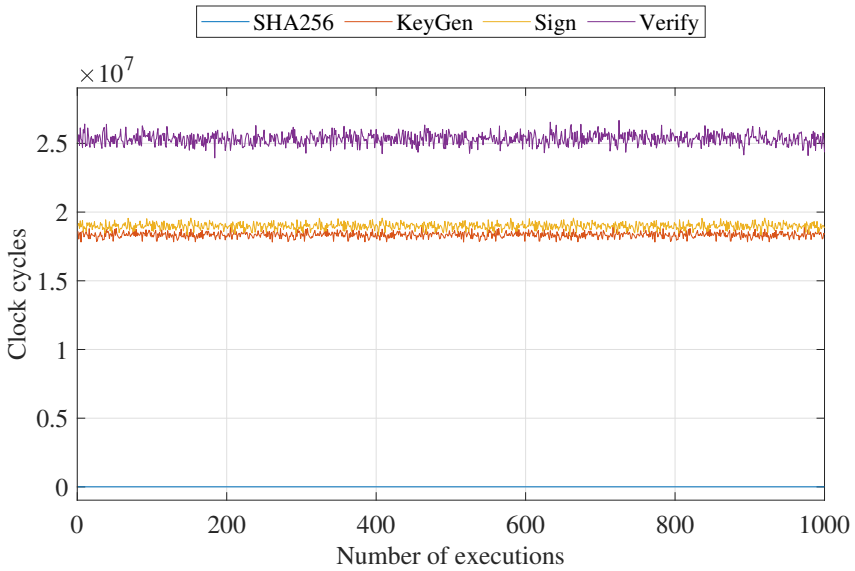


Figura 18: *Andamento del numero dei cicli di clock per tutte e 4 le operazioni*

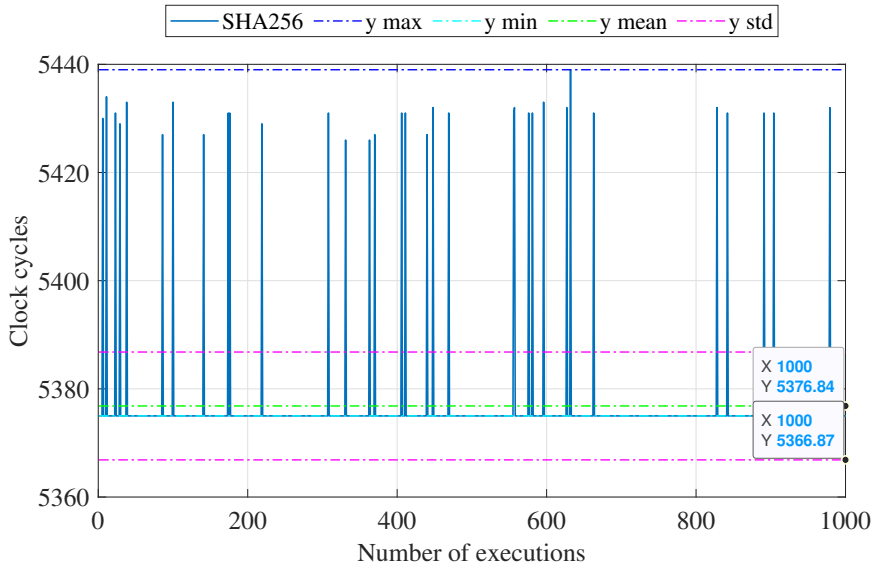


Figura 19: *Andamento del numero dei cicli di clock per l'esecuzione della funzione SHA256*

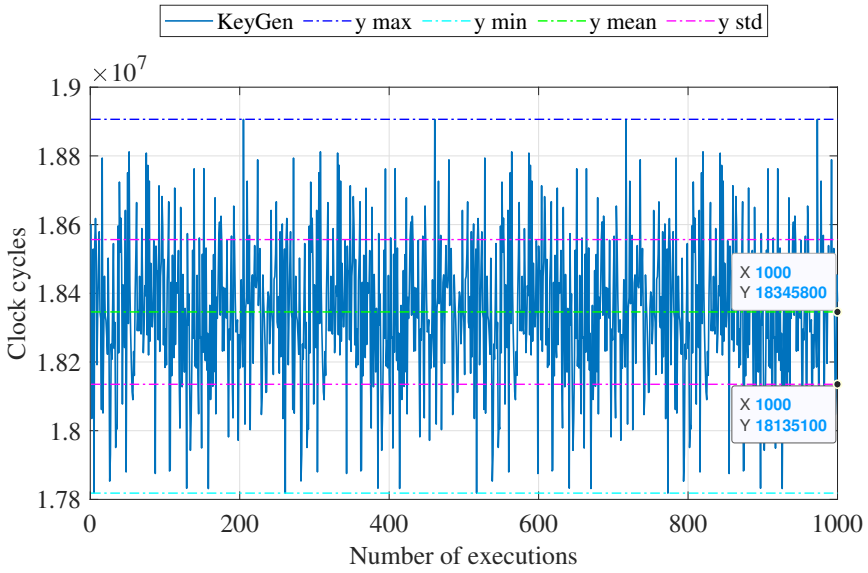


Figura 20: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la generazione della coppia di chiavi*

Funzione	**Valore minimo**	**Valore massimo**	**Media**	**Deviazione standard**
SHA256	5379	5441	5381	10.551
KeyGen	$1782 \cdot 10^4$	$1891 \cdot 10^4$	$1835 \cdot 10^4$	$21,063 \cdot 10^4$
Sign	$1833 \cdot 10^4$	$1957 \cdot 10^4$	$1898 \cdot 10^4$	$23,089 \cdot 10^4$
Verify	$2394 \cdot 10^4$	$2667 \cdot 10^4$	$2533 \cdot 10^4$	$40,798 \cdot 10^4$

Tabella 3: Valore minimo, massimo, media e deviazione standard dei cicli di clock necessari per l'esecuzione di ciascuna funzione su 1000 misurazioni effettuate.

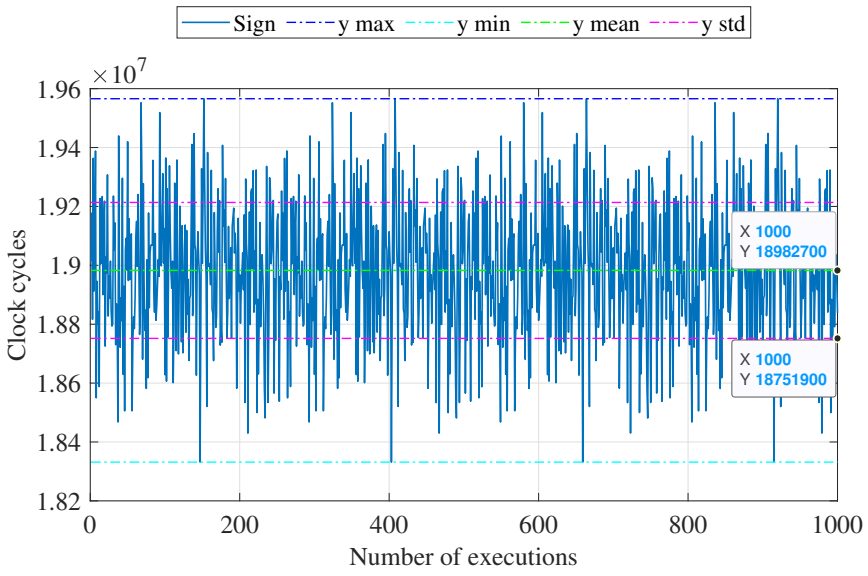


Figura 21: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la firma*

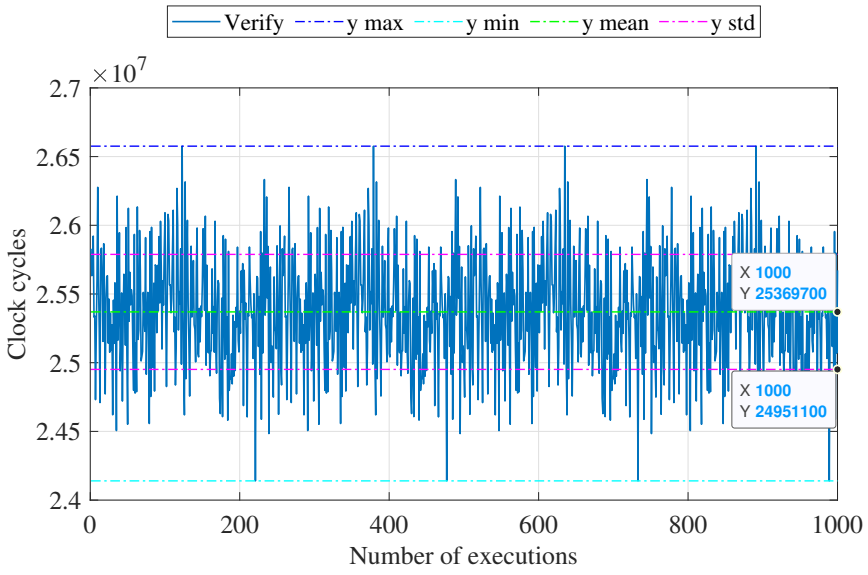


Figura 22: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la verifica*

Dai risultati ottenuti si può constatare come, anche qui, la funzione di hashing sia quella più efficiente e rapida, come in generale ottenuto dai test precedenti; anche dalla generazione della coppia di chiavi e dalla firma digitale del messaggio si ottengono risultati del tutto simili a quelli del test 1, in quanto, in entrambe le situazioni, la chiave privata viene ricavata a partire da un generatore pseudo-random. La verifica della firma risulta invece più rapida e meno variabile rispetto alla corrispondente del test 1, rimanendo tuttavia la funzione più lenta delle 4 esaminate.

5.2 Risultati ottenuti da chiavi generate in modo random da Matlab

Per questa serie di test è stato utilizzato un dataset comune per poter confrontare le misure effettuate con la libreria X-CUBE-CRYPTOLIB e quella uECC. Lo scopo è quello di valutare le performance del microcontrollore e individuare l'algoritmo più efficiente tra i due.
Per la generazione delle stesse chiavi private è stato utilizzato un codice Matlab che produce 32000 byte random e li trascrive in un file binario, che viene poi passato alla memoria della scheda; in questo modo, dato che ciascun test prevede 1000 misurazioni, si avranno 32000/1000= 32 byte di chiave per ogni misura; alla misura successiva verranno considerati i 32 byte del file binario successivi alla prima misura e così via.
Per i vari test sono riportati sia i grafici ottenuti con la libreria X-CUBE-CRYPTOLIB sia quelli ottenuti con la libreria uECC.

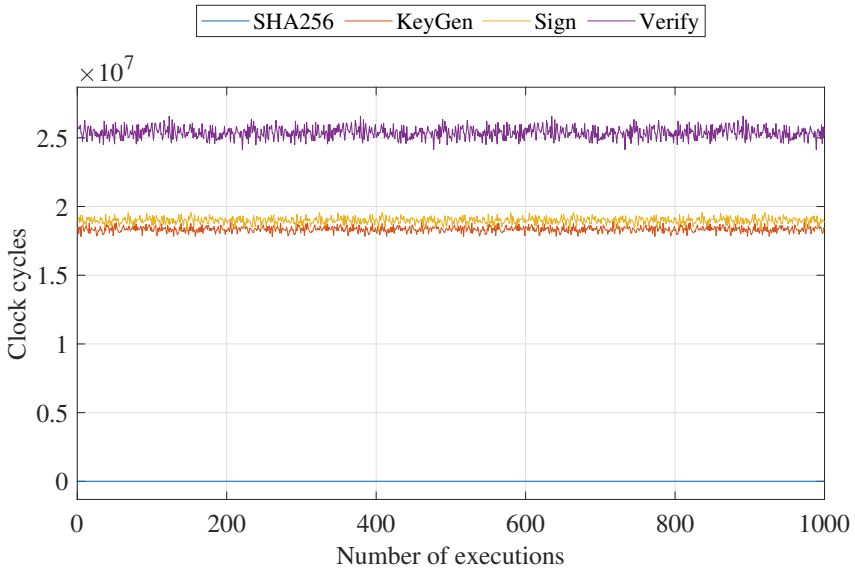
5.2.1 Test 1- Fixed message, random keys

Utilizzando in tutte e 1000 le misurazioni lo stesso messaggio di 32byte (in cui il primo byte è pari a 1 e tutti gli altri sono 0) e chiavi private ogni volta diverse e prelevate ciclicamente dal file binario, sono stati ottenuti in Matlab i grafici illustrati nelle figure 23, 24, 25, 26, 27 e i valori numerici riportati in tabella 4, che mostra un confronto tra i parametri ricavati a partire dalle funzioni della libreria X-CUBE-CRYPTO e quella uECC. Per la generazione del messaggio è stato implementato lo stesso codice usato nella sezione 5.1.1, mentre il valore della chiave privata viene prelevato dal file binario che contiene i 32000 byte. Il codice sotto riportato illustra come viene creata la chiave privata ad ogni iterazione: per prima cosa viene creata una variabile globale di tipo "extern", chiamata *_binary_data_start* e associata al file binario contenente i 32000 byte; ad ogni ciclo di iterazione (ossia per ogni valore di Num_Test che va da 0 a 999) vengono prelevati 32 byte alla volta, che vengono puntati dalla variabile *memPointer*. Il valore puntato rappresenta proprio la chiave privata corrente per quella misurazione.

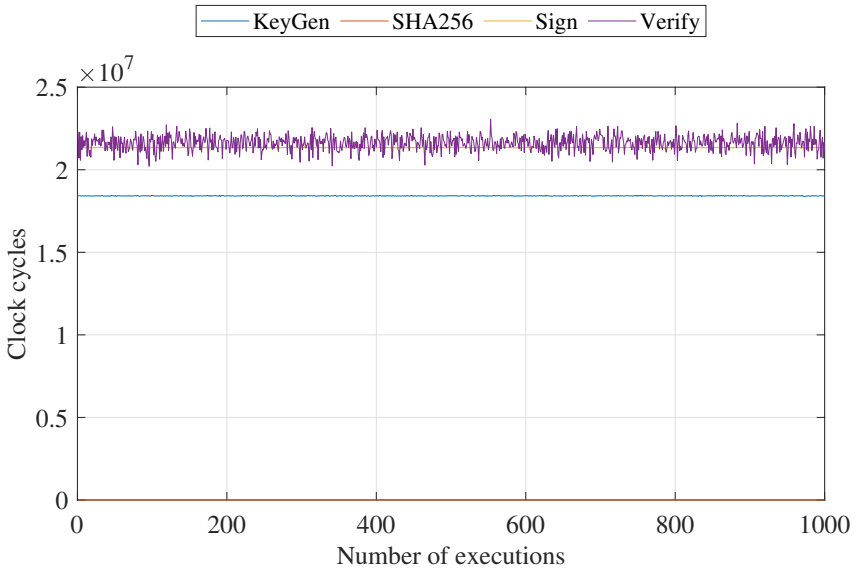
```
/* Create pseudo random keys (TEST 1 and 3) */
 uint8_t *memPointer = &_binary_data_start + 32*Num_Test;
 for(int k=0; k<32; k++) {
     privatekey[k]=memPointer[k];
     }
```

Anche qui viene attivato il registro DWT per il counter del numero di cicli di clock

e le funzioni interessate dal calcolo dei tempi di esecuzione sono le stesse descritte nella sezione 5.1.1.

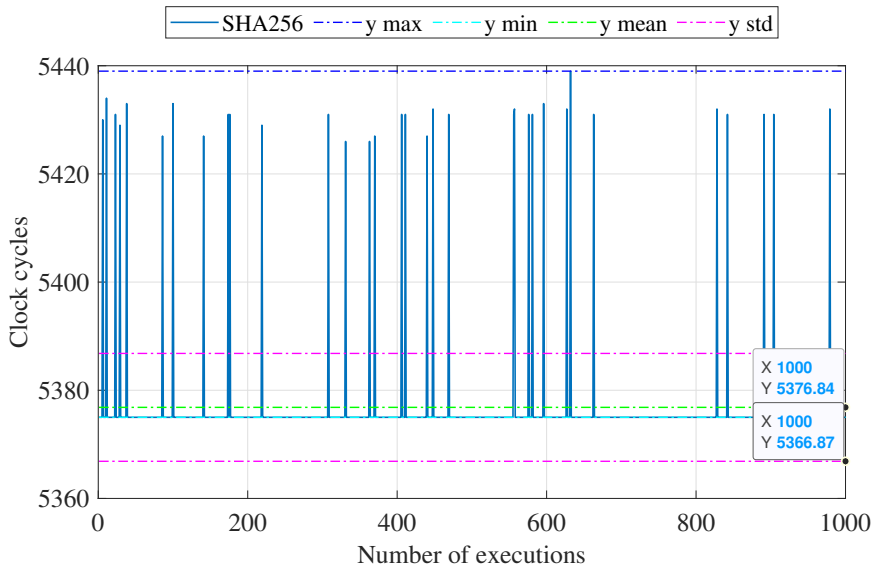


(a)

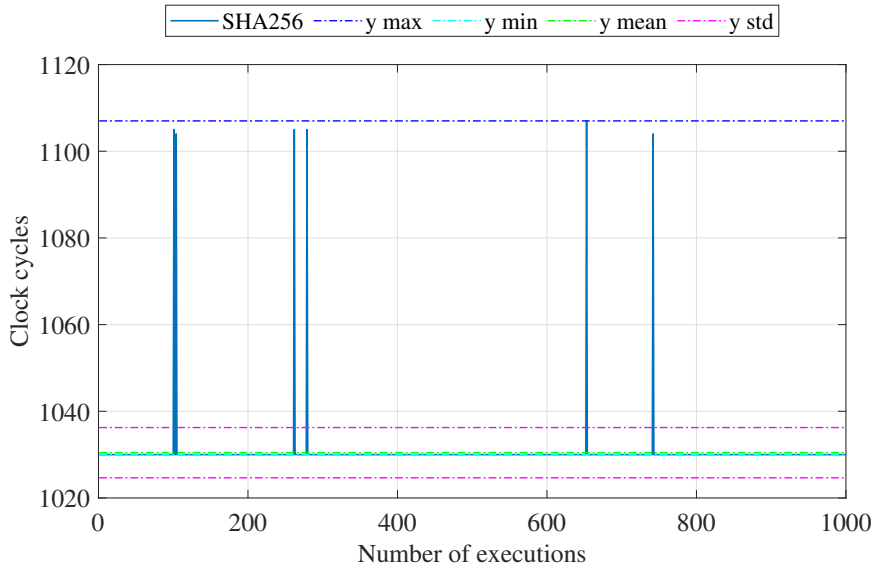


(b)

Figura 23: *Andamento del numero dei cicli di clock per tutte e 4 le operazioni:(a) per le funzioni di X-CUBE-CRYPTOLIB e (b) per le funzioni di uECC*

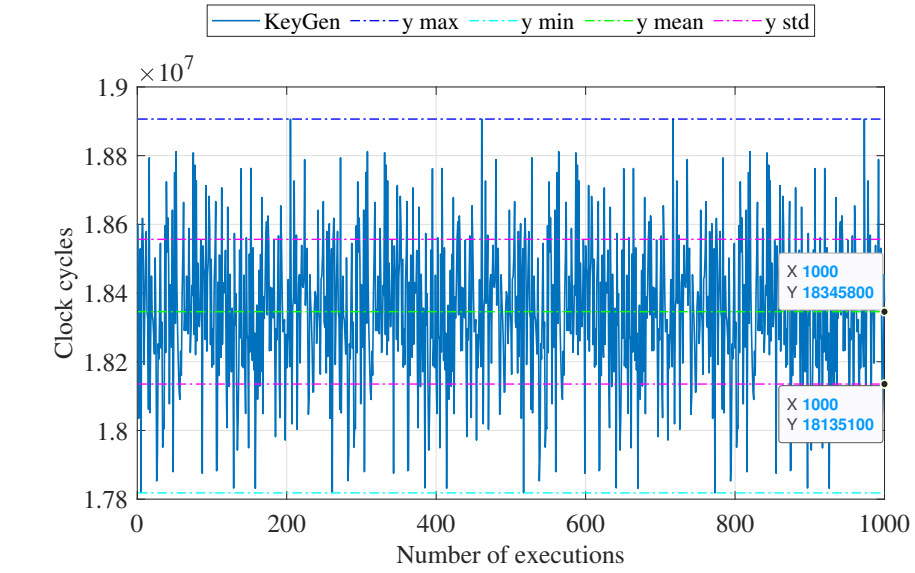


(a)

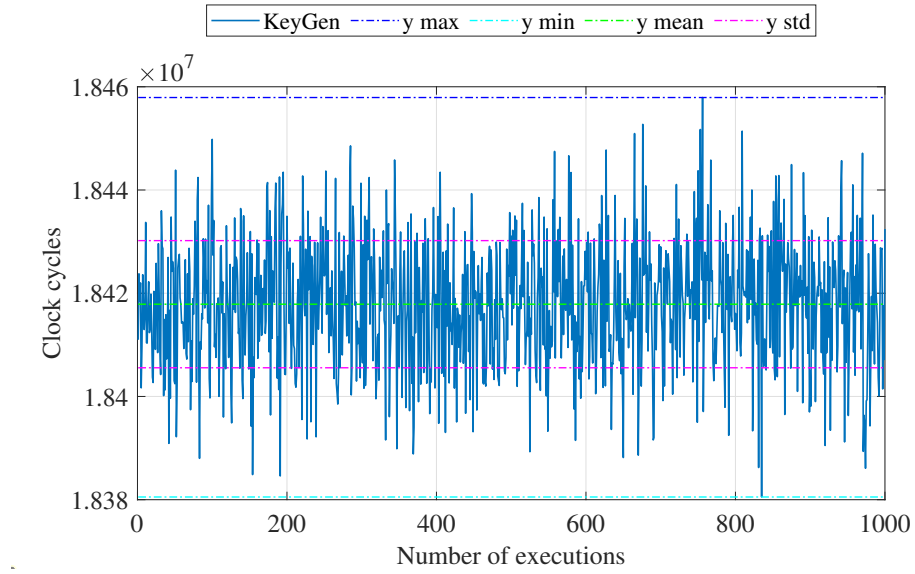


(b)

Figura 24: *Andamento del numero dei cicli di clock per l'esecuzione della funzione SHA256: (a) per le funzioni di X-CUBE-CRYPTOLIB e (b) per le funzioni di uECC*

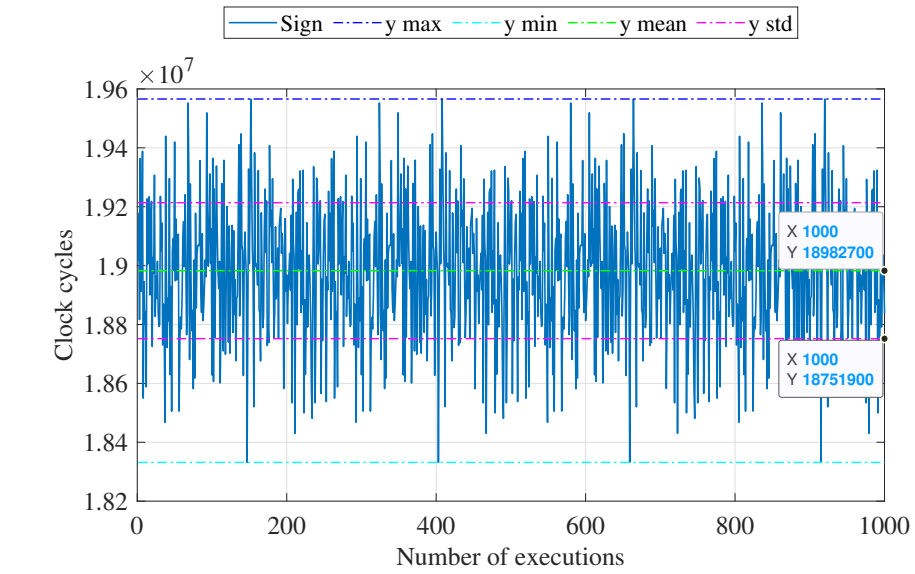


(a)

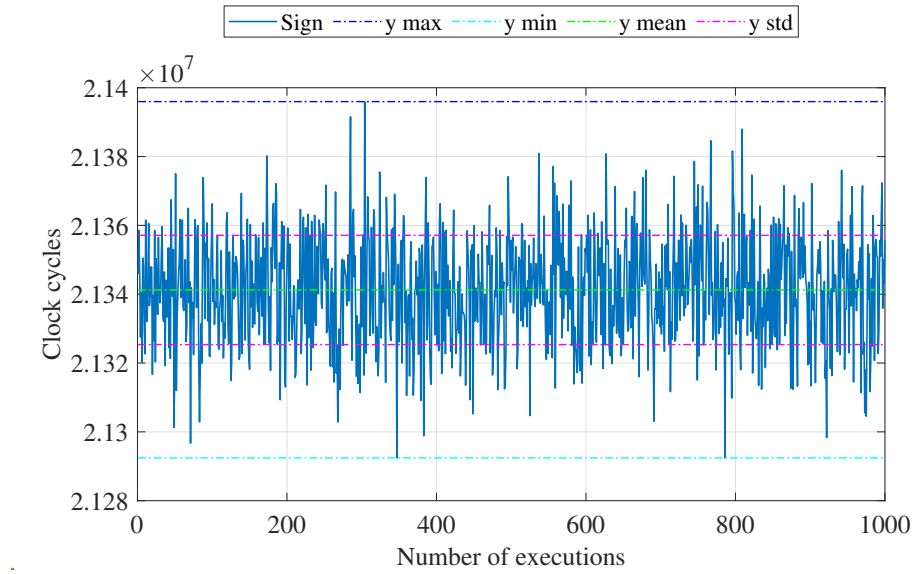


(b)

Figura 25: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la generazione della coppia di chiavi: (a) per le funzioni di X-CUBE-CRYPTOLIB e (b) per le funzioni di uECC*

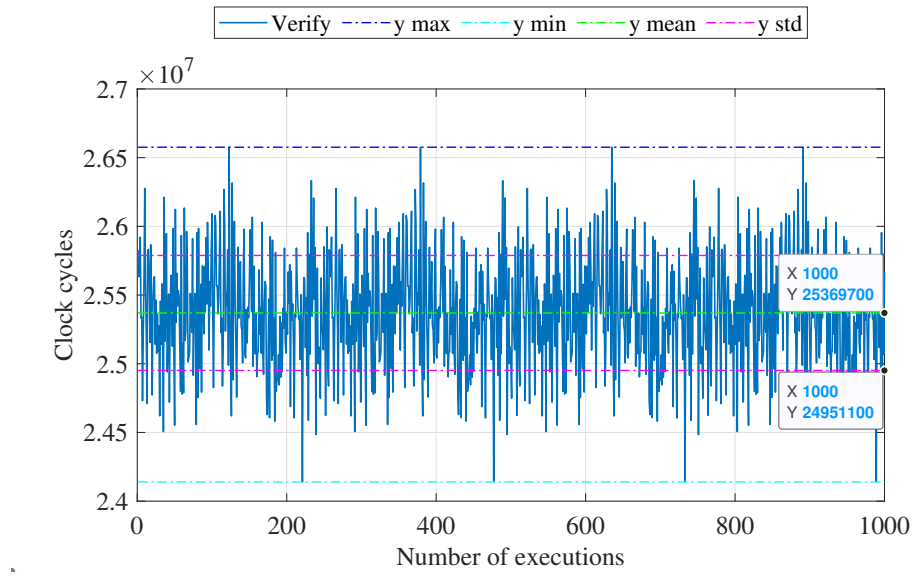


(a)

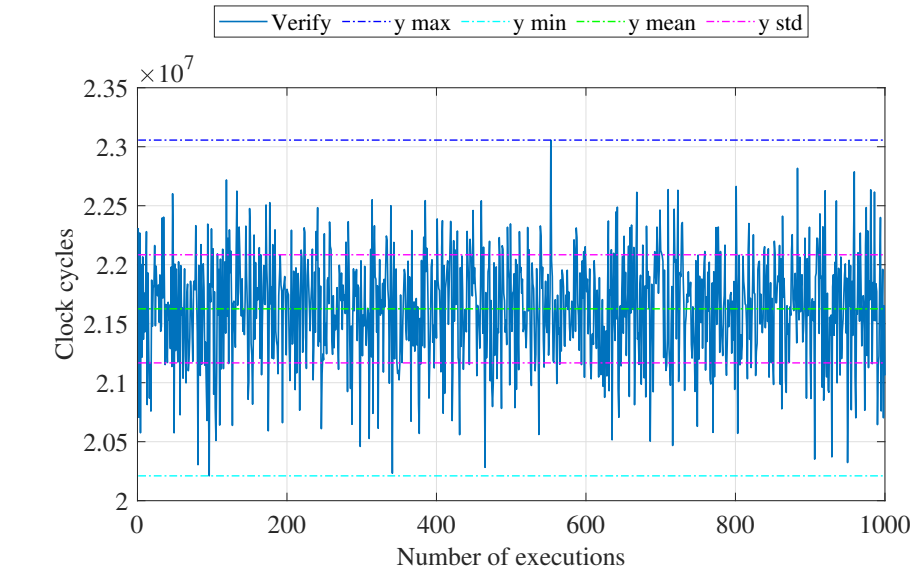


(b)

Figura 26: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la firma: (a) per le funzioni di X-CUBE-CRYPTOLIB e (b) per le funzioni di uECC*



(a)



(b)

Figura 27: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la verifica: (a) per le funzioni di X-CUBE-CRYPTOLIB e (b) per le funzioni di uECC*

Funzione		Valore minimo	Valore massimo	Media	Deviazione standard
SHA256	Cryptolib uECC lib	5394 1030	5450 1107	5396 1030	10.038 5.795
KeyGen	Cryptolib uECC	$1741 \cdot 10^4$ $1838 \cdot 10^4$	$1910 \cdot 10^4$ $1846 \cdot 10^4$	$1829 \cdot 10^4$ $1842 \cdot 10^4$	$22.729 \cdot 10^4$ $1.232 \cdot 10^4$
Sign	Cryptolib uECC	$1810 \cdot 10^4$ $2129 \cdot 10^4$	$1959 \cdot 10^4$ $2140 \cdot 10^4$	$1892 \cdot 10^4$ $2134 \cdot 10^4$	$23.768 \cdot 10^4$ $1.588 \cdot 10^4$
Verify	Cryptolib uECC	$2373 \cdot 10^4$ $2021 \cdot 10^4$	$2682 \cdot 10^4$ $2306 \cdot 10^4$	$2524 \cdot 10^4$ $2163 \cdot 10^4$	$39.690 \cdot 10^4$ $45.825 \cdot 10^4$

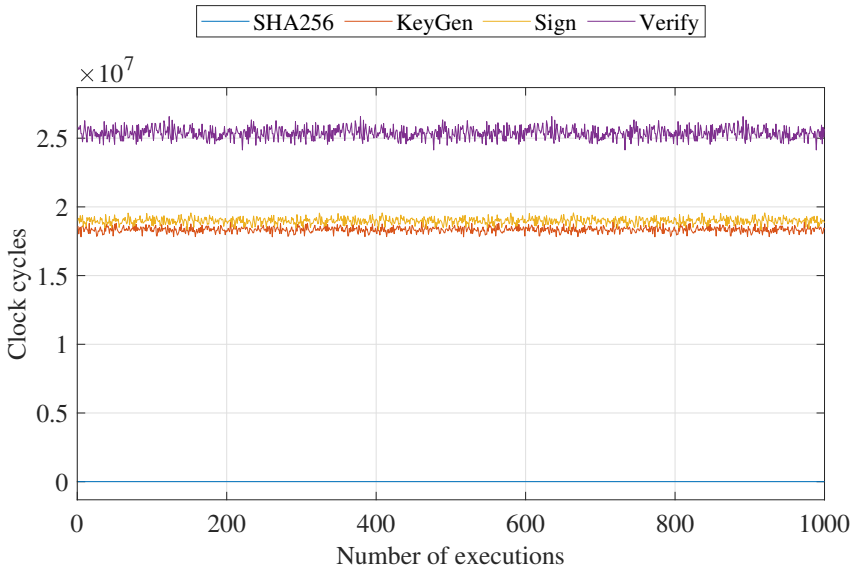
Tabella 4: Confronto tra i parametri ottenuti dal benchmark delle funzioni di X-CUBE-CRYPTOLIB e quelle della libreria uECC per il Test *Fixed message, random key* su un totale di 1000 misurazioni effettuate.

Da un primo confronto dei grafici, si può constatare come i risultati ottenuti siano del tutto simili a quelli del primo test illustrato nella sezione 5.1.1, che però prevedeva una generazione della chiave privata mediante una funzione pseudo-random della libreria X-CUBE. Esaminando invece i dati estratti dalle funzioni implementate dalle due diverse librerie, si possono fare diverse considerazioni:

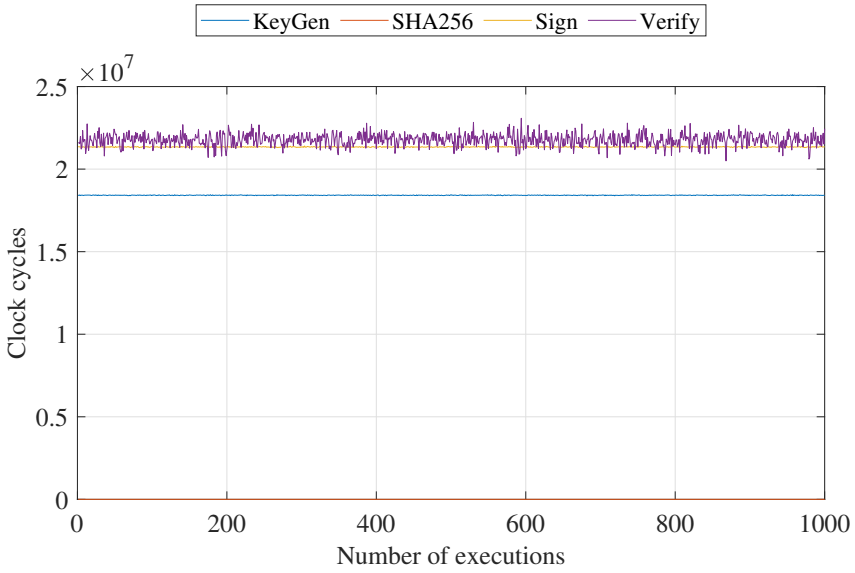
- il tempo di esecuzione impiegato per il calcolo dell'hash è dello stesso ordine di grandezza per le funzioni di entrambe le librerie, anche se l'algoritmo uECC risulta leggermente più efficiente;
- il numero di cicli di clock impiegati per la generazione della coppia di chiavi è circa lo stesso per entrambe; tuttavia, uECC presenta un andamento più lineare e costante al variare delle misurazioni;
- in X-CUBE-CRYPTOLIB, il numero di cicli di clock impiegati per la firma digitale è minore rispetto a quelli utilizzati da uECC; tuttavia, l'andamento di quest'ultima è meno variabile da una misurazione all'altra;
- il numero di cicli di clock per la verifica della firma in X-CUBE è maggiore rispetto a uECC ma la sua deviazione standard (e quindi la sua variabilità) è minore.

5.2.2 Test 2- Random messages, fixed key

Per questa serie di misurazioni, i cui risultati sono visibili nelle figure 28, 29, 30, 31, 32 e in tabella 5, la chiave privata utilizzata è la stessa usata per il test descritto nel sottocapitolo 5.1.2 ed è ovviamente fissa per tutte le misurazioni; il messaggio in input, invece, viene creato a partire dai valori random contenuti nel file binario, prelevati ciclicamente a gruppi di 32 byte, seguendo esattamente lo stesso principio con cui è stata generata la chiave privata nella sezione 5.2.1.

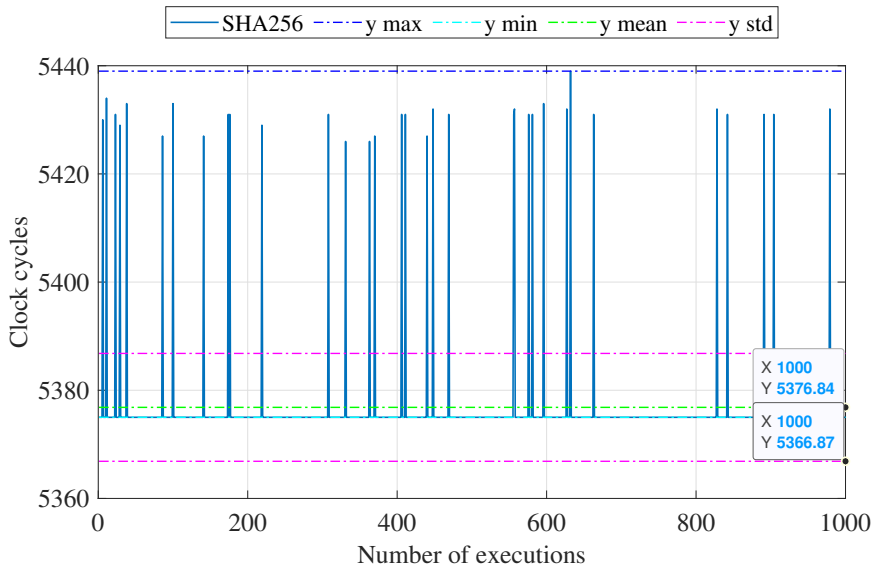


(a)

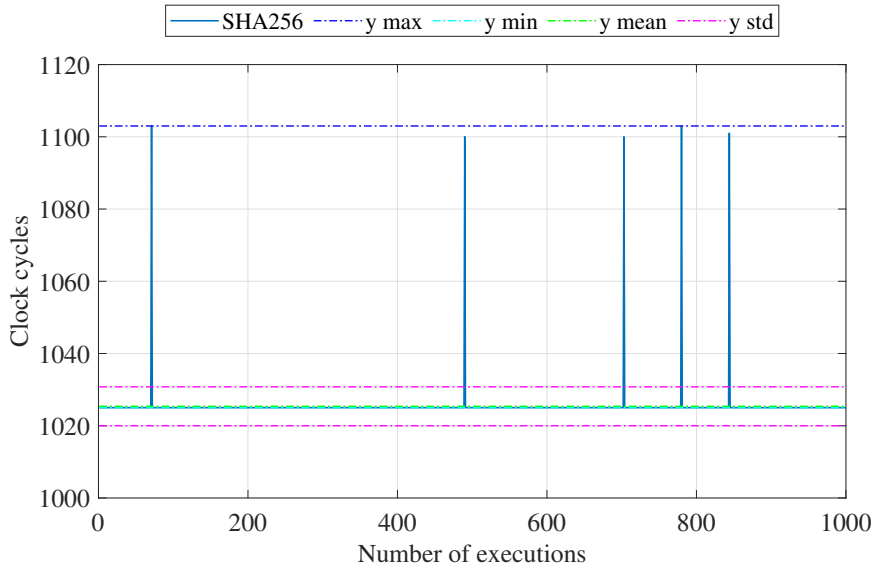


(b)

Figura 28: *Andamento del numero dei cicli di clock per tutte e 4 le operazioni:(a) per le funzioni di X-CUBE-CRYPTOLIB e (b) per le funzioni di uECC*

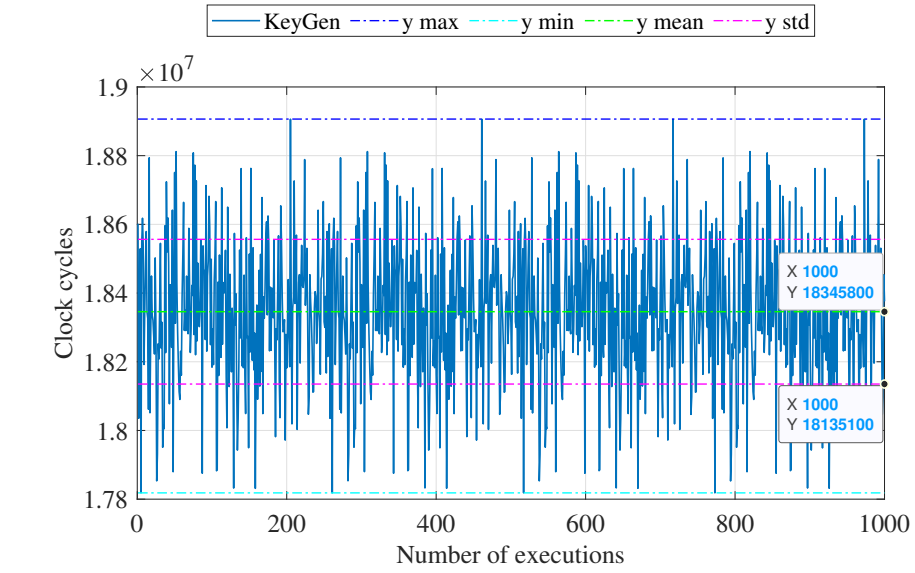


(a)

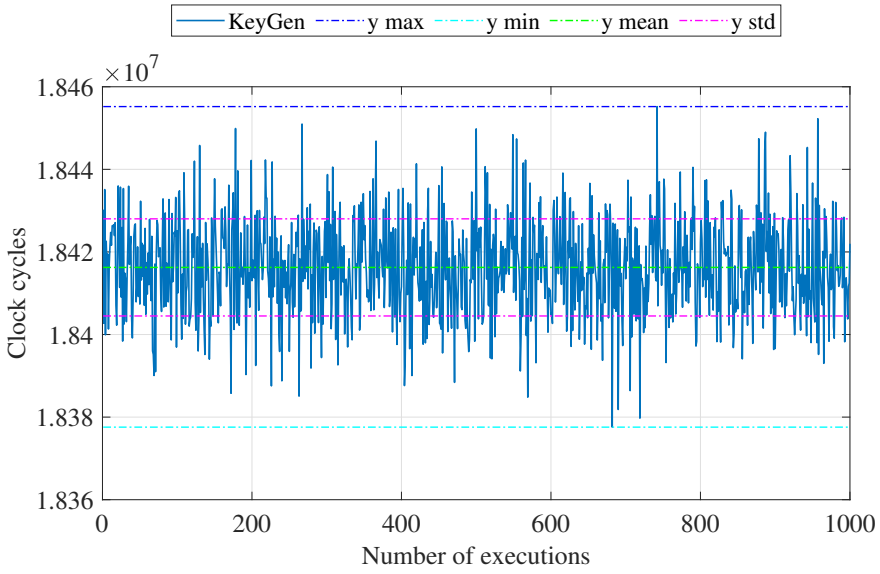


(b)

Figura 29: *Andamento del numero dei cicli di clock per l'esecuzione della funzione SHA256: (a) per le funzioni di X-CUBE-CRYPTOLIB e (b) per le funzioni di uECC*

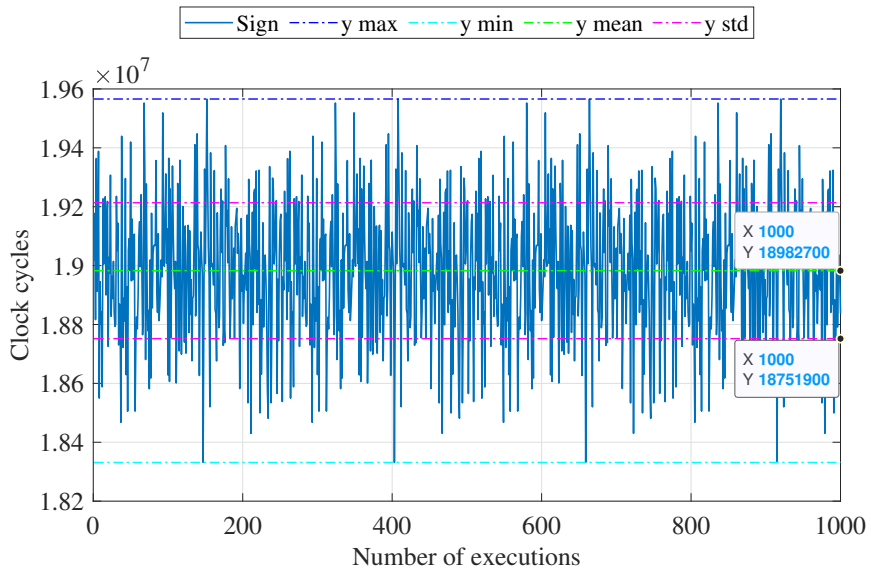


(a)

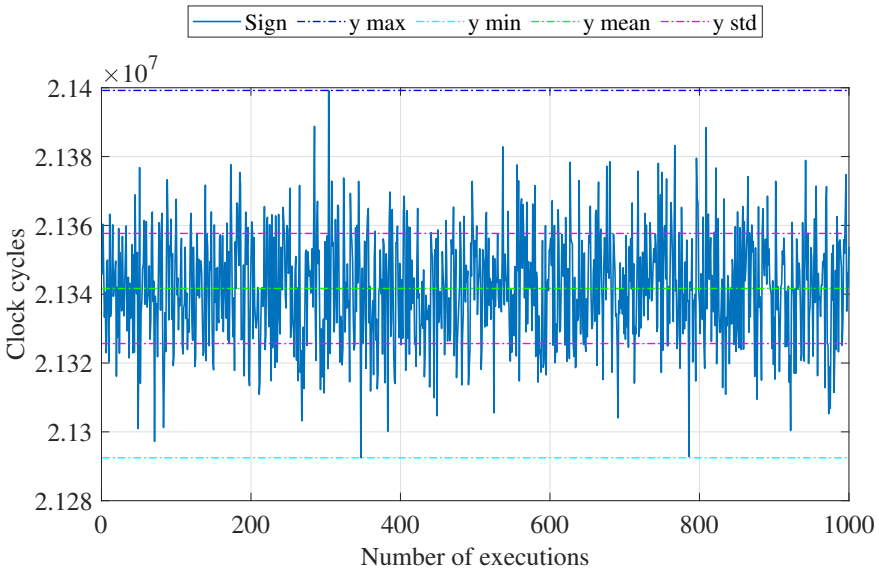


(b)

Figura 30: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la generazione della coppia di chiavi: (a) per le funzioni di X-CUBE-CRYPTOLIB e (b) per le funzioni di uECC*

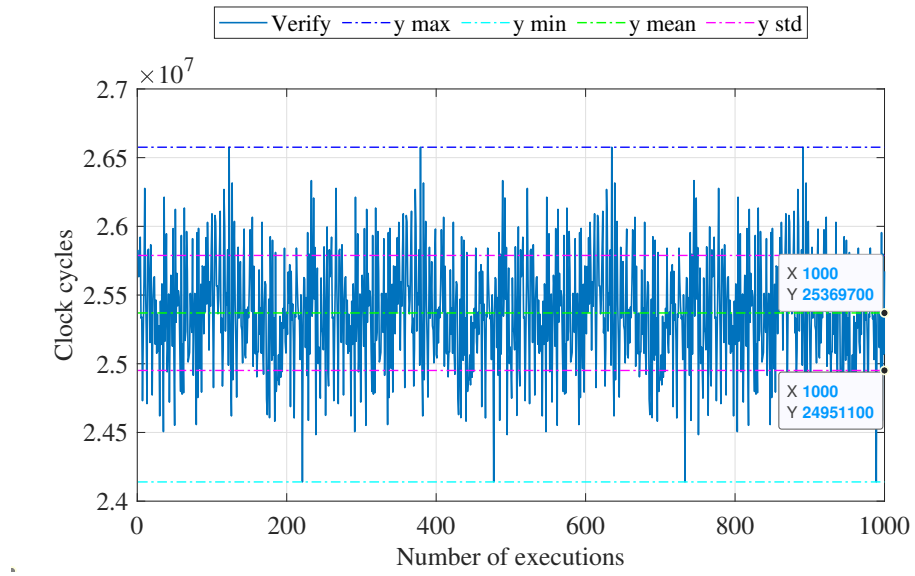


(a)

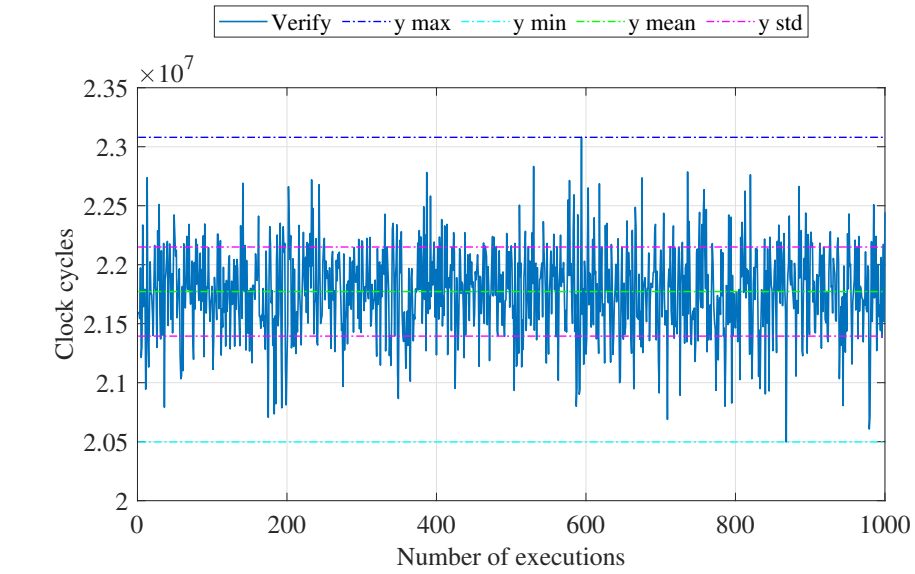


(b)

Figura 31: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la firma: (a) per le funzioni di X-CUBE-CRYPTOLIB e (b) per le funzioni di uECC*



(a)



(b)

Figura 32: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la verifica: (a) per le funzioni di X-CUBE-CRYPTOLIB e (b) per le funzioni di uECC*

Funzione		Valore minimo	Valore massimo	Media	Deviazione standard
SHA256	Cryptolib uECC lib	5387 1025	5455 1103	5389 1025	10.165 5.392
KeyGen	Cryptolib uECC	$16623 \cdot 10^3$ $1838 \cdot 10^4$	$16624 \cdot 10^3$ $1846 \cdot 10^4$	$16623 \cdot 10^3$ $1842 \cdot 10^4$	113.702 $1.176 \cdot 10^4$
Sign	Cryptolib uECC	$1819 \cdot 10^4$ $2129 \cdot 10^4$	$1967 \cdot 10^4$ $2140 \cdot 10^4$	$1896 \cdot 10^4$ $2134 \cdot 10^4$	$23.813 \cdot 10^4$ $1.599 \cdot 10^4$
Verify	Cryptolib uECC	$2384 \cdot 10^4$ $2050 \cdot 10^4$	$2661 \cdot 10^4$ $2308 \cdot 10^4$	$2528 \cdot 10^4$ $2177 \cdot 10^4$	$39.392 \cdot 10^4$ $37.762 \cdot 10^4$

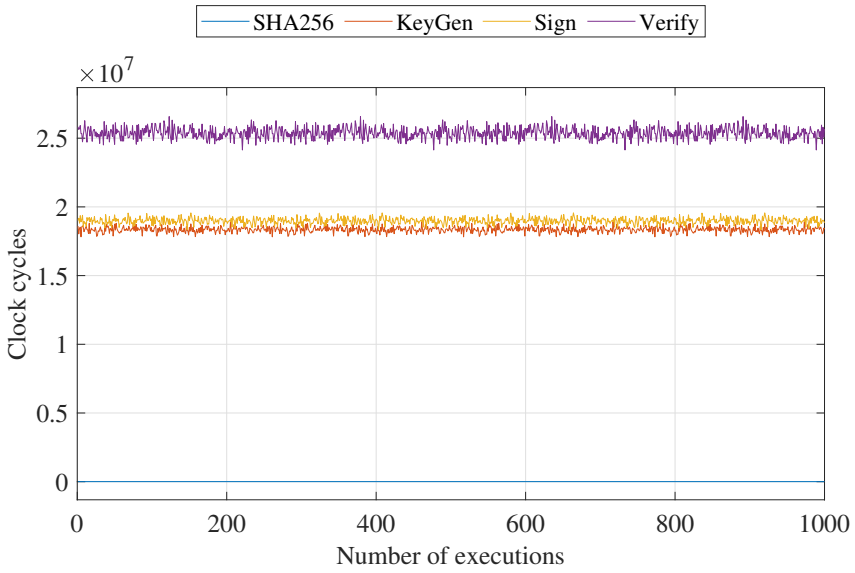
Tabella 5: Confronto tra i parametri ottenuti dal benchmark delle funzioni di X-CUBE-CRYPTOLIB e quelle della libreria uECC per il Test *Random message, fixed key* su un totale di 1000 misurazioni effettuate

Dal confronto dei risultati ottenuti con X-CUBE-CRYPTOLIB, del tutto simili a quelli della sezione 5.1.2, e con uECC, si possono trarre le seguenti conclusioni:

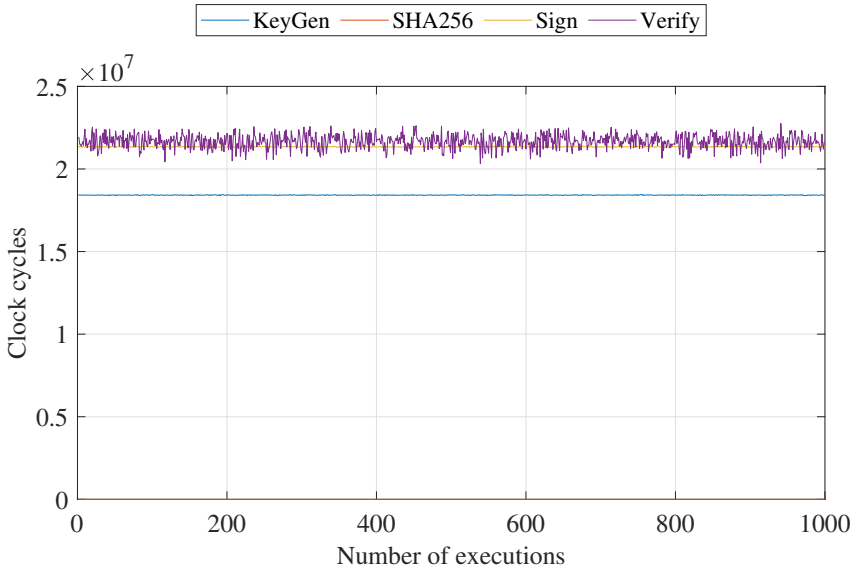
- l'algoritmo di hashing produce dei risultati molto simili a quelli ottenuti nel precedente test;
- l'algoritmo di generazione della coppia di chiavi implementato da X-CUBE-CRYPTO è più veloce, oltre ad essere più lineare e costante tra una misurazione e l'altra;
- il numero di cicli di clock impiegati per la firma digitale è minore quando si utilizza la funzione della libreria X-CUBE-CRYPTO; tuttavia, questo numero è molto più variabile tra le diverse misure;
- il numero di cicli di clock impiegati per la verifica della firma è minore in uECC, che presenta anche una deviazione standard minore.

5.2.3 Test 3- Random message, random key

Le ultime 1000 misurazioni sono state effettuate usando messaggi e chiavi private che cambiano ogni volta; entrambi i parametri vengono prelevati dal file binario generato in Matlab contenente i numeri random; di conseguenza, ad ogni iterazione, messaggio e chiave privata sono uguali tra loro poiché vengono presi gli stessi 32 byte, dopodiché alla successiva iterazione assumono il valore dei 32 byte successivi e così via. Anche qui è stato implementato lo stesso codice utilizzato nella sezione 5.2.1 per la generazione sia del messaggio da firmare sia della chiave privata.
I valori numerici e i grafici ottenuti dall'analisi del numero di cicli richiesti per le diverse operazioni crittografiche sono illustrati nella tabella 6 e nelle figure 33, 34, 35, 36 e 37.

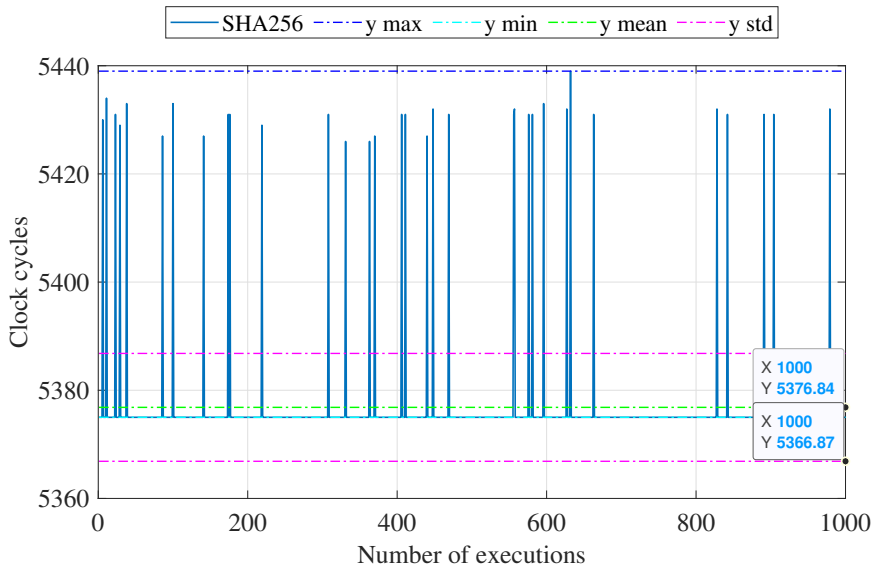


(a)

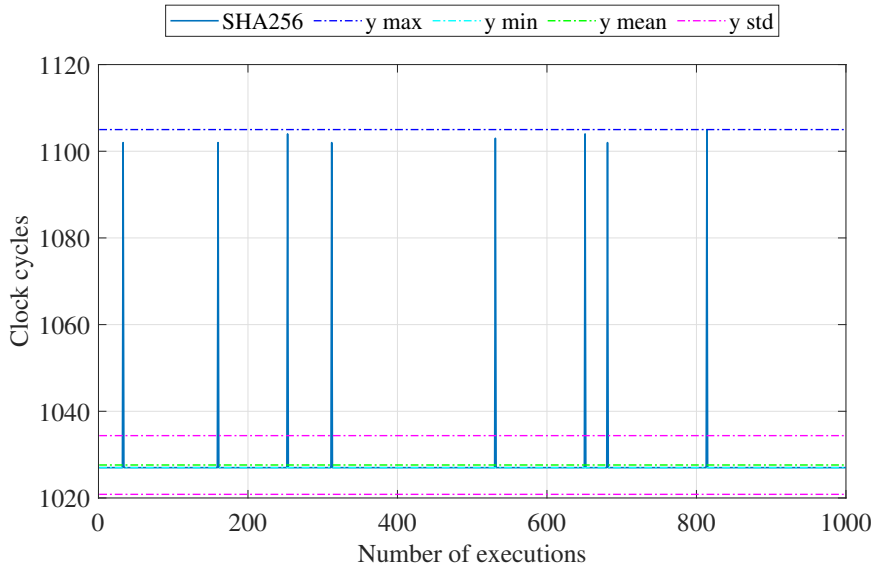


(b)

Figura 33: *Andamento del numero dei cicli di clock per tutte e 4 le operazioni:(a) per le funzioni di X-CUBE-CRYPTOLIB e (b) per le funzioni di uECC*

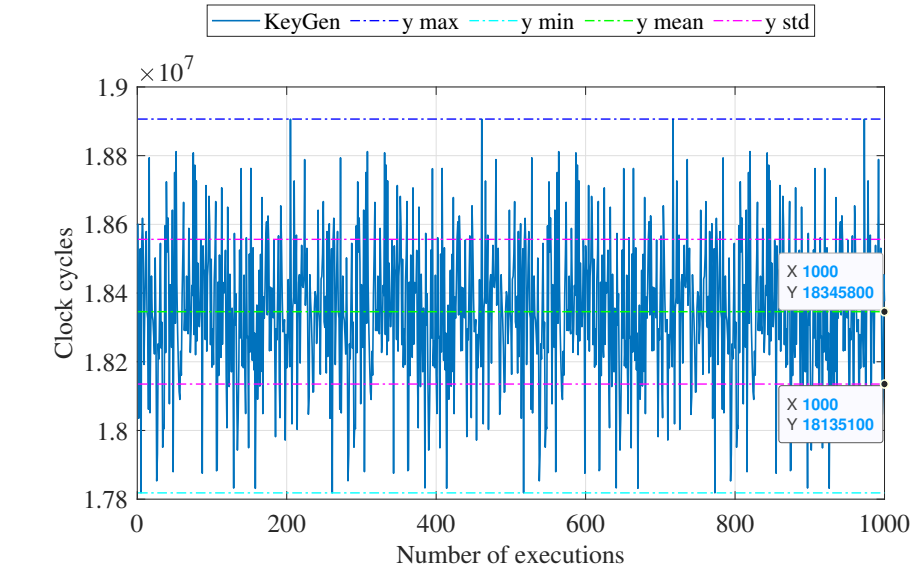


(a)

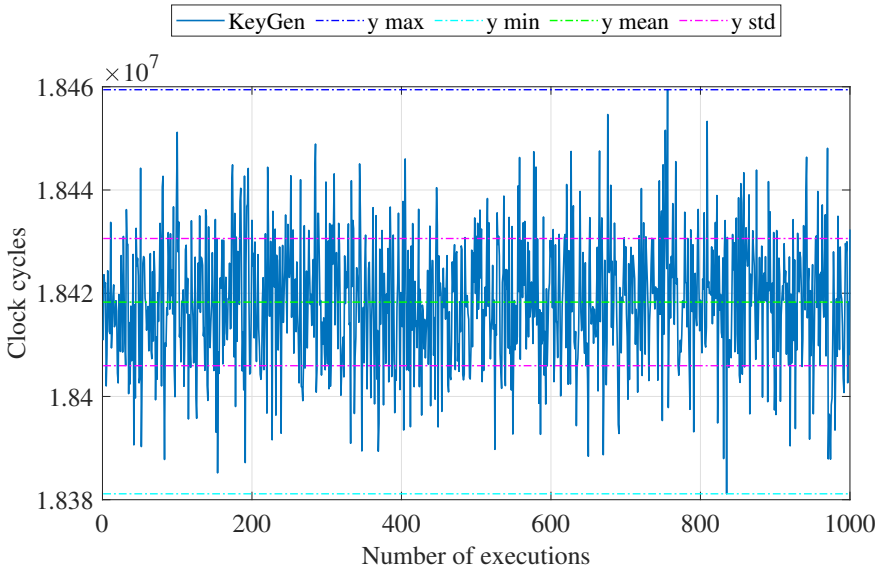


(b)

Figura 34: *Andamento del numero dei cicli di clock per l'esecuzione della funzione SHA256: (a) per le funzioni di X-CUBE-CRYPTOLIB e (b) per le funzioni di uECC*

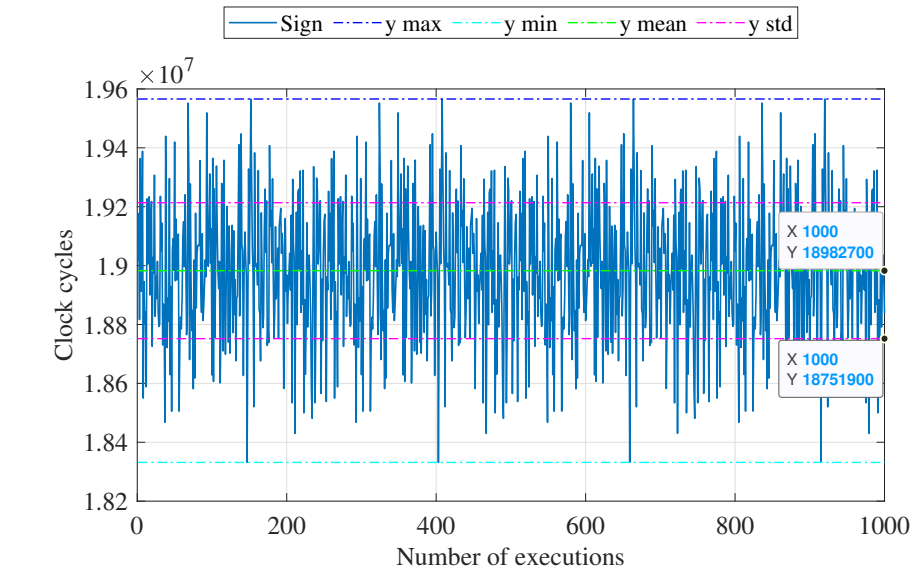


(a)

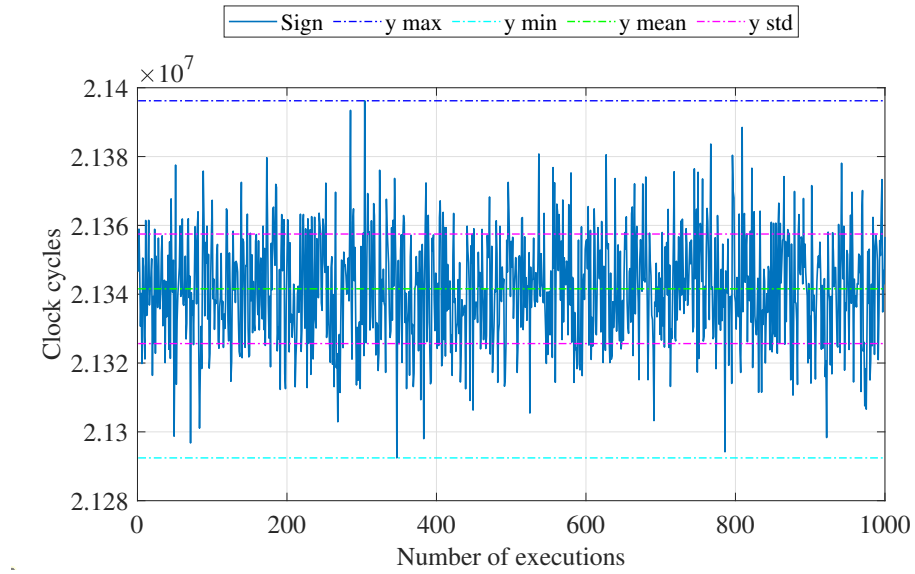


(b)

Figura 35: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la generazione della coppia di chiavi: (a) per le funzioni di X-CUBE-CRYPTOLIB e (b) per le funzioni di uECC*

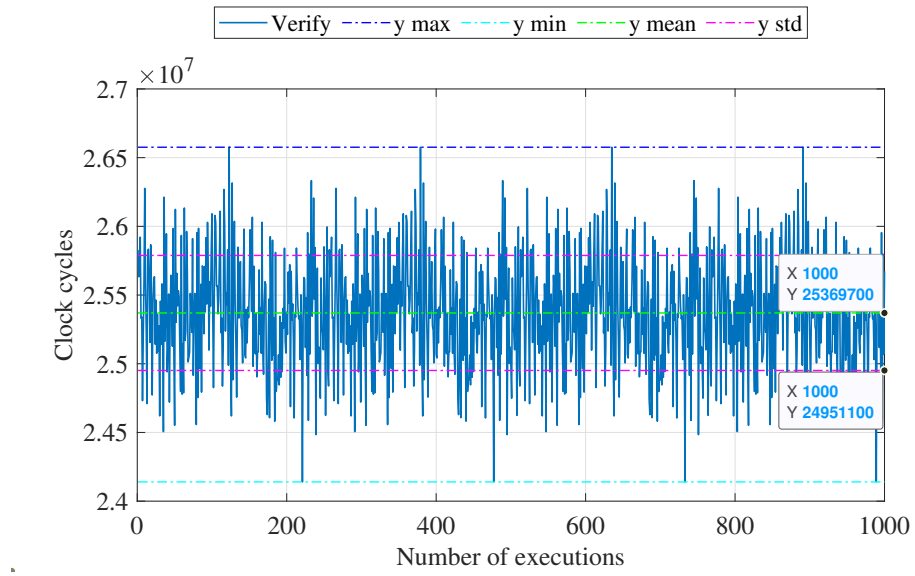


(a)

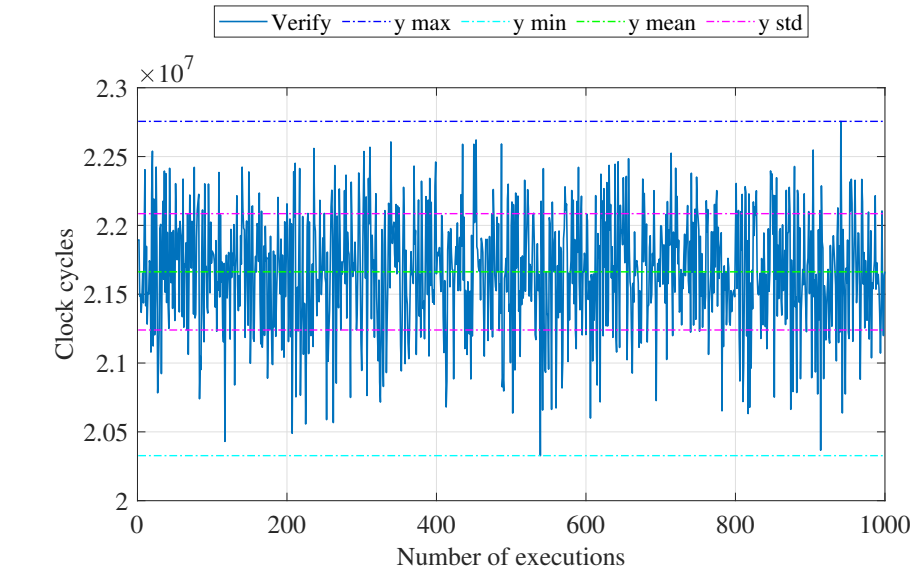


(b)

Figura 36: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la firma: (a) per le funzioni di X-CUBE-CRYPTOLIB e (b) per le funzioni di uECC*



(a)



(b)

Figura 37: *Andamento del numero dei cicli di clock per l'esecuzione della funzione per la verifica: (a) per le funzioni di X-CUBE-CRYPTOLIB e (b) per le funzioni di uECC*

Funzione		Valore minimo	Valore massimo	Media	Deviazione standard
SHA256	Cryptolib uECC lib	5387 1027	5456 1105	5389 1028	11.603 6.775
KeyGen	Cryptolib uECC	$1743 \cdot 10^4$ $1838 \cdot 10^4$	$1912 \cdot 10^4$ $1846 \cdot 10^4$	$1831 \cdot 10^4$ $1842 \cdot 10^4$	$22.754 \cdot 10^4$ $1.232 \cdot 10^4$
Sign	Cryptolib uECC	$1800 \cdot 10^4$ $2129 \cdot 10^4$	$1964 \cdot 10^4$ $2140 \cdot 10^4$	$1895 \cdot 10^4$ $2134 \cdot 10^4$	$25.425 \cdot 10^4$ $1.592 \cdot 10^4$
Verify	Cryptolib uECC	$2390 \cdot 10^4$ $2033 \cdot 10^4$	$2664 \cdot 10^4$ $2276 \cdot 10^4$	$2528 \cdot 10^4$ $2166 \cdot 10^4$	$41.158 \cdot 10^4$ $42.264 \cdot 10^4$

Tabella 6: Confronto tra i parametri ottenuti dal benchmark delle funzioni di X-CUBE-CRYPTOLIB e quelle della libreria uECC per il Test *Random message, random key* su un totale di 1000 misurazioni effettuate

Anche in quest'ultima analisi si possono fare delle considerazioni circa le differenze tra le due librerie:

- esattamente come nei due casi precedenti, la funzione di hashing produce dei risultati molto simili, anche se la funzione della libreria uECC risulta essere poco più rapida;
- la generazione della coppia di chiavi avviene con circa lo stesso numero medio di cicli per entrambe le funzioni delle due librerie; tuttavia, la deviazione standard del numero di cicli relativo alla funzione di X-CUBE-CRYPTO è molto maggiore di quella di uECC, per cui quest'ultimo algoritmo risulta avere un andamento più lineare
- la generazione della firma digitale produce un minor numero di cicli di esecuzione per quanto riguarda la libreria X-CUBE-CRYPTOLIB, tuttavia, anche qui, l'andamento è molto più variabile rispetto a quello di uECC;
- l'algoritmo di verifica della firma viene implementato con meno cicli di clock usando l'algoritmo uECC; tuttavia la sua deviazione standard è maggiore, anche se di poco, rispetto a quella che si riscontra usando la relativa funzione di X-CUBE-CRYPTO.

6 Conclusioni

La realizzazione di questo lavoro di tesi ha permesso di approfondire più nel dettaglio alcuni aspetti legati alla tecnologia blockchain; nello specifico, è stato possibile analizzare alcuni elementi alla base della crittografia basata sulle curve ellittiche, come la curva secp256k1, ed eseguire le principali operazioni crittografiche su una scheda embedded.
La piattaforma blockchain che è stata presa in esame è Ethereum, per cui sono state esaminate tutte le sue caratteristiche e peculiarità allo scopo di implementare su un microcontrollore STM32 un nodo leggero che fosse compatibile con gli standard della rete Ethereum. Il nodo così realizzato, essendo un nodo light, non è in grado di effettuare il mining e partecipare attivamente alle attività di validazione dei blocchi di transazioni sul network, tuttavia è in grado di provvedere in maniera autonoma alla generazione di transazioni che rispettano lo standard delle transazioni Ethereum. Il nodo in esame è quindi in grado di generare una coppia di chiavi a partire, a seconda della situazione, da una chiave privata random, pseudorandom o fornita dall'utente stesso; è inoltre in grado di creare un payload grezzo contenente tutti i dati che si vuole trasmettere a un altro nodo Ethereum, di firmare questo pacchetto di dati e provvedere alla verifica di validità dello stesso; infine, è capace di implementare e inviare la transazione completa a un altro nodo della rete.
Dopo aver ottenuto un certo numero di transazioni e dopo aver verificato la loro validità sia attraverso un tool Ethereum sia attraverso un nodo smart della rete, sono state prese in esame le prestazioni del microcontrollore nel momento in cui esegue le principali operazioni di crittografia come la generazione dell'hash, della coppia di chiavi, la firma del messaggio e la sua verifica. Si è potuto constatare come, in generale, le funzioni di hashing siano le più rapide, mentre l'operazione di verifica risulti quella che richiede un maggior numero di cicli di clock e, dunque, un maggior tempo di esecuzione; i tempi per la generazione della coppia di chiavi e della firma, invece, dipendono principalmente dalla chiave privata, in particolare se, per la generazione di un certo numero di transazioni, essa rimane fissa o se viene cambiata di volta in volta ad ogni esecuzione in maniera pseudorandom.
Infine sono state messe a confronto le prestazioni, in termini di tempi di esecuzione, delle funzioni crittografiche di due librerie, X-CUBE-CRYPTOLIB e MicroECC; è stato possibile osservare come le funzioni della libreria presa in esame per questo lavoro di tesi, ossia X-CUBE-CRYPTOLIB, siano in linea generale più lente e più variabili rispetto alle corrispondenti implementazioni della libreria MicroECC, tranne nel caso della generazione della coppia di chiavi quando la chiave privata rimane la medesima per tutte le transazioni.
Pur svolgendo quanto richiesto, è chiaro che in questo lavoro di tesi possano essere apportati dei miglioramenti che potrebbero essere introdotti in possibili lavori futuri: in primo luogo, può essere utile l'implementazione di una libreria ausiliaria per il corretto calcolo del parametro v o, per lo meno, per il recupero della coordinata y derivante dalla moltiplicazione tra il numero casuale k e il punto generatore della curva ellittica utilizzata in fase di firma della transazione. Un altro lavoro futuro potrebbe riguardare l'interconnessione tra il nodo light descritto in questo elaborato di tesi e un altro nodo smart della stessa rete, in modo da rendere possibile uno scambio bidirezionale dei dati tra i due nodi.

Riferimenti bibliografici

[1] M. Crnjac, I. Veža, N. Banduka, “From concept to the introduction of Industry 4.0”, 2017.

[2] N. Teslya, I. Ryabchikov “Blockchain-based platform architecture for industrial IoT”, 2017.

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

[4] Karl Wüst, Arthur Gervais, "Do you need a blockchain?", 2017.

[5] Andreas Antonopoulos, "Mastering Bitcoin", 2010.

[6] https://ethereum.org/en/developers/docs/nodes-and-clients/.

[7] V. Buterin, "On Public and Private Blockchains", 2015. Consultabile al sito https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/.

[8] https://developer.bitcoin.org/devguide/block_chain.html.

[9] https://blog.ethereum.org/2021/01/20/the-state-of-eth2-january-2021/.

[10] A. Mushtaq, I. Ul Haq, “Implication of Blockchain in Industry 4.0”, 2019.

[11] https://ethdocs.org/en/latest/contracts-and-transactions/.

[12] https://cryptobook.nakov.com/digital-signatures/ecdsa-sign-verify-messages.

[13] https://ethereum.stackexchange.com/questions/3542/how-are-ethereum-addresses-generated.

[14] D. Mahto, D. K. Yadav, "RSA and ECC: A comparative analysis", 2017. Consultabile al sito https://www.ripublication.com/ijaer17/ijaerv12n19_140.pdf.

[15] https://github.com/ethereumbook/ethereumbook/blob/develop/book.asciidoc.

[16] https://medium.com/coinmonks/ecdsa-the-art-of-cryptographic-signatures-d0bb254c8b96.

[17] https://eth.wiki/fundamentals/rlp.

[18] https://www.st.com/resource/en/datasheet/stm32f439zi.pdf.

[19] https://www.st.com/resource/en/user_manual/cd00208802-stm32-cryptographic-library-stmicroelectronics.pdf.

[20] https://github.com/firefly/wallet/tree/master/source/libs/ethers/src.

[21] https://github.com/KingHodor/Ethereum-RLP.

[22] https://www.secg.org/SEC2-Ver-1.0.pdf.

[23] K. Bjoernsen, "Koblitz Curves and its practical uses in Bitcoin security", 2015.

[24] A. Antipa, D. Brown, A. Menezes, R. Struik, S. Vanstone, "Validation of Elliptic Curve Public Keys", 2003.

[25] https://besu.hyperledger.org/en/1.3.5/Concepts/NetworkID-And-ChainID/.

[26] https://medium.com/mycrypto/the-magic-of-digital-signatures-on-ethereum-98fe184dc9c7.

[27] https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md.