SAPIENZA
UNIVERSITÀ DI ROMA

TEE-based execution of Smart Contract

Facoltà Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea in Informatica

Candidato

Alexandru Andrei Colacel
Matricola 1941345

Relatore

Mattia Samory

Relatore Esterno

Prof. Claudio Di Ciccio,
Utrecht University (Netherlands)

Correlatori

Dott. Valerio Goretti
Dott. Davide Basile

Anno Accademico 2023/2024

TEE-based execution of Smart Contract
Tesi di Laurea. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: colacel.1941345@studenti.uniroma1.it

A mia sorella...

Indice

1 Introduzione **3**

2 Background **5**
2.1 blockchain 5
2.1.1 Punti di Forza della blockchain 5
2.1.2 Trilemma della blockchain 6
2.1.3 Concetti chiave delle blockchain Ethereum 9
2.1.4 Macchina a stati in Ethereum 11
2.1.5 Smart Contract: Fondamenti, Implementazione e Implicazioni 12
2.2 Trusted Execution Environment 19
2.2.1 Come viene misurata la fiducia 19
2.2.2 Fiducia nelle TEEs 20
2.2.3 Il ruolo delle TEEs nella sicurezza 21
2.2.4 Implementazioni di una TEE 21
2.2.5 Configurazioni di Architetture di una TEE 22
2.2.6 Nascita di Trusted Execution Technology e Intel SGX 22
2.3 Esecuzione degli Smart Contracts nelle TEEs 25
2.3.1 Vantaggi in termini di sicurezza e efficienza 25
2.3.2 Sicurezza 25
2.3.3 Efficienza 27
2.3.4 Meccanismi per garantire l'integrità e la trasparenza 27

3 Esempio di utilizzo in campo sanitario **29**

4 Progettazione e Realizzazione **32**
4.1 Sviluppo su Ethereum con Geth e Clef 32
4.1.1 Avvio di Clef e Geth 32
4.1.2 Recupero di Ether dalle Testnet 33
4.1.3 Interazione con Geth 33
4.1.4 Testnet 33
4.2 Implementazione 33
4.2.1 Conversione del codice 34
4.2.2 Ego 34
4.2.3 Codice Sorgente Solidity 35

5 Discussione e prospettive future **55**
5.0.1 Discussione 55
5.0.2 Limitazioni e Lavori Futuri 56

Bibliografia **57**

Sommario

La blockchain rappresenta una tecnologia che ha rivoluzionato il modo in cui vengono registrate e gestite le transazioni digitali. Si tratta di registri digitali distribuiti, immutabili e altamente sicuri, che garantiscono la trasparenza delle transazioni e resistono a qualsiasi tentativo di manipolazione.

Gli Smart Contract, scritti in linguaggi di programmazione come Solidity per la blockchain Ethereum, o Plutus su blockchain Cardano, sono programmi autoeseguibili che automatizzano e gestiscono transazioni in base a condizioni predefinite.

Questa tesi si concentra sull'esplorazione di come le blockchain consentano l'esecuzione affidabile degli Smart Contracts. Saranno analizzati i principi fondamentali delle blockchain, tra cui la decentralizzazione e la sicurezza, con un focus particolare sull'esecuzione degli Smart Contracts all'interno delle Trusted Execution Environments (TEEs).

Attraverso l'analisi di casi studio e una revisione della letteratura, questo lavoro cerca di gettare luce sui vantaggi e le sfide dell'esecuzione degli Smart Contracts sulla blockchain. Infine, vengono delineate alcune prospettive future per l'evoluzione di questa tecnologia, aprendo la strada a un'ulteriore integrazione degli Smart Contracts nella vita quotidiana e nei settori aziendali.

Struttura relazione

Durante il percorso di ricerca intrapreso in questo lavoro, sono state esplorate una vasta gamma di argomenti pertinenti nel contesto di questa tecnologia. Seguendo un approccio metodico, abbiamo suddiviso il lavoro in cinque capitoli principali che delineano le diverse aree di indagine.

Il Capitolo 1 fornisce un resoconto dettagliato del tirocinio e delle attività svolte durante il periodo di studio. Vengono esaminate le sfide affrontate, le metodologie adottate e gli obiettivi stabiliti per il lavoro.

Il Capitolo 2 approfondisce i concetti fondamentali della blockchain, con particolare enfasi sulla blockchain Ethereum e sugli Smart Contract. Si esamina inoltre il ruolo delle TEEs nella sicurezza delle operazioni blockchain e si discute l'esecuzione degli Smart Contract all'interno di esse, mettendo in luce vantaggi e sfide tecniche. Si presta inoltre attenzione alla critica questione della verificabilità e dell'integrità nell'esecuzione degli Smart Contract.

Nel Capitolo 3 viene presentato uno scenario reale nell'ambito sanitario. Vengono inoltre esaminati i vari problemi presenti nel processo di acquisizione e utilizzo dei dati relativi al periodo di degenza pazienti.

Il Capitolo 4 è dedicato alla descrizione dei metodi sperimentali impiegati e delle simulazioni condotte durante il corso della ricerca. Vengono analizzati i risultati ottenuti e se ne discutono le implicazioni.

Infine, nel Capitolo 5, vengono presentate le principali conclusioni emerse dalla ricerca e si prospettano possibili direzioni future per lo sviluppo della tecnologia, individuando potenziali aree di ricerca e di approfondimento.

Capitolo 1

Introduzione

Questa tesi si propone di affrontare una sfida cruciale nell'ambito delle blockchain: l'esecuzione efficiente e sicura degli Smart Contracts (SCs) all'interno delle TEEs. Gli Smart Contracts costituiscono il nucleo delle blockchain, ma la loro esecuzione pubblica presenta sfide come la trasparenza e i costi computazionali. Le TEEs vengono perciò impiegati nella loro esecuzione per garantire un ambiente sicuro e isolato, proteggendo sia il codice degli Smart Contracts che i dati sensibili associati. La loro capacità di offrire una sicurezza robusta, verificabilità dell'esecuzione e protezione contro vulnerabilità esterne li rende una scelta cruciale per migliorare l'affidabilità e la sicurezza delle transazioni nelle blockchain. Questo lavoro di ricerca si concentra su un approccio innovativo per affrontare tali problematiche. Il fulcro del progetto perciò è l'esecuzione degli Smart Contracts nelle TEEs, richiedendo una stretta interazione con l'ecosistema blockchain per avviare e concludere le computazioni off-chain in modo affidabile.

Uno degli obiettivi principali è garantire la verificabilità e l'integrità dell'esecuzione del codice degli Smart Contracts al di fuori della blockchain, permettendo loro di essere eseguiti senza rivelare i dettagli del loro funzionamento interno o i dati sensibili a terze parti non autorizzati.

Questo lavoro mira a studiare l'esecuzione degli Smart Contracts all'interno delle TEEs, un'iniziativa volta a garantire la tutela della privacy e della sicurezza nell'ambito delle transazioni blockchain. Le ragioni di questo approccio risiedono nella crescente consapevolezza delle implicazioni etiche e legali [26] associate all'utilizzo delle tecnologie blockchain, dove l'emergere di questioni sensibili come la privacy e la sicurezza dei dati ha richiesto l'adozione di soluzioni innovative e affidabili.

Il contesto digitale attuale evidenzia una crescente necessità di preservare la riservatezza delle informazioni personali e finanziarie degli individui, assicurando la conformità alle normative sulla protezione dei dati, come il GDPR [1] europeo, oltre a garantire l'integrità e l'autenticità delle transazioni eseguite sulla blockchain. In questo scenario, l'implementazione delle Trusted Execution Environments, come Intel SGX, si è rivelata una soluzione promettente, in grado di fornire un ambiente protetto e isolato all'interno del quale eseguire Smart Contracts in modo sicuro e affidabile.

Questo periodo si è pertanto incentrato sull'analisi e lo sviluppo di metodi per trasferire l'esecuzione degli Smart Contract da Solidity a linguaggi come Go, con l'obiettivo di renderli adatti all'esecuzione all'interno delle TEEs. Questa strategia

[1] `https://commission.europa.eu/law/law-topic/data-protection_en`

mirava a garantire la protezione dei dati sensibili e la confidenzialità delle transazioni eseguite sulla blockchain, offrendo un livello aggiuntivo di sicurezza e affidabilità rispetto alle implementazioni tradizionali.

Nel percorso di tirocinio, si sono affrontate sfide significative legate alla comprensione dei meccanismi di funzionamento delle blockchain e all'implementazione di soluzioni tecniche per garantire la compatibilità e l'integrazione degli Smart Contracts con gli ambienti di esecuzione protetti. Tale esperienza ha contribuito ad arricchire le mie competenze nel campo dello sviluppo blockchain e della sicurezza informatica, offrendo un quadro completo delle sfide e delle opportunità associate all'adozione di tecnologie innovative per la gestione e l'esecuzione di transazioni digitali.

Capitolo 2

Background

Questo capitolo darà luogo ad un'analisi esaustiva dei concetti fondamentali inerenti alla blockchain, tracciandone l'evoluzione storica e delineando le nozioni cardine. Successivamente, si procederà con un'esposizione introduttiva relativa alle TEEs, con particolare attenzione rivolta alla sinergia e all'ottimizzazione dell'implementazione degli Smart Contract all'interno di tale contesto.

2.1 blockchain

La blockchain, nella sua forma più rudimentale, è stata concepita da Stuart Haber e W. Scott Stornetta nel 1991. Essi hanno proposto una struttura crittograficamente sicura a catena per documenti digitali, mirando a rendere i timestamp dei documenti resistente alla manomissione [25]. Successivamente, nel 1992, Bayer, Haber e Stornetta hanno introdotto gli alberi di Merkle, migliorando notevolmente l'efficienza e l'affidabilità della tecnologia [6].

Il punto di svolta per la blockchain è arrivato con il lancio di Bitcoin nel 2008, ideato da un individuo o gruppo sotto lo pseudonimo di Satoshi Nakamoto. Bitcoin ha introdotto un sistema di proof-of-work per gestire un ledger distribuito in modo decentralizzato, risolvendo il problema del doppio-spese in un ambiente senza un'autorità centrale [37].

2.1.1 Punti di Forza della blockchain

La blockchain offre diversi vantaggi significativi rispetto ai sistemi tradizionali. Primo fra tutti, opera senza un'autorità centrale [2], riducendo il rischio di fallimento centralizzato e aumentando la resistenza alla censura [39], un concetto fondamentale per la decentralizzazione. Inoltre, offre trasparenza [40] e immutabilità [28]; ogni transazione effettuata sulla blockchain è tracciabile e permanente, rendendo quasi impossibile la falsificazione o la manipolazione retroattiva dei dati [45].

Dal punto di vista della sicurezza, la blockchain utilizza crittografia avanzata, garantendo la protezione delle informazioni e la sicurezza delle transazioni. Infine, contribuisce all'efficienza e alla riduzione dei costi nei processi aziendali. La natura decentralizzata e automatizzata della blockchain riduce la necessità di intermediari, potenzialmente abbassando i costi di transazione e semplificando le operazioni complesse.

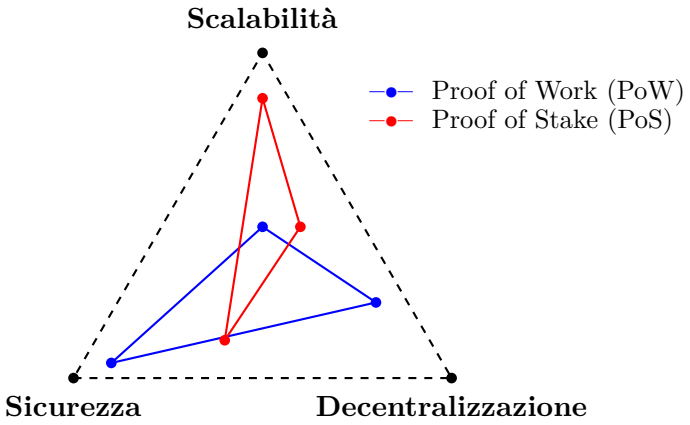

Figura 2.1. Trilemma della blockchain PoW vs PoS.

2.1.2 Trilemma della blockchain

La blockchain, una tecnologia innovativa che ha rivoluzionato numerosi settori, si trova ad affrontare una sfida cruciale conosciuta come il Trilemma della blockchain. Questo concetto teorico suggerisce che una blockchain non può massimizzare simultaneamente tre attributi fondamentali: scalabilità, sicurezza e decentralizzazione. In altre parole, l'aumento di uno di questi attributi può avvenire solo a discapito degli altri due.

La scalabilità rappresenta la capacità di una blockchain di gestire un grande volume di transazioni in modo rapido ed efficiente. Tuttavia, aumentare la scalabilità spesso comporta compromessi sulla sicurezza e sulla decentralizzazione. Una blockchain altamente scalabile potrebbe sacrificare la decentralizzazione trasformandosi in una rete centralizzata controllata da pochi nodi.

La sicurezza è un altro aspetto critico della blockchain, essenziale per garantire che le transazioni siano immutabili e protette da attacchi malevoli. I protocolli di consenso come Proof of Work (PoW) e Proof of Stake (PoS) giocano un ruolo fondamentale nella sicurezza della blockchain [44]. PoW richiede ai miner di risolvere complessi problemi crittografici per validare i blocchi, mentre PoS assegna il diritto di validazione dei blocchi in base alla quantità di criptovaluta posseduta.

La decentralizzazione, infine, si riferisce alla distribuzione del potere decisionale e del controllo su una blockchain tra un gran numero di partecipanti. Una blockchain decentralizzata è resistente alla censura e all'interferenza esterna, ma può essere meno efficiente e scalabile rispetto a una rete centralizzata.

La differenza tra PoW e PoS rappresentata nella fig. 2.1 risiede principalmente nel modo in cui vengono selezionati i validatori dei blocchi e nella sicurezza della rete. PoW richiede un'enorme potenza computazionale per risolvere i complessi problemi crittografici, rendendo la rete più sicura ma meno efficiente in termini di consumo energetico. PoS, d'altra parte, si basa sulla partecipazione economica, consentendo ai possessori di criptovalute di validare i blocchi in base alla loro partecipazione nella rete. Anche se PoS è considerato più efficiente ed ecologico rispetto a PoW, potrebbe essere più vulnerabile a certi tipi di attacchi.

Le chiavi nella blockchain giocano un ruolo cruciale nel garantire la sicurezza, l'integrità e l'accesso ai dati e alle transazioni all'interno di questo registro distribuito e immutabile. Nell'ambiente della blockchain, le chiavi vengono utilizzate per

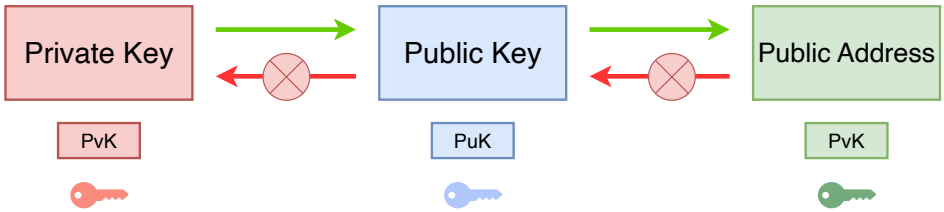

Figura 2.2. La figura illustra i vari passaggi tra Private Key, Public Key e Public Address.

fornire un'identità digitale agli utenti e per firmare digitalmente le transazioni, confermandone l'autenticità e la provenienza [3].

Quando un utente desidera inviare una transazione sulla blockchain, utilizza la sua chiave privata per firmare digitalmente la transazione. Questa firma viene poi verificata dagli altri partecipanti alla rete blockchain utilizzando la chiave pubblica corrispondente, confermando che la transazione è stata autorizzata dal legittimo proprietario della chiave privata.

L'uso corretto delle chiavi nella blockchain è fondamentale per garantire la sicurezza dei fondi e delle informazioni registrate sulla catena di blocchi. Una gestione accurata delle chiavi private è essenziale per prevenire accessi non autorizzati e frodi all'interno del sistema blockchain. Inoltre, la crittografia a chiave pubblica utilizzata nella blockchain [33] offre un alto livello di sicurezza, consentendo a utenti sconosciuti di interagire in modo sicuro e affidabile senza dover rivelare informazioni sensibili come le proprie chiavi private.

La Figura 2.2 illustra il processo di generazione di una chiave pubblica e un indirizzo pubblico a partire da una chiave privata. Le frecce verdi indicano il flusso di informazioni consentito dalla chiave privata alla chiave pubblica, e dalla chiave pubblica all'indirizzo pubblico. Le frecce rosse con una linea attraverso di esse indicano che non è possibile risalire dalla chiave pubblica alla chiave privata o dall'indirizzo pubblico alla chiave pubblica. Questo è un principio fondamentale della crittografia a chiave pubblica.

Le chiavi private costituiscono un componente cruciale all'interno delle infrastrutture blockchain, fungendo da meccanismo di autenticazione e di firma digitale per le transazioni. In conformità con la definizione fornita da Zamani et al. [49], le chiavi private sono stringhe di dati segrete utilizzate per firmare digitalmente le transazioni e garantire l'accesso agli account blockchain. La loro confidenzialità è essenziale, in quanto la loro detenzione conferisce il pieno controllo sugli asset e gli account associati [49]. Tale controllo è garantito attraverso la crittografia a chiave privata, la quale implica la generazione casuale di una sequenza di caratteri alfanumerici che costituiscono la chiave stessa.

Chiavi Pubbliche e Private nelle blockchain

La sicurezza delle chiavi private è prioritaria poiché la loro perdita o esposizione può risultare nella compromissione dell'accesso agli asset contenuti nella blockchain. La generazione delle chiavi private è generalmente affidata all'Algoritmo di Firma Digitale a Curva Ellittica [31], il quale garantisce un elevato grado di sicurezza crittografica [49]. Parallelamente, la chiave pubblica, che ha una visibilità pubblica

all'interno della rete blockchain, viene utilizzata per convalidare le transazioni e generare gli indirizzi blockchain, essenziali per l'interazione all'interno della rete. Tali indirizzi, ottenuti mediante funzioni di hash della chiave pubblica, servono come punti di destinazione per la ricezione di criptovalute e altri asset digitali [49].

L'impiego delle chiavi pubbliche e private all'interno delle blockchain rappresenta un pilastro fondamentale della sicurezza e dell'autenticazione degli utenti, garantendo la validità e l'integrità delle operazioni all'interno di questo contesto decentralizzato. La chiave pubblica, essendo pubblicamente nota, è utilizzata per autorizzare le transazioni e verificare l'identità degli utenti, contribuendo così alla validità e all'affidabilità delle operazioni effettuate sulla blockchain.

L'algoritmo di firma digitale a curva ellittica (ECDSA) [31] è una tecnica crittografica ampiamente utilizzata per garantire l'autenticità e l'integrità dei messaggi nelle comunicazioni digitali. Si basa sull'uso delle proprietà matematiche delle curve ellittiche per generare firme digitali che possono essere verificate in modo efficiente. Il primo passo coinvolge la generazione di una coppia di chiavi pubblica e privata. La chiave privata è un numero casuale selezionato nell'intervallo definito dalla curva ellittica, mentre la chiave pubblica è il punto corrispondente sulla curva ellittica ottenuto moltiplicando la chiave privata per il punto base della curva. Per firmare un messaggio, il mittente utilizza la propria chiave privata per calcolare una firma digitale unica per quel messaggio. Questo coinvolge la generazione di un valore di firma e l'applicazione di una serie di operazioni matematiche che coinvolgono la curva ellittica. Il destinatario del messaggio utilizza la chiave pubblica del mittente per verificare la firma digitale. Questo processo coinvolge il calcolo di una serie di punti sulla curva ellittica e la verifica che la firma corrisponda al messaggio originale.

Rete Bitcoin

Nel contesto di Bitcoin [37], si utilizza comunemente il termine "**registro distribuito**" per descrivere la sua struttura fondamentale. Questo registro distribuito permette la creazione e la gestione di una valuta decentralizzata, utilizzando principi di crittografia per assicurare l'integrità e la sicurezza della rete. Il registro di Bitcoin traccia tutte le transazioni, operando secondo regole ben definite per mantenere la coerenza e la validità delle transazioni.

Un esempio di queste regole è l'impossibilità per un indirizzo Bitcoin di spendere più Bitcoin di quanti ne ha ricevuti [16]. Questi principi sono essenziali per il funzionamento della rete Bitcoin e rappresentano la base per molte altre blockchain.

Il Proof of Work (PoW) è un meccanismo di consenso fondamentale nella rete Bitcoin, che consente di raggiungere un accordo distribuito in modo decentralizzato e sicuro. In questo meccanismo, i membri della rete, noti come "miner" [37], sono tenuti a eseguire un lavoro computazionale impegnativo ma facilmente verificabile dagli altri.

Nel processo di mining di Bitcoin, il lavoro consiste nel calcolare un hash che soddisfi determinati criteri predefiniti. Questo processo coinvolge la generazione di hash a partire dai dati dei nuovi blocchi, includendo le transazioni da registrare nella blockchain. La difficoltà del problema crittografico da risolvere è variabile, consentendo alla rete di mantenere un tempo medio costante per la scoperta di nuovi blocchi, in base alla potenza computazionale complessiva dei minatori.

I nodi validatori che riescono a risolvere il problema e ad aggiungere un nuovo blocco alla blockchain ricevono una ricompensa in Bitcoin e le commissioni delle transazioni. Questo non solo incentiva la manutenzione e l'operatività della rete,

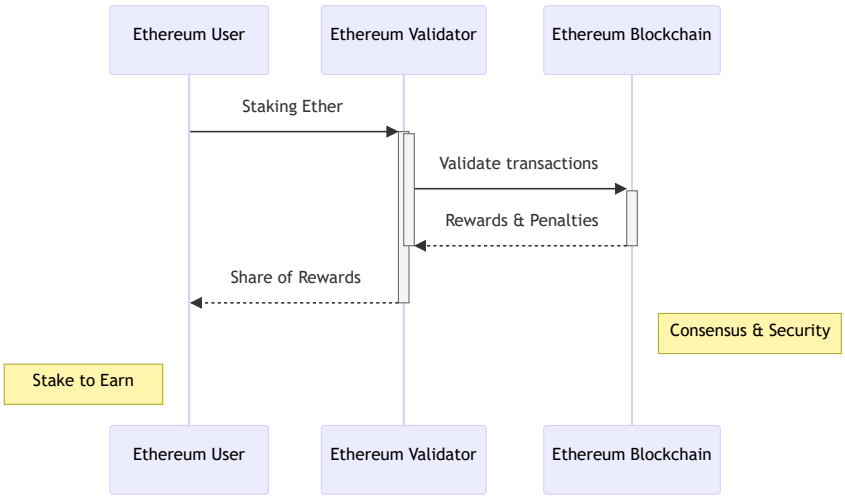

Figura 2.3. Il diagramma illustra il processo del Proof of Stake nella blockchain Ethereum.

ma contribuisce anche alla sicurezza del sistema. Il Proof of Work protegge la rete da attacchi arbitrari, rendendo proibitivo il costo di alterazione della blockchain. Modificare anche un solo blocco richiederebbe un'enorme quantità di potenza di calcolo per ricalcolare il lavoro di PoW per quel blocco e tutti quelli successivi.

Inoltre, il PoW sostiene il principio di decentralizzazione della blockchain di Bitcoin, permettendo a chiunque abbia sufficiente potenza di calcolo di partecipare al mining. Questo aiuta a prevenire il problema del doppio-spese, mantenendo un registro pubblico e cronologico unico di tutte le transazioni. Infine, il processo di mining attraverso il PoW è l'unico metodo di creazione di nuovi Bitcoin, imitando un sistema di estrazione simile a quello delle risorse naturali e limitando l'offerta nel tempo.

2.1.3 Concetti chiave delle blockchain Ethereum

La blockchain **Ethereum**, fondata da Vitalik Buterin nel 2015 [12], introduce concetti rivoluzionari nel panorama delle tecnologie distribuite e delle criptovalute. Ethereum non solo funziona come una blockchain per transazioni digitali, ma agisce anche come una piattaforma per Smart Contracts e applicazioni decentralizzate (DApps) [9]. Questo capitolo approfondirà i concetti principali legati a Ethereum, esplorando le sue funzionalità e il suo impatto nel settore tecnologico e finanziario.

Ethereum, dalla sua nascita, ha adottato il meccanismo di consenso Proof of Work (PoW), simile a quello utilizzato da Bitcoin. In PoW, i minatori impiegano potenza di calcolo per risolvere puzzle crittografici complessi al fine di aggiungere nuovi blocchi alla blockchain, ricevendo in cambio Ether come ricompensa [11]. Tuttavia, questa metodologia è stata oggetto di critiche a causa del suo elevato consumo energetico e impatto ambientale.

Per affrontare queste preoccupazioni, Ethereum ha iniziato una transizione verso il Proof of Stake (PoS) con il suo aggiornamento Ethereum 2.0. Nel PoS, la creazione di nuovi blocchi e la validazione delle transazioni sono basate sulla partecipazione

degli utenti che "bloccano" una parte dei loro Ether come "stake" [22]. I validatori vengono scelti in modo pseudo-casuale per proporre e votare sui blocchi, in base alla quantità di Ether che hanno messo in gioco e per quanto tempo l'hanno fatto. La Figura 2.3 illustra il concetto di PoS nella blockchian Ethereum. Gli utenti Ethereum partecipano al staking di Ether, che attivano i validatori. Questi ultimi validano le transazioni sulla blockchain, ricevendo ricompense o penalità a seconda della loro performance. Dopo la validazione, i validatori condividono le ricompense con gli utenti che hanno effettuato lo staking. Questo processo è cruciale per il consenso e la sicurezza della rete Ethereum. Le barre laterali grigie nel diagramma rappresentano la durata dell'attività degli attori. La barra più grande a destra del validatore indica un periodo di attività prolungato, necessario per la validazione delle transazioni e la distribuzione delle ricompense. La barra più piccola a sinistra dell'utente riflette un coinvolgimento più breve, limitato allo staking di Ether e alla ricezione delle ricompense.

Il passaggio a PoS porta diversi vantaggi significativi. Innanzitutto, si stima una riduzione drastica del consumo energetico [5], rendendo la rete Ethereum più ecologica [14]. Inoltre, il PoS promette di migliorare la scalabilità della rete, facilitando transazioni più rapide e a costi più bassi [23]. Infine, si prevede che il PoS offra una maggiore sicurezza, rendendo teoricamente più costoso e meno vantaggioso per un attaccante tentare di manipolare la blockchain [13].

Rete Ethereum e Ethereum Virtual Machine (EVM)

Ethereum si distingue come una blockchain che supporta una criptovaluta nativa, l'Ether, la quale è fondamentale per l'esecuzione degli Smart Contracts. Per comprendere appieno Ethereum, è necessario considerarla non solo come un registro distribuito, ma come una **macchina di stato distribuita**.

Il concetto di "**stato**" in Ethereum si riferisce a una struttura dati complessa che va oltre il semplice tracciamento dei saldi degli account. Include anche lo stato della macchina, che cambia con ogni blocco aggiunto alla catena, seguendo regole predefinite. Queste regole di transizione di stato sono definite dall'Ethereum Virtual Machine (EVM), che permette l'esecuzione di codice arbitrario sotto forma di Smart Contracts.

La EVM è una componente dell'infrastruttura Ethereum, permettendo non solo di eseguire transazioni, ma anche di implementare una varietà di applicazioni decentralizzate e servizi complessi. Ciò fa di Ethereum una piattaforma più versatile rispetto ad altre blockchain, come Bitcoin, che è principalmente focalizzata sulle transazioni finanziarie. La Figura 2.4 illustra proprio l'Architettura dell'EVM.

EVM e la sua esecuzione

Il documento [47] discute di Ethereum, una piattaforma che implementa un registro di transazioni decentralizzato. Ethereum è progettato per essere una macchina a stati, in grado di supportare applicazioni decentralizzate e anche Smart Contracts. L'autore spiega che Ethereum va oltre il concetto di singola risorsa computazionale decentralizzata, offrendo una pluralità di risorse con stati e codici operativi distinti che possono interagire tra loro tramite un framework di messaggistica.

L'obiettivo principale di Ethereum è facilitare le transazioni tra individui che altrimenti non avrebbero mezzi per fidarsi l'uno dell'altro, a causa di separazioni geografiche, difficoltà di interfacciamento o inefficienze dei sistemi legali esistenti. Ethereum cerca di fornire un sistema in cui le transazioni siano eseguite autono-

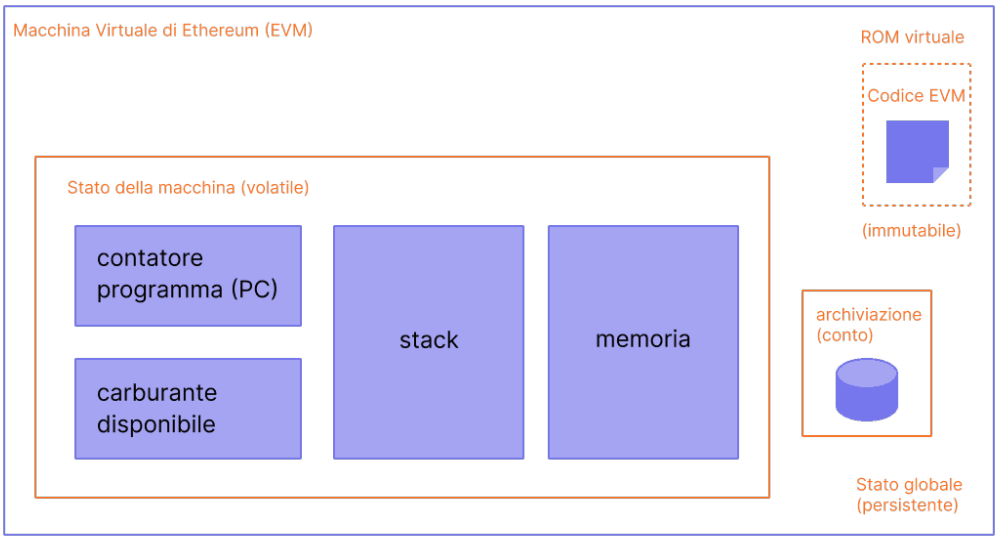

Figura 2.4. Architettura EVM [42]

mamente attraverso un linguaggio chiaro e inequivocabile, garantendo agli utenti fiducia assoluta nei risultati e nelle modalità di realizzazione degli stessi.

Il documento esamina anche il lavoro precedente nel campo delle criptovalute e degli Smart Contracts, tra cui Bitcoin, Namecoin e altre soluzioni basate su blockchain. Si fa riferimento all'utilizzo di algoritmi di proof-of-work per garantire la sicurezza e la validità delle transazioni. Inoltre, vengono menzionati i concetti di incorruttibilità, trasparenza e applicazione di Smart Contracts come strumenti chiave nella costruzione di un sistema fidato.

Wood fornisce un'analisi del paradigma della blockchain e del ruolo che svolge in Ethereum. La blockchain viene descritta come una macchina a stato basata su transazioni, in cui gli stati vengono modificati attraverso transazioni valide. Le transazioni vengono raggruppate in blocchi e concatenati tra loro tramite hash crittografici. Il mining viene introdotto come il processo di dedicare risorse computazionali per garantire la validità dei blocchi e viene spiegato il concetto di proof-of-work come meccanismo per incentivare i partecipanti alla rete.

Infine, il documento accenna all'incentivazione della computazione all'interno della rete Ethereum attraverso l'utilizzo dell'Ether come valuta intrinseca. Viene menzionato il fatto che Ethereum supporta anche la creazione e il tracciamento di token personalizzati attraverso Smart Contracts.

2.1.4 Macchina a stati in Ethereum

La macchina a stati in Ethereum si riferisce al modo in cui vengono gestiti e modificati gli stati all'interno del sistema. Inizialmente, c'è uno stato di partenza chiamato "genesis state" rappresentato nella Figura 2.5, che rappresenta lo stato iniziale del sistema. Successivamente, le transazioni vengono eseguite in modo incrementale per trasformare lo stato corrente in uno stato successivo.

La macchina a stati di Ethereum è basata su un modello chiamato EVM, che è una macchina virtuale che esegue il codice degli Smart Contracts sulla piattaforma

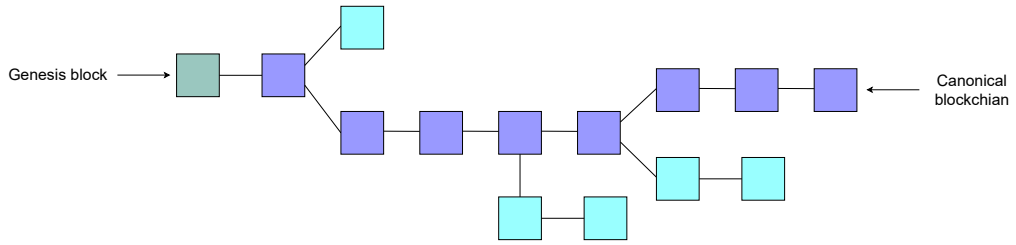

Figura 2.5. Diagramma del Genesis block EVM [42]

Ethereum. L'EVM è progettato per essere uno strato di esecuzione sicuro e isolato all'interno della rete Ethereum.

Gli sviluppatori possono scrivere e distribuire Smart Contracts sulla piattaforma Ethereum utilizzando un linguaggio di programmazione di alto livello come Solidity. Questi Smart Contracts vengono compilati in bytecode che può essere eseguito dall'EVM.

L'EVM interpreta il bytecode e lo esegue su ogni nodo della rete Ethereum in modo coerente, garantendo che tutti i nodi arrivino allo stesso stato finale dopo l'esecuzione di una transazione. Ciò significa che il codice degli Smart Contracts è deterministico e produce risultati consistenti su tutti i nodi della rete.

L'EVM implementa un modello di esecuzione a sandbox, che significa che i Smart Contracts vengono eseguiti in un ambiente isolato dal resto del sistema. Ciò fornisce un livello di sicurezza, in quanto i Smart Contracts non possono interferire direttamente con il funzionamento del sistema sottostante o con altri Smart Contracts.

Inoltre, l'EVM è una macchina virtuale Turing-completa [10], il che significa che può eseguire qualsiasi programma calcolabile. Questa caratteristica consente ai Smart Contracts di implementare logiche complesse e di eseguire calcoli avanzati sulla piattaforma Ethereum.

2.1.5 Smart Contract: Fondamenti, Implementazione e Implicazioni

Gli Smart Contracts

Gli Smart Contract rappresentano un'evoluzione significativa nelle applicazioni della tecnologia blockchain. Creati per automatizzare e garantire l'esecuzione autonoma di accordi digitali, gli Smart Contract hanno il potenziale per rivoluzionare numerosi settori, tra cui finanza, diritto e logistica [21].
Gli Smart Contracts sono programmi autoeseguibili [38] che operano su una blockchain. Sono progettati per automatizzare, facilitare e verificare la negoziazione o l'esecuzione di Smart Contract digitali senza la necessità di intermediari [12].

```
contract Token {
    mapping(address=>uint) balances;
    function deposit() payable {
        // (msg is a global representing
            // the current transaction)

    balances[msg.sender] += msg.value;
}
    function transfer(address recipient,
        uint amount) returns(bool success) {
            if (balances[msg.sender] >= amount) {
                balances[msg.sender] -= amount;
                balances[recipient] += amount;
                return true;
            }
        return false;
    }
}
```

Listing 2.1. Esempio di uno Smart Contract

Questi Smart Contract sono scritti in linguaggi di programmazione specifici e sono immutabili una volta registrati sulla blockchain.

Nel contesto di una transazione su una blockchain come Ethereum, l'utilizzo di uno Smart Contract elimina la necessità di affidarsi a una banca o a un intermediario. In questo scenario, lo Smart Contract contiene istruzioni chiare e regole precise. Una volta che le condizioni specificate nello Smart Contract sono soddisfatte, ad esempio, con il ricevimento di una determinata somma di denaro, il Smart Contract si autoesegue, attuando automaticamente la transazione programmata senza coinvolgere un intermediario.

La decentralizzazione, caratteristica fondamentale di tali Smart Contract, si manifesta nel fatto che essi non risiedono su un server centrale controllato da un'autorità, ma sono distribuiti su una rete di nodi [4]. Ciò garantisce resistenza alla censura e instilla fiducia nella corretta esecuzione dello Smart Contract.

L'autenticità e l'integrità emergono grazie alla registrazione immutabile di tutte le operazioni sulla blockchain. Una volta implementato il Smart Contract, ogni transazione è tracciata in modo trasparente e verificabile, eliminando il rischio di manipolazioni o dispute, poiché tutte le parti coinvolte possono accedere alla stessa versione di verità.

L'aspetto programmabile degli Smart Contract apre la porta a un ampio spettro di applicazioni. È possibile implementare condizioni complesse, gestire dati e agevolare interazioni avanzate tra diversi Smart Contract o applicazioni decentralizzate (DApp). Un esempio tangibile potrebbe essere la gestione automatica del trasferimento di proprietà digitale in base a condizioni predefinite.

La sicurezza è garantita dall'esecuzione su tutti i nodi della rete e dall'immunità al cambiamento del codice una volta implementato. Tale approccio impedisce attacchi mirati o manipolazioni, contribuendo a creare un ambiente affidabile e sicuro. Gli Smart Contract agevolano anche la tokenizzazione e la rappresentazione digitale di asset fisici o astratti [1]. Un esempio concreto è la suddivisione di una proprietà fisica in token, consentendo agli investitori globali di detenere una frazione di quella proprietà senza l'intervento di intermediari complicati.

L'interoperabilità consente a diversi Smart Contract di interagire, creando ecosistemi decentralizzati in cui le applicazioni collaborano per offrire servizi più ampi.

Questa sinergia tra Smart Contracts può portare a soluzioni complesse e innovative.

In accordo con Marchesi et al.(2020)[35] uno Smart Contract deve essere:

- Deterministico: Deve produrre sempre lo stesso output dato lo stesso input, evitando di chiamare funzioni non deterministiche o utilizzare risorse dati non deterministiche.
- Terminabile: Deve poter terminare entro un limite di tempo definito. Ciò può essere garantito utilizzando un timer per interrompere l'esecuzione se supera il limite di tempo, utilizzando catene di blocchi di Turing incomplete che non possono entrare in cicli infiniti o utilizzando un metodo di misurazione dei passaggi per interrompere l'esecuzione dopo un certo numero di passaggi.
- Isolato: Ogni contratto deve essere mantenuto isolato per evitare la corruzione dell'intero sistema in caso di virus o bug.
- Immutabile: Una volta implementato sulla blockchain, un contratto intelligente non può essere modificato, ma può essere disabilitato in modo permanente. Questa proprietà, insieme alla possibilità di pubblicare il codice sorgente del contratto intelligente, garantisce il massimo livello di trasparenza e fiducia.

La figura 2.1 illustra anche un frammento annotato di uno Smart Contract token in Solidity. In dettaglio, questo estratto legge il saldo del mittente dallo stato dello Smart Contract, incrementa il valore della transazione attuale inviata al Smart Contract a questo saldo (creando nuovi token in cambio di Ether inviato al Smart Contract) e memorizza questa nuova somma nell'entrata pertinente della mappatura dei saldi.

Al fine di prevenire esecuzioni infinite dei programmi e garantire una tariffazione equa per tutte le applicazioni client del sistema nei Smart Contracts, il mittente di ogni transazione è tenuto a corrispondere una "tassa" ai staker per la loro interazione con lo Smart Contract sulla blockchain (i stacker sono utenti che sequenziano tali transazioni nei blocchi della blockchain). Questa tassa è proporzionale alla quantità di gas consumato dallo Smart Contract. Lo schema delle tariffe per l'esecuzione è completamente concordato dalla rete, e ogni transazione specifica un importo massimo di gas che è disposta a utilizzare, insieme al tasso di cambio tra Ether e gas. Nel caso in cui una transazione esaurisca il gas durante l'esecuzione, viene annullata, tutti gli aggiornamenti di stato vengono revocati e gli stackers conservano l'intera tariffa di gas associata alla transazione. Ciò impone un limite all'esecuzione di tutte le transazioni EVM, garantendo la terminazione e consentendo alla rete di addebitare le transazioni in base al costo computazionale che comportano.

Transazioni e Gas Fee su Ethereum

Il gas su Ethereum non è un concetto tangibile, ma piuttosto un'unità di misura per quantificare l'energia computazionale necessaria per eseguire operazioni sulla rete. Ogni operazione richiede una quantità specifica di gas, che può essere tradotta in costo in Ether, la criptovaluta nativa di Ethereum.

Il concetto di gas è stato introdotto per risolvere alcuni problemi fondamentali nella progettazione delle blockchain. Una delle sfide principali era garantire che gli attaccanti non sfruttassero la rete eseguendo operazioni computazionalmente onerose senza incorrere in costi proporzionali.

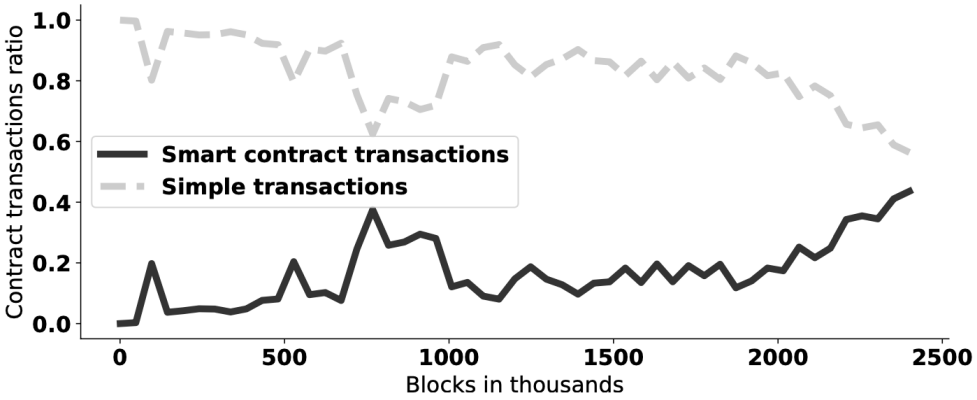

Figura 2.6. Evoluzione temporale del rapporto tra transazioni semplici e transazioni di Smart Contract all'interno di una blockchain.

Quando un utente invia una transazione su Ethereum, questa comporta l'esecuzione di una serie di operazioni sui nodi della rete. Ogni operazione ha un costo in gas, e il totale del gas richiesto per una transazione è chiamato "gas limit". Gli utenti devono specificare il gas limit quando creano e inviano una transazione.

Il "gas price" è un altro elemento chiave. Rappresenta la quantità di Ether che un utente è disposto a pagare per ogni unità di gas. Il gas fee totale, che rappresenta il costo della transazione, è ottenuto moltiplicando il gas limit per il gas price.

Un esempio pratico può aiutare a comprendere meglio il processo. Supponiamo che un utente imposti un gas limit di 100.000 e un gas price di 0,00000001 ETH. Il gas fee totale sarà di 0,001 ETH (100.000 * 0,00000001). Questo importo sarà la ricompensa per gli stackers che confermano la transazione.

Gli stackers o validatori, infatti, giocano un ruolo cruciale nel processo. Essi selezionano le transazioni da includere nei blocchi e vengono ricompensati con le fee di transazione. Un gas price più elevato rende la transazione più attraente per gli stackers, poiché ricevono una ricompensa maggiore.

La scelta del gas price e del gas limit è un bilanciamento delicato. Un gas price troppo basso può causare ritardi nell'inclusione della transazione nei blocchi, mentre un gas limit troppo basso potrebbe far terminare prematuramente l'esecuzione delle operazioni, con il rischio che la transazione non venga confermata.

Ogni operazione all'interno di uno Smart Contract ha un costo associato in termini di gas. La complessità e la quantità di calcoli richiesti influenzano direttamente il gas necessario per eseguire un'operazione.

Gli sviluppatori di Smart Contract devono considerare attentamente la progettazione dei loro Smart Contract per minimizzare il consumo di gas. La riduzione della complessità del codice e l'ottimizzazione degli algoritmi sono cruciali per garantire l'efficienza ed evitare costi eccessivi.

Un aspetto interessante riguarda la gestione degli errori e delle eccezioni negli Smart Contract. Se uno Smart Contract dovesse incontrare un errore durante l'esecuzione, il gas speso fino a quel momento sarà comunque consumato. Pertanto, gli sviluppatori devono scrivere codice robusto per evitare sprechi di gas inutili.

La Figura 2.6 illustra l'evoluzione temporale del rapporto tra transazioni semplici e transazioni dello Smart Contract all'interno della blockchain Ethereum. L'asse orizzontale rappresenta il numero di blocchi in migliaia, servendo come indicatore

temporale. L'asse verticale mostra il rapporto delle transazioni, variando da 0 (nessuna transazione di quel tipo) a 1 (tutte le transazioni di quel tipo). La linea continua rappresenta le transazioni degli Smart Contracts, mostrando una tendenza all'incremento nel tempo, mentre la linea tratteggiata indica le transazioni semplici, che dimostrano una maggiore variabilità e un trend generalmente decrescente. Questo suggerisce un crescente utilizzo degli Smart Contracts rispetto alle transazioni semplici eseguite.[32]

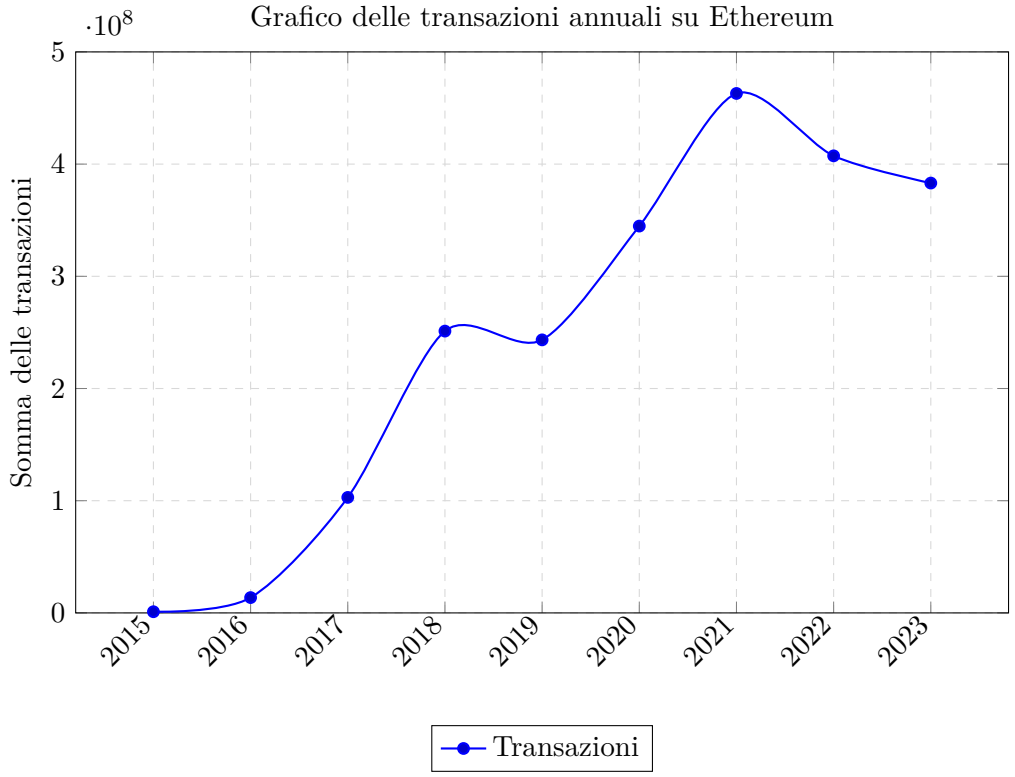

Figura 2.7. Numero di Transazioni eseguite. Fonte [20]

Importanza della Sicurezza e Ottimizzazione del Gas

La sicurezza è un aspetto critico quando si tratta degli Smart Contracts e gas fee. Dato che gli Smart Contracts gestiscono spesso asset finanziari o dati sensibili, errori di programmazione possono avere conseguenze finanziarie gravi e irreversibili. La pratica delle "formal verification" [7] può essere impiegata per garantire che uno Smart Contract si comporti come previsto in tutte le situazioni. La verifica formale permette di analizzare il codice matematicamente per identificare possibili vulnerabilità o bug. Questo processo aiuta gli sviluppatori a individuare e correggere potenziali falle di sicurezza prima che gli Smart Contracts vengano implementati sulla rete principale.

La sicurezza si estende anche alla gestione delle chiavi private, poiché una chiave compromessa potrebbe portare a un accesso non autorizzato agli asset associati.

Operazione	Gas	Descrizione
ADD/SUB	3	Operazione Aritmetiche
JUMP	8	Unconditional Jump
SSTORE	5,000/20,000	Operazioni di storage
BALANCE	400	Saldo di un conto
CALL	25,000	Creazione di un nuovo account
CREATE	32,000	Creazione di un nuovo account

Tabella 2.1. Gas fee su Ethereum

Gli Smart Contracts ben progettati dovrebbero includere meccanismi di aggiornamento del codice solo con il consenso degli utenti coinvolti, evitando così potenziali vulnerabilità.

L'ottimizzazione del gas è una competenza cruciale per gli sviluppatori degli Smart Contracts. Utilizzare librerie efficienti, evitare iterazioni e calcoli onerosi quando possibile sono solo alcune delle pratiche consigliate per ridurre i costi operativi. Esistono anche strumenti di analisi del gas che consentono agli sviluppatori di valutare il consumo di gas del proprio codice durante lo sviluppo.

Il problema dei costi e delle congestioni sulla rete Ethereum è un tema dibattuto nella comunità. Le fasi di transizione verso Ethereum 2.0, un upgrade che ha portato ad una migrazione da un consenso basato su proof-of-work a proof-of-stake, sono infatti aggiornamenti seguiti con grande interesse. Si spera che ciò possa contribuire a ridurre i problemi di congestione e a migliorare la scalabilità, influenzando positivamente i costi associati alle transazioni e agli Smart Contracts.

L'utilizzo del gas nella prevenzione degli attacchi

L'importanza del gas diventa evidente nel contesto della prevenzione degli attacchi di loop infiniti. Un attacco di loop infinito si verifica quando un malintenzionato cerca di sfruttare la possibilità di eseguire cicli di calcolo infiniti all'interno di uno Smart Contract, portando a una congestione della rete e consumando risorse in modo eccessivo. Il meccanismo del gas agisce come una misura preventiva [43] introducendo un costo associato all'esecuzione di ciascuna operazione. In altre parole, gli utenti devono pagare in anticipo per l'esecuzione delle operazioni, e se il gas disponibile viene esaurito durante l'esecuzione, gli Smart Contracts vengono interrotti.

Questo approccio serve a impedire che attacchi malevoli provochino il blocco della rete o il rallentamento delle operazioni, garantendo che ciascuna operazione richieda una quantità finita di risorse computazionali. In questo modo, la blockchain rimane resiliente e efficiente, evitando l'esecuzione infinita di operazioni dannose. La tabella 2.1 rappresenta le operazioni principali all'interno della macchina virtuale Ethereum (EVM) insieme al costo in gas associato a ciascuna operazione e una breve descrizione di cosa fa l'operazione. Il gas è l'unità di misura della quantità di lavoro eseguita o della complessità computazionale di un'operazione all'interno della rete Ethereum.

Sicurezza negli Smart Contracts

L'approccio di Ethereum alla sicurezza degli Smart Contracts è caratterizzato da diverse considerazioni importanti. Una delle principali caratteristiche di sicurezza di Ethereum è l'implementazione di un'architettura a macchina a stati, che fornisce un ambiente isolato e sandboxed per l'esecuzione degli Smart Contracts. L'EVM

garantisce che gli Smart Contract non possano interferire direttamente con il funzionamento del sistema sottostante o con altri contratti, riducendo così il rischio di attacchi dannosi o comportamenti indesiderati.
Un altro aspetto cruciale per la sicurezza di Ethereum è il determinismo. Le transizioni di stato e gli stati stessi sono deterministici [42], il che significa che l'esecuzione di uno Smart Contract produrrà risultati coerenti su tutti i nodi della rete. Questa consistenza è fondamentale per prevenire manipolazioni malevole o risultati imprevedibili.

Inoltre, Ethereum offre la possibilità di sottoporre gli Smart Contracts a un audit di sicurezza. Gli audit di sicurezza coinvolgono una revisione approfondita del codice da parte di esperti di sicurezza al fine di identificare potenziali vulnerabilità o debolezze. Questo processo consente agli sviluppatori di apportare eventuali miglioramenti per garantire la sicurezza degli Smart Contracts.

Ethereum fornisce anche meccanismi per gli aggiornamenti e miglioramenti degli Smart Contracts nel tempo. Questa caratteristica è particolarmente importante per correggere eventuali vulnerabilità identificate dopo l'implementazione iniziale. Gli sviluppatori possono distribuire nuove versioni degli Smart Contracts che includono miglioramenti di sicurezza.

Il paper di Daojing et al. [27] discute proprio sulle vulnerabilità di sicurezza negli Smart Contract, evidenziando esempi significativi di incidenti passati, come il caso DAO del 2016 e altri episodi che hanno portato a ingenti perdite finanziarie. Si sottolinea la natura immutabile degli Smart Contracts una volta implementati sulla blockchain, rendendo cruciale la prevenzione di errori e vulnerabilità durante lo sviluppo e la distribuzione.

Le vulnerabilità sono categorizzate in base a diversi tipi di errori di programmazione, tra cui la mancata verifica dei risultati delle chiamate di funzione, che può portare a errori di esecuzione e perdita di risorse. Viene fornito un esempio con lo Smart Contract di RichMan , il quale non effettua una verifica adeguata della riuscita del trasferimento prima di procedere con l'assegnazione del titolo di "rich man", consentendo a potenziali attaccanti di sfruttare la situazione per ottenere il titolo senza effettuare il trasferimento richiesto.

In sintesi, il testo mette in luce l'importanza della sicurezza nell'implementazione dei contratti intelligenti sulla blockchain, evidenziando la necessità di una rigorosa analisi e verifica del codice per prevenire gravi conseguenze finanziarie e potenziali abusi da parte di malintenzionati.

La stessa comunità attiva di sviluppatori e professionisti aiutano il miglioramento nella sicurezza condividendo best practice e linee guida per lo sviluppo sicuro degli Smart Contracts. Questo scambio di conoscenze contribuisce a una cultura di sicurezza e migliora la sicurezza complessiva degli Smart Contracts su Ethereum.

Esempio di vulnerabilità di uno Smart Contract

Consideriamo lo Smart Contract 2.2 vulnerabile che gestisce un registro di bilanci. Il seguente codice rappresenta uno Smart Contract vulnerabile a un attacco di overflow:

```
1 //Nota: Questo Smart Contract    intenzionalmente vulnerabile a
    ↪ scopo illustrativo.
2
3 contract VulnerableWallet {
4     mapping(address => uint) public balances;
5
```

```
6       // Funzione per depositare fondi nel portafoglio
7       function deposit() public payable {
8           balances[msg.sender] += msg.value;
9       }
10
11      // Funzione per prelevare fondi dal portafoglio
12      function withdraw(uint _amount) public {
13          require(balances[msg.sender] >= _amount, "Insufficient
    ↪ funds");
14          msg.sender.call{value: _amount}("");
15          balances[msg.sender] -= _amount;
16      }
17  }
```

Listing 2.2. Esempio di uno Smart Contract vulnerabile

Eseguito questo Smart Contract è vulnerabile a un attacco di overflow, con conseguenti rischi di perdita di fondi.

2.2 Trusted Execution Environment

Le TEEs è un ambiente di esecuzione resistente alle manipolazioni, che garantisce l'autenticità del codice eseguito, l'integrità degli stati di runtime (ad esempio i registri CPU, la memoria e le I/O sensibili) e la riservatezza del codice, dei dati e degli stati di runtime memorizzati su memoria persistente. Inoltre, deve essere in grado di fornire una garanzia di affidabilità a terze parti. Il contenuto delle TEEs può essere aggiornato in modo sicuro e resistere a tutti gli attacchi software e fisici alla memoria del sistema.

La definizione delle TEEs comprende la protezione dei suoi stati di runtime e degli asset memorizzati, la necessità di isolamento e di memorizzazione sicura. Deve essere in grado di gestire facilmente il suo contenuto tramite l'installazione o l'aggiornamento del suo codice e dei dati e di definire meccanismi per attestare in modo sicuro la sua affidabilità a terze parti. Il modello di minaccia include tutti gli attacchi software e fisici alla memoria principale e alla memoria non volatile da parte di un avversario potente.

Perciò la definizione delle TEEs si basa sulla protezione dell'ambiente di esecuzione e sulla gestione sicura dei suoi contenuti, garantendo la sua affidabilità a terze parti.

2.2.1 Come viene misurata la fiducia

Il paper [41] espone un confronto diretto tra due sistemi TEE su come è possibile quantificare la loro fiducia.
Nell'informatica, la fiducia è **statica** o **dinamica**. La fiducia statica si basa su una valutazione esaustiva contro un insieme specifico di requisiti di sicurezza. Il **Common Criteria (CC)** è uno standard internazionale che fornisce misure di garanzia per la valutazione della sicurezza. Il CC specifica sette livelli di garanzia di valutazione (EAL1-EAL7) [8], dove i livelli con numeri più alti includono tutti i requisiti dei livelli precedenti. Nella fiducia statica, l'affidabilità di un sistema viene misurata solo una volta e prima della sua distribuzione.

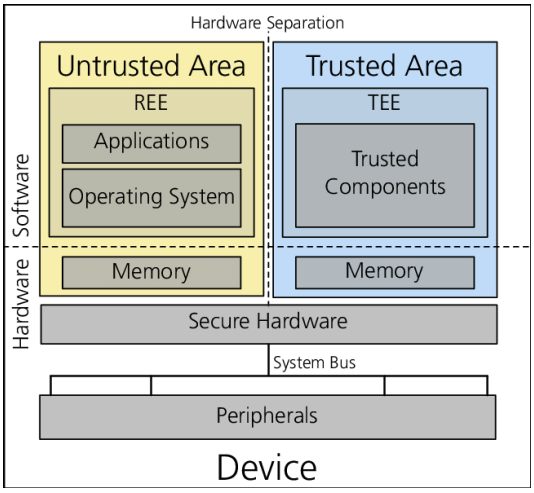

Figura 2.8. Descrizione del hardware e software di un TEE.

La fiducia dinamica rappresenta un concetto in cui la fiducia in un sistema è continuamente valutata e adattata in base al suo stato in evoluzione. Questo tipo di fiducia si basa sulla misurazione costante dell'affidabilità del sistema durante l'intero ciclo di vita. Essenzialmente, un sistema cambia costantemente il suo "stato di fiducia" in risposta alle condizioni in cui si trova.

La fiducia dinamica richiede l'esistenza di un mezzo sicuro e affidabile che fornisca prove riguardo allo stato di fiducia di un sistema specifico. Questa definizione presuppone l'importanza di un'entità fidata conosciuta come Root of Trust (RoT) [18] per fornire tali prove attendibili.

Il RoT ha due ruoli distinti: la misurazione affidabile e il calcolo del punteggio di fiducia. L'affidabilità del sistema, ovvero il valore del punteggio di affidabilità generato, dipende strettamente dall'affidabilità delle misurazioni di fiducia. Qualora un'entità malintenzionata potesse influenzare tali misurazioni, il punteggio di affidabilità non avrebbe alcun valore. Di conseguenza, il RoT deve essere implementato come un modulo hardware resistente alle manipolazioni.

2.2.2 Fiducia nelle TEEs

La fiducia nelle TEEs è una fiducia **ibrida** [41]; è sia statica che semidinamica. Prima della distribuzione, le TEEs devono essere certificate verificando approfonditamente il loro livello di sicurezza in conformità con un profilo di protezione, un documento che contiene un insieme predefinito di requisiti di sicurezza. Ad esempio, la **GlobalPlatform** definisce un profilo di protezione che conforma a EAL2. Inoltre, durante ogni avvio, il RoT assicura che la TEE caricata sia quella certificata dal fornitore della piattaforma, proteggendo l'integrità del codice. Una volta in esecuzione, l'integrità è protetta dal sottostante kernel di separazione.

La fiducia nelle TEEs è considerata **semidinamica** perché la TEEs non dovrebbe cambiare il suo livello di fiducia durante l'esecuzione poiché è protetta dal kernel di separazione. In questo modello di fiducia, le misurazioni di fiducia sono misurazioni di integrità, e il punteggio di fiducia è un booleano che indica lo stato di integrità del codice. La TEE è considerata affidabile quando il suo punteggio di fiducia è vero,

altrimenti è considerata non affidabile. La qualità del punteggio di fiducia dipende dalle misurazioni definite per l'integrità. La Figura 2.8 definisce come la TEE implica la creazione di un'area sicura separata dall'OS principale e dalle applicazioni, in cui esegue solo codice fidato. Questa area, chiamata Trusted Area, si distingue da un'area non fidata, chiamata Untrusted Area. La configurazione specifica varia tra le tecnologie, ma tutte si basano sulla distinzione di un'area sicura e di una non sicura per definire la TEE.[24]

2.2.3 Il ruolo delle TEEs nella sicurezza

La tecnologia del Trusted Computing [51] svolge un ruolo cruciale nell'assicurare la secure computation, la privacy e la protezione dei dati. Il Trusted Platform Module (TPM) rappresenta un componente crittografico hardware dedicato, attentamente progettato per garantire la sicurezza delle piattaforme informatiche. Questo modulo fornisce funzionalità di sicurezza hardware integrando un chip di sicurezza nella scheda madre di un computer o in altri dispositivi quali router, stampanti e dispositivi IoT.

Il TPM riveste un ruolo fondamentale nella salvaguardia delle informazioni sensibili memorizzate nel sistema, come ad esempio le password utente e le chiavi di crittografia. Oltre a ciò, il TPM può generare chiavi crittografiche utilizzate per autenticazione e crittografia dei dati. L'ambiente affidabile e protetto fornito dal modulo TPM consente la gestione sicura delle chiavi crittografiche, in quanto queste vengono memorizzate internamente e non sono accessibili dall'esterno.

Un ulteriore utilizzo del TPM consiste nella fornitura di prove riguardo all'integrità del sistema, fornendo informazioni sulla presenza di software o hardware dannosi che potrebbero compromettere la sicurezza complessiva del sistema. Importante sottolineare che il TPM è un'iniziativa sviluppata dalla *Trusted Computing Group* (TCG), un'organizzazione impegnata a promuovere la sicurezza informatica attraverso lo sviluppo di standard aperti e interoperabili per la sicurezza hardware. [30]

2.2.4 Implementazioni di una TEE

Una TEE può essere realizzata attraverso un coprocessore, si tratta di un nucleo separato, spesso dotato delle proprie periferiche, viene impiegato per gestire attività critiche per la sicurezza al di fuori dell'ambiente operativo principale. Questa configurazione offre vantaggi come l'isolamento completo delle operazioni, consentendo simultaneità con il nucleo principale. Tuttavia, presenta svantaggi legati al costo associato al trasferimento dei dati da e verso il nucleo principale, oltre alla potenziale limitazione di potenza computazionale del coprocessore. All'interno di questo approccio, possiamo distinguere due sottocategorie:

- **Coprocessore di sicurezza esterno**: Questa implementazione impiega un modulo hardware discreto, esterno al chip fisico con il nucleo principale. Ciò assicura un completo isolamento, senza condividere risorse con il nucleo principale.
- **Coprocessore Embedded Security**: Viene integrato direttamente nel System-on-Chip (SoC) [29] principale, questo coprocessore ha la capacità di condividere alcune risorse con il sistema principale pur mantenendo un isolamento completo dal processore principale.

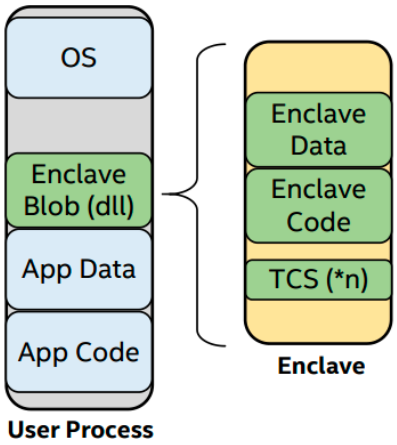

Figura 2.9. Logica enclave Intel-SGX [17]

2.2.5 Configurazioni di Architetture di una TEE

Intel Software Guard Extensions (SGX) rappresentano una tecnologia di sicurezza hardware sviluppata da Intel con l'obiettivo di proteggere applicazioni e dati in esecuzione su processori Intel. SGX consente ai programmi di istanziare delle aree sicure, chiamate enclave, all'interno della memoria del processore, dove sia i dati che il codice sono al riparo da accessi non autorizzati, compresi quelli con privilegi di amministratore del sistema.

Le enclave SGX sono progettate per mitigare rischi quali l'accesso non autorizzato ai dati da parte di malware, attacchi di reverse engineering e tentativi di spoofing. In aggiunta, le enclave SGX possono essere sfruttate per la protezione della proprietà intellettuale, come segreti commerciali e chiavi di crittografia. La Figura 2.9 rappresenta la logica dell'enclave Intel-SGX.

L'implementazione di SGX è disponibile su processori Intel Core di sesta generazione o successive, processori Xeon E3 v5 o successivi e processori Atom di ottava generazione o successive. È importante sottolineare che l'utilizzo di SGX richiede che il sistema supporti questa tecnologia e che il software sia stato appositamente sviluppato per adottarla. [50]

2.2.6 Nascita di Trusted Execution Technology e Intel SGX

Intel Trusted Execution Technology (TXT) rappresenta una collezione di funzioni di sicurezza validate mediante un metodo che verifica la configurazione del sistema e del codice [15]. Questo metodo misura la sicurezza e stabilisce un livello di affidabilità basato su metriche specifiche. TXT fornisce inoltre una funzione di sicurezza aggiuntiva per sistemi operativi attendibili.

Intel ha ufficialmente introdotto SGX2 nel 2016, accompagnato dal lancio della prima CPU fisica (Skylake) che supporta SGX. Questo ha anche portato allo sviluppo di documentazione dettagliata. SGX offre un ambiente di esecuzione sicuro e confidenziale, consentendo a un'applicazione o a una parte di essa di operare in un contenitore sicuro noto come Enclave. L'applicazione è suddivisa in parti attendibili e non attendibili.

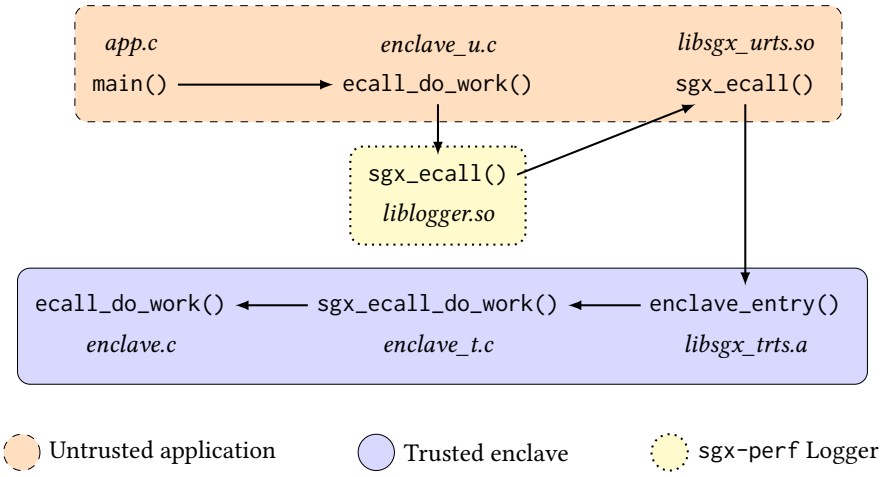

Figura 2.10. Le transizioni operative dell'enclave avvengono nell'Untrusted Runtime System (URTS) e nel Trusted Runtime System (TRTS). L'SDK utilizza un punto di ingresso comune con un trampoline per gestire le chiamate ECALL e OCALL. [46]

I dati sensibili e le relative funzioni risiedono all'interno dell'enclave, con accesso esterno rigorosamente vietato per prevenire possibili attacchi. Le funzioni sensibili vengono elaborate all'interno dell'enclave, garantendo la riservatezza dei dati. L'enclave e la struttura dati associata sono memorizzate nella CPU Enclave Page Cache (EPC) [36]. L'accesso al codice e ai dati dell'enclave può avvenire solo attraverso un'interfaccia specifica, mentre qualsiasi tentativo di accesso esterno viene interrotto. L'immagine presentata rappresenta un diagramma che illustra il flusso di lavoro tra un'applicazione non attendibile e un enclave attendibile in un ambiente di esecuzione fidato (TEE) utilizzando la tecnologia SGX.

La Figura 2.10 evidenzia tre file chiave nel processo di esecuzione: `app.c`, `enclave_u.c` e `enclave_t.c`. La funzione `main()` nel file `app.c` inizia il flusso di lavoro invocando la funzione `ecall_do_work()` nel file `enclave_u.c`. Quest'ultimo, a sua volta, utilizza la libreria `libsgx_urts.so` per effettuare una `sgx_ecall()` alla libreria `liblogger.so`.

La funzione `ecall_do_work()` in `enclave_u.c` è mappata alla corrispondente funzione nel file `enclave_t.c` attraverso le chiamate e le librerie indicate. Infine, la funzione `ecall_do_work()` in `enclave_t.c` invoca la funzione `enclave_entry()` nella libreria `libsgx_trts.a`.

Il diagramma utilizza linee tratteggiate e frecce per indicare il flusso del processo da un file o una libreria all'altra, fornendo una rappresentazione visiva del flusso di controllo tra i vari componenti in un ambiente TEE con SGX. Questo diagramma può essere utilizzato come riferimento per comprendere la sequenza di operazioni e le interazioni tra i vari componenti in una TEE.

Le caratteristiche chiave di Intel SGX includono:

1. Suddivisione dell'applicazione in parti attendibili e non attendibili.
2. Incapsulamento delle operazioni di sicurezza dei dati sensibili in un'enclave.
3. Accesso alle funzioni dell'enclave solo attraverso l'interfaccia designata.

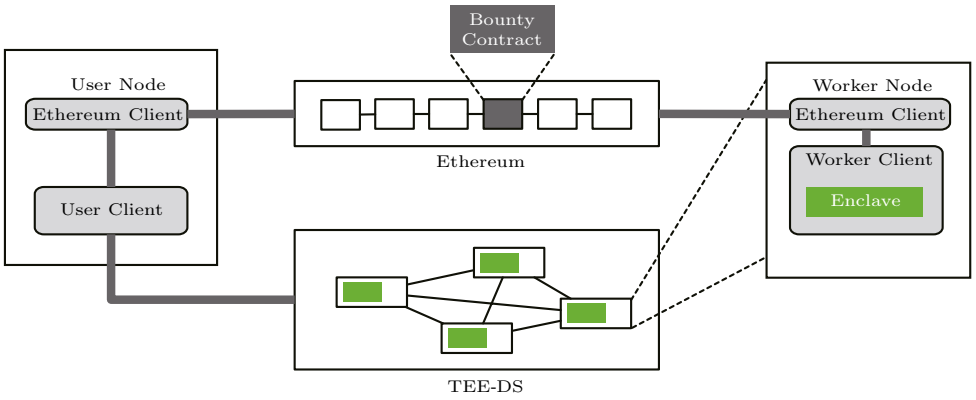

Figura 2.11. Architettura ShadowEth [48]

Solo il codice interno dell'enclave può visualizzare i dati, rifiutando sempre l'accesso esterno. Al termine della chiamata, i dati dell'enclave rimangono nella memoria protetta.

Con il crescente spostamento di servizi nel cloud, emergono nuovi rischi di sicurezza e metodi di attacco. Intel ha affrontato questa sfida con TXT, estensioni hardware progettate per migliorare la sicurezza e prevenire attacchi software. [46]

Caratteristiche principali di Intel SGX:

1. Area di esecuzione fidata per la protezione dei dati sensibili.
2. Integrazione con TXT per una sicurezza avanzata.
3. Sviluppo di un Software Development Kit (SDK) per facilitare l'implementazione.

Esempio ShadowEth

Il paper "ShadowEth: Private Smart Contract on Public blockchain" [48] propone ShadowEth, un sistema innovativo che sfrutta l'hardware enclave per garantire la privacy degli Smart Contracts su blockchain pubbliche come Ethereum. ShadowEth crea un ambiente confidenziale e sicuro al di fuori della blockchain pubblica, eseguendo e archiviando contratti privati all'interno di un TEE. Il suo obiettivo principale è proteggere la specifica, l'esecuzione e lo stato dei contratti privati, garantendo la riservatezza dei dati in modo crittograficamente sicuro.

Per mantenere la privacy dello stato persistente, ShadowEth adotta una strategia specifica: memorizzare solo l'hash del registro su Ethereum anziché tutti i dati. Questo permette di gestire e visualizzare i dati solo all'interno delle enclave, assicurando che siano accessibili solo agli utenti autenticati tramite ShadowEth. Tuttavia, date le limitazioni della memoria sicura delle enclave come SGX, potrebbe essere necessario spostare i dati su memoria non attendibile o su disco e, talvolta, trasmetterli attraverso la rete per backup o sincronizzazione. Prima di scrivere i dati, ShadowEth li cripta con una chiave hardware conservata solo dai CPU, garantendo un ulteriore livello di sicurezza.

Questo approccio innovativo offre un'elevata protezione dei dati e della privacy, creando un ambiente sicuro e conforme per l'esecuzione dei contratti privati su

blockchain pubbliche come Ethereum. La figura 2.11 illustra l'architettura di ShadowEth, evidenziando la sua struttura e il flusso dei dati all'interno del sistema.

2.3 Esecuzione degli Smart Contracts nelle TEEs

La trasparenza degli Smart Contracts può rappresentare una minaccia per la privacy dei dati. La divulgazione pubblica di informazioni sensibili, come dati e diagnosi di un sistema sanitario, non solo viola la privacy dei pazienti, ma crea anche vulnerabilità per la sicurezza dei dati medici.

Per affrontare efficacemente questo problema, si propone l'adozione degli Smart Contracts assistiti da TEE. L'utilizzo delle TEEs consente di eseguire in modo sicuro i calcoli critici e di gestire i dati sensibili all'interno di un ambiente protetto e isolato. In questo modo, la riservatezza dei dati medici viene preservata, garantendo che solo le informazioni necessarie vengano rese accessibili sulla blockchain pubblica.

Nel contesto del sistema sanitario decentralizzato, per esempio, le TEEs agiscono come scudi protettivi intorno alle operazioni sensibili eseguite dagli Smart Contracts. All'interno di una TEE, il codice dello Smart Contract e i dati sensibili dei pazienti possono essere crittografati e resi inaccessibili a entità non autorizzate. Solo le parti di dati necessarie per la validazione e l'esecuzione delle transazioni vengono condivise sulla blockchain pubblica, garantendo la riservatezza delle informazioni mediche.

Ad esempio, un paziente potrebbe eseguire uno Smart Contract per consentire l'accesso ai suoi dati medici a un professionista sanitario specifico. Utilizzando un TEE, uno Smart Contract può garantire che solo il professionista autorizzato sia in grado di accedere ai dati sensibili, proteggendo così la privacy del paziente. Inoltre, l'uso delle TEEs consente di verificare l'integrità dell'esecuzione dello Smart Contract, proteggendo da possibili manipolazioni o frodi.

L'adozione degli Smart Contracts assistiti da TEE rappresenta una soluzione promettente per preservare la privacy dei dati basati su blockchain. L'utilizzo di TEE consente di garantire la riservatezza delle informazioni sensibili e di proteggere l'integrità dell'esecuzione degli Smart Contracts, fornendo un ambiente sicuro e affidabile per la gestione dei dati degli utenti.

2.3.1 Vantaggi in termini di sicurezza e efficienza

L'esecuzione dello Smart Contract all'interno di una TEE comporta una serie di vantaggi significativi, sia in termini di sicurezza che di efficienza. Le TEEs sono ambienti isolati e sicuri progettati per proteggere l'esecuzione di codice, garantendo confidenzialità, integrità e autenticità. In questa sezione, esploreremo approfonditamente i vantaggi derivanti dall'uso di una TEE per l'esecuzione dello Smart Contract. La Figura 2.12 descrive come una transazione viene eseguita prima sul TEE e successivamente la risposta viene inviata alla blockchain.

2.3.2 Sicurezza

L'utilizzo delle TEEs negli Smart Contract rappresenta un'avanzata soluzione di sicurezza e privacy nell'ambito delle tecnologie blockchain. Le TEEs offrono un ambiente di esecuzione isolato, che è una componente fondamentale per la sicurezza degli Smart Contract. Questo ambiente separato protegge il codice e i dati degli Smart Contract da interventi esterni, inclusi gli attacchi software che potrebbero provenire dal sistema operativo o da altri processi in esecuzione sulla stessa macchina. Questo tipo di isolamento è cruciale perché impedisce ai malware o ad attaccanti di alterare o carpire le informazioni gestite dagli Smart Contract, mantenendo

!

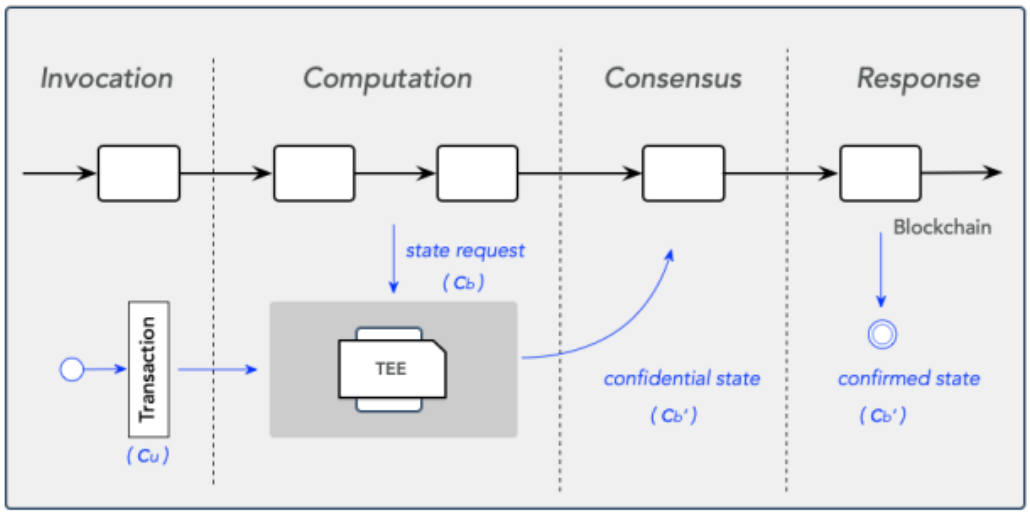

Figura 2.12. Workflow Sc & TEE

l'integrità e la fiducia nel sistema. Le TEE non solo proteggono dagli attacchi software, ma offrono anche una robusta difesa contro le manomissioni fisiche. Questo è particolarmente importante in scenari dove i dispositivi possono essere soggetti a tentativi di violazione fisica, come il furto o la manipolazione hardware. Questa resistenza alle manomissioni fisiche assicura che l'integrità del TEE e, di conseguenza, quella dello Smart Contract rimangano inalterate, anche in condizioni avverse.

Un altro aspetto fondamentale riguardo la sicurezza è la protezione delle chiavi crittografiche in quanto sono vitali in qualsiasi sistema basato su blockchain e Smart Contract. La protezione delle chiavi private è cruciale per prevenire accessi non autorizzati. Soluzioni come l'uso di hardware sicuro all'interno delle TEEs e la criptografia basata su chiavi asimmetriche offrono un livello aggiuntivo di sicurezza. Le chiavi asimmetriche, costituite da una chiave pubblica e una chiave privata, fondamentali nella crittografia moderna. La chiave pubblica è divulgata apertamente e utilizzata per crittografare dati o verificare firme digitali, mentre la chiave privata, mantenuta segreta, consente la decrittografia o la generazione di firme. Questo sistema permette transazioni sicure, autenticità dei dati e condivisione sicura di informazioni, svolgendo un ruolo cruciale in blockchain e sicurezza informatica. Oltre a proteggere il codice, le TEEs garantiscono che i dati trattati all'interno degli Smart Contract rimangano confidenziali. Questo è particolarmente rilevante quando si gestiscono dati sensibili o personali. La capacità di processare dati in modo confidenziale all'interno di un TEE assicura che le informazioni siano accessibili solo alle entità autorizzate, impedendo a Smart Contract esterni o ad altri attori sulla blockchain di accedervi.

La Figura 2.8 rappresenta la definizione concettuale di un TEE. Preservare la privacy e la confidenzialità dei dati è una preoccupazione significativa quando si eseguono Smart Contract sulle TEEs. La memorizzazione di dati sensibili all'interno di un ambiente sicuro può essere un punto di forza, ma è fondamentale garantire che i dati non siano esposti nemmeno durante l'esecuzione dello Smart Contract. L'utilizzo di tecniche di crittografia omomorfica e di protocolli di scambio di dati sicuri può mitigare questa sfida.

2.3.3 Efficienza

L'impiego delle TEEs negli Smart Contract offre notevoli vantaggi in termini di efficienza e riduzione dei costi. Grazie all'elevato rendimento delle TEEs, è possibile garantire un'esecuzione veloce e puntuale degli Smart Contract, un aspetto fondamentale in scenari dove la tempestività delle transazioni, come nei sistemi finanziari o nelle applicazioni di pagamento, è cruciale. Questa accelerazione delle operazioni si accompagna alla minimizzazione dell'overhead legato alla gestione della sicurezza a livello di sistema operativo, portando a un miglioramento generale delle prestazioni degli Smart Contract e a una maggiore efficienza nelle transazioni. Inoltre, le TEEs possono ridurre il carico sulla blockchain poiché permettono di eseguire operazioni senza dover registrare ogni dettaglio su una blockchain pubblica, consentendo così di conservare solo le informazioni essenziali e contribuendo a mantenere la leggerezza e l'efficienza della catena di blocchi. Infine, l'esecuzione all'interno di una TEE può ridurre la quantità di risorse computazionali richieste rispetto all'esecuzione su una blockchain pubblica, portando a un risparmio significativo sui costi di gas associati alle transazioni, un vantaggio non trascurabile in termini economici.

2.3.4 Meccanismi per garantire l'integrità e la trasparenza

Il documento *"SoK: TEE-assisted Confidential Smart Contract"* [34] affronta il problema della mancanza di privacy negli Smart Contracts basati su blockchain e propone l'utilizzo delle TEEs come soluzione efficace. Gli Smart Contracts basati su blockchain sono noti per la loro trasparenza, in cui lo stato degli Smart Contracts e il codice di istruzioni sono accessibili al pubblico. Tuttavia, questa trasparenza limita l'adozione su larga scala degli Smart Contracts, in particolare per le applicazioni che richiedono la protezione dei dati sensibili dell'utente.

Per affrontare questo problema, si propone l'utilizzo proprio delle TEEs. I calcoli critici e i dati sensibili sono gestiti in modo sicuro all'interno di una TEE, preservando la riservatezza, mentre solo informazioni necessarie sono rese accessibili sulla blockchain pubblica.

Per affrontare questa problematica, i ricercatori hanno esplorato diverse soluzioni crittografiche, come le prove a conoscenza zero (ZKP), la crittografia omomorfica (HE) e il calcolo multi-partecipante sicuro (MPC). Tuttavia, queste soluzioni sono adatte solo per applicazioni con calcoli semplici.

Sono state proposte diverse piattaforme di Smart Contract che sfruttano i TEE, come Alibaba CONFIDE, LucidiTEE di Visa e CHANG'AN Chain della Cina. Tuttavia, molti progetti recenti non hanno tratto insegnamento dai protocolli esistenti e possono presentare rischi di sicurezza a causa di errori di progettazione noti o dell'applicazione non sicura della crittografia.

Per affrontare questa mancanza di esperienza e per fornire una guida per lo sviluppo di Smart Contracts assistiti da TEE, il documento propone una classificazione sistematica dei sistemi esistenti in due categorie: soluzioni di livello uno e soluzioni di livello due. Come anticipato precedentemente

Inoltre, viene presentato un framework di valutazione unificato che considera le proprietà desiderabili per gli Smart Contracts, i modelli di minaccia e le considerazioni di sicurezza dei sistemi sottostanti. Vengono identificate le funzionalità ideali dei sistemi degli Smart Contracts assistiti dalle TEEs e vengono analizzate le lacune

fondamentali e le sfide di progettazione di ciascuna specifica.

Il documento include un'analisi comparativa dei protocolli esistenti basata sul framework di valutazione proposto. Vengono discussi sia i design comuni che le caratteristiche distintive dei sistemi, evidenziando le idee coerenti e gli aspetti innovativi di ciascun progetto.

Capitolo 3

Esempio di utilizzo in campo sanitario

In Europa, vige il Regolamento Generale sulla Protezione dei Dati (GDPR) [1], entrato in vigore nel maggio 2018 che stabilisce regole chiare per la raccolta, l'elaborazione e la conservazione dei dati personali, inclusi i dati medici, mentre negli Stati Uniti, la legge principale che regola i dati medici dei pazienti è la Health Insurance Portability and Accountability Act (HIPAA) [2], approvata nel 1996. HIPAA stabilisce standard per la protezione della privacy e della sicurezza delle informazioni sanitarie identificabili (PHI) e impone ai dipendenti, ai fornitori di servizi sanitari e agli assicuratori sanitari di proteggere la riservatezza e la sicurezza delle informazioni sanitarie dei pazienti. Nel contesto sanitario, la gestione dei dati dei pazienti è di fondamentale importanza al fine di osservare queste leggi e quindi garantire la riservatezza, l'integrità e la sicurezza delle informazioni sensibili.

La segregazione dei dati per la privacy è fondamentale per proteggere le informazioni sensibili dei pazienti. I dati che devono essere segregati in modo appropriato includono le informazioni personali identificative. Questi dati possono essere utilizzati solo da personale autorizzato per scopi specifici legati alla cura del paziente e devono essere accessibili solo secondo le necessità.

Al contrario, i dati clinici del paziente, come i risultati degli esami, le diagnosi e i trattamenti medici, possono essere condivisi all'interno dell'ospedale tra il personale sanitario coinvolto nella cura del paziente. Tuttavia, anche questi dati devono essere accessibili solo da operatori autorizzati e devono essere utilizzati solo per scopi clinici legittimi.

I dati sensibili dei pazienti possono essere suddivisi in due categorie principali, dati altamente sensibili e dati meno sensibili. La prima categoria include informazioni personali identificative come nome, data di nascita, codice fiscale, numero di assicurazione sanitaria e contatti. Questi dati sono considerati altamente sensibili poiché possono essere utilizzati per identificare inequivocabilmente un individuo e possono essere soggetti a gravi rischi per la privacy se non gestiti correttamente. Anche informazioni mediche specifiche di un paziente, come diagnosi mediche, risultati degli esami, trattamenti, prescrizioni e note cliniche sono altamente sensibili per natura e devono essere trattati con estrema cautela per garantire la privacy del paziente.

Secondo il GDPR, i dati considerati meno sensibili o non particolarmente identificativi sono quelli che non rivelano informazioni dirette o indirette riguardanti un individuo specifico. Questi dati sono generalmente categorizzati come "dati non

[1] https://gdpr-info.eu/

[2] https://www.hhs.gov/hipaa/index.html

sensibili" o "dati anonimi". Essi includono informazioni generiche come età, genere, nazionalità e lingua madre, che non possono essere facilmente utilizzate per identificare un individuo in modo univoco. Allo stesso modo, i dati di contatto generici come un indirizzo e-mail generico o un numero di telefono aziendale rientrano in questa categoria, in quanto non forniscono dettagli personali specifici di un individuo.

Questi tipi di dati possono essere utilizzati per scopi di analisi, ricerca e marketing senza richiedere il consenso esplicito dell'individuo, a condizione che vengano trattati in conformità con i principi di protezione dei dati stabiliti dal GDPR, inclusi il principio di minimizzazione dei dati e il principio di anonimizzazione. Tuttavia, è importante tenere presente che anche i dati meno sensibili devono essere trattati in modo responsabile e protetti da accessi non autorizzati.

Immaginiamo un tipico scenario in un ospedale, dove un sistema è utilizzato per registrare e gestire i dati sanitari dei pazienti e immaginiamo un nuovo sistema chiamato "HospitalData" che mira a facilitare l'accesso ai dati sanitari. Alice è un paziente che si presenta in ospedale con problemi diabetici, mentre Bob è un infermiere responsabile dell'accoglienza e della registrazione dei pazienti.

Alice viene registrata nel sistema di HospitalData, dove vengono raccolte informazioni personali come nome, data di nascita, contatti e altre informazioni utili sulla sua salute. Durante il suo periodo di degenza, ogni interazione medica o aggiornamento relativo al paziente, tra cui esami e trattamenti, viene registrata nel sistema. Quando Alice viene dimessa, la sua uscita viene annotata nel registro sanitario.

Nel contesto dell'esame per valutare il livello di insulina di Alice, il sistema HospitalData oltre a registrare i risultati dell'esame, abbia anche la capacità di elaborare automaticamente il dosaggio della cura in base ai risultati ottenuti. Dopo che il medico ha inserito i risultati dell'esame nel sistema, il sistema stesso utilizza questi dati sensibili per calcolare il dosaggio ottimale di insulina da somministrare ad Alice. Utilizzando algoritmi specifici e regole cliniche predefinite, il sistema può analizzare i risultati dell'esame e determinare la quantità di insulina necessaria per mantenere i livelli di zucchero nel sangue di Alice entro i limiti accettabili. Una volta calcolato il dosaggio, il sistema può generare automaticamente una raccomandazione di trattamento per il medico, indicando la quantità specifica di insulina da somministrare ad Alice e le istruzioni per l'amministrazione. Il medico può quindi valutare la raccomandazione del sistema e decidere se accettarla o apportare eventuali modifiche in base alla sua valutazione clinica e alla situazione specifica di Alice. Questo processo automatizzato consente una gestione più efficiente e precisa della terapia insulinica di Alice, riducendo al minimo il rischio di errori umani e garantendo che il dosaggio della cura sia personalizzato in base ai suoi bisogni individuali e ai risultati dell'esame. Inoltre, i dati sensibili di Alice sono protetti da esposizione a fattori esterni grazie al sistema HospitalData, il quale, basandosi sui dati di Alice, calcola automaticamente il livello di insulina senza compromettere la riservatezza dei dati sensibili.

HospitalData impone regole rigide per l'accesso e l'utilizzo dei dati dei pazienti. Ad esempio, solo il personale medico autorizzato può accedere ai dati, che devono essere utilizzati solo per scopi clinici e solo all'interno dell'ospedale.

Quando Bob cerca di accedere ai dati di Alice per una consultazione successiva, il sistema verifica la sua autorizzazione e applica le regole di accesso stabilite. Bob può visualizzare solo le informazioni pertinenti al suo ruolo come infermiere e solo se autorizzato.

L'utilizzo di HospitalData offre numerosi vantaggi, tra cui la tracciabilità completa delle informazioni sanitarie dei pazienti, la conformità alle normative sulla privacy e la protezione dei dati sensibili dei pazienti da accessi non autorizzati. Inoltre,

consente una migliore gestione e condivisione delle informazioni tra gli operatori sanitari, migliorando così la qualità dell'assistenza sanitaria complessiva.

Capitolo 4

Progettazione e Realizzazione

4.1 Sviluppo su Ethereum con Geth e Clef

Il capitolo 4 si concentra sullo sviluppo e l'implementazione su Ethereum utilizzando Geth e Clef, fornendo istruzioni dettagliate sull'avvio, il recupero di Ether dalle Testnet e l'interazione con Geth. Successivamente, viene discusso l'implementazione pratica, inclusa la conversione del codice e la presentazione del codice sorgente Solidity.

Nel mondo della blockchain, Ethereum è una delle piattaforme più utilizzate per lo sviluppo e l'esecuzione degli Smart Contracts e applicazioni decentralizzate (DApp). Geth è uno dei client Ethereum più diffusi e offre una vasta gamma di funzionalità per interagire con la blockchain Ethereum. Clef è un gestore di account che può essere utilizzato insieme a Geth per gestire in modo sicuro le chiavi private degli account Ethereum.
Prima di interagire con la blockchain Ethereum, è necessario disporre di un account Ethereum. Un account Ethereum è costituito da una coppia di chiavi: una chiave privata, che consente di firmare le transazioni, e una chiave pubblica, che viene utilizzata per derivare l'indirizzo dell'account sulla blockchain Ethereum. È importante mantenere la chiave privata al sicuro e non condividerla con altri.

Per generare un nuovo account Ethereum utilizzando Geth, eseguire il seguente comando[1]:

```
geth account new
```

Questo comando guiderà attraverso il processo di creazione di un nuovo account e richiederà di inserire una passphrase per proteggere la chiave privata.

4.1.1 Avvio di Clef e Geth

Prima di avviare Geth, è necessario avviare Clef se si desidera utilizzarlo per gestire gli account. Clef fornisce un'interfaccia utente sicura per la gestione delle chiavi private.

Per avviare Clef, eseguire il seguente comando[2]:

```
clef sign
```

[1] https://geth.ethereum.org/docs/interface/command-line-options/account-new

[2] https://geth.ethereum.org/docs/dapp/clef

Clef verrà avviato e inizierà a gestire gli account.
Una volta avviato Clef, è possibile avviare Geth con il seguente comando[3]:

```
geth --syncmode "fast" --cache 1024
```

Questo comando avvia Geth in modalità "syncmode rapido", che consente una sincronizzazione più veloce con la blockchain Ethereum. L'opzione `-cache` specifica la dimensione della cache in megabyte da utilizzare durante la sincronizzazione.

4.1.2 Recupero di Ether dalle Testnet

Per ottenere Ether (ETH) sulla testnet Ethereum, è possibile utilizzare vari metodi, tra cui il mining, i faucet o lo scambio di Ether con altri utenti. Un modo comune per ottenere Ether sulla testnet di Ropsten è utilizzare Sepolia faucet [4]. Basta visitare il sito web e inserire l'indirizzo del proprio account Ethereum per ricevere Ether gratuito.

4.1.3 Interazione con Geth

Una volta avviato Geth, è possibile interagire con esso utilizzando la console JavaScript integrata. Ecco alcuni esempi di comandi che è possibile utilizzare nella console Geth:

- `eth.accounts`: Mostra gli account Ethereum disponibili.
- `eth.getBalance(account)`: Restituisce il saldo dell'account specificato.
- `eth.sendTransaction({from: account, to: recipient, value: amount})`: Invia una transazione dall'account specificato a un destinatario con una quantità specificata di Ether.

4.1.4 Testnet

Le testnet Ethereum sono reti di sviluppo parallele alla rete principale Ethereum, utilizzate per lo sviluppo e il test di applicazioni senza spendere Ether reale. Le testnet consentono agli sviluppatori di sperimentare con la blockchain Ethereum e testare le proprie applicazioni in un ambiente sandbox.
Alcune delle testnet più utilizzate sono Ropsten, Rinkeby e Goerli. È possibile ottenere Ether gratuito per le testnet utilizzando faucet dedicati o partecipando a community di sviluppatori che distribuiscono Ether di test.

4.2 Implementazione

Questa sezione discute i passaggi eseguiti al fine di analizzare come è stata sviluppata una possibile soluzione al problema descritto nei capitoli precedenti.

[3] `https://geth.ethereum.org/docs/interface/command-line-options`
[4] `https://sepoliafaucet.com/`

4.2.1 Conversione del codice

```
solc --abi Storage.sol -o build

abigen --abi Storage.abi --pkg main --type Storage --out Storage.
    ↪ go
```

Listing 4.1. Compilazione degli Smart Contracts e generazione dell'interfaccia in Go

Il codice 4.1 è un esempio di comandi bash utilizzati per compilare uno Smart Contract e generare un'interfaccia in Go per interagire con esso. Il primo comando, `solc -abi Storage.sol -o build`, utilizza il compilatore Solidity (`solc`) per compilare il file `Storage.sol` e generare il file ABI (Application Binary Interface), che definisce l'interfaccia degli Smart Contracts. Il secondo comando, `abigen -abi Storage.abi -pkg main -type Storage -out Storage.go`, utilizza lo strumento `abigen` per generare un'interfaccia Go (`Storage.go`) dal file ABI generato precedentemente. L'opzione `-pkg main` specifica il nome del pacchetto Go da utilizzare, mentre `-type Storage` specifica il nome del tipo da generare.

Questi comandi sono tipicamente utilizzati nello sviluppo di applicazioni blockchain per compilare gli Smart Contract scritti in Solidity e generare le interfacce necessarie per interagire con essi tramite codice Go.

4.2.2 Ego

EGo è un framework che semplifica la creazione di applicazioni per Intel SGX in Go, offrendo un SDK intuitivo, leggero e basato su standard dell'industria. Con EGo, è possibile sviluppare facilmente applicazioni di calcolo confidenziale senza compromettere la sicurezza.

```
# Comando per ottenere il pacchetto go-ethereum
go get github.com/ethereum/go-ethereum

# Comando per ottenere tutte le dipendenze
go get all

# Costruzione dell'eseguibile con EGo
ego-go build smartContract.go

# Firma dell'eseguibile per Intel SGX
ego sign smartContract

# Visualizzazione della configurazione dell'enclave e della
    ↪ chiave privata
cat enclave.json private.pem
```

Listing 4.2. Comandi per la gestione degli Smart Contracts con EGo

Il codice 4.2 sopra mostrato illustra una serie di comandi bash utilizzati per gestire gli Smart Contracts utilizzando il framework EGo [19]. Il primo comando, `go get github.com/ethereum/go-ethereum`, viene utilizzato per ottenere il pacchetto `go-ethereum` necessario per interagire con la blockchain Ethereum utilizzando il linguaggio di programmazione Go. Il secondo comando, `go get all`, viene utilizzato per ottenere tutte le dipendenze del progetto Go corrente, inclusi i pacchetti specificati nel file `go.mod`. Il terzo comando, `ego-go build smartContract.go`, viene utilizzato per compilare il file `smartContract.go` utilizzando EGo, che è un framework per lo sviluppo di applicazioni sicure basate

su Intel SGX. Il quarto comando, `ego sign smartContract`, firma l'eseguibile generato precedentemente per Intel SGX, garantendo l'integrità e l'autenticità del codice eseguibile all'interno dell'enclave. Il quinto comando, `cat enclave.json private.pem`, viene utilizzato per visualizzare la configurazione dell'enclave e la chiave privata associata al processo di firma dell'eseguibile per Intel SGX.
Questi comandi sono tipicamente utilizzati nello sviluppo di applicazioni blockchain che richiedono un elevato livello di sicurezza e protezione dei dati utilizzando la tecnologia Intel SGX. Puoi eseguire questi comandi da una shell bash per gestire gli Smart Contracts e garantire la sua esecuzione sicura all'interno di un ambiente protetto.

4.2.3 Codice Sorgente Solidity

Nel contesto della gestione dei dati sanitari, è fondamentale garantire la sicurezza, la privacy e la conformità normativa. Lo Smart Contract `HealthDataRegistry` implementa un registro sanitario su blockchain che rispetta le regole del GDPR e assicura la tracciabilità dei dati dei pazienti.

Lo smart contract è presentato nel listato 4.3.

```
1  pragma solidity ^0.8.0;
2
3  contract HealthDataRegistry {
4      address public owner;  // Indirizzo del proprietario dello
       ↪ Smart Contract
5      mapping(address => bool) public authorizedUsers;  // Mappa
       ↪ degli utenti autorizzati
6      mapping(address => mapping(bytes32 => bool)) public
       ↪ accessPermissions;  // Mappa dei permessi di accesso
7
8      // Struttura per memorizzare i dati sanitari
9      struct HealthRecord {
10         bytes32 hash;  // Hash dei dati sanitari
11         address patient;  // Indirizzo del paziente
12         bool isDeleted;  // Indica se il record    stato
       ↪ cancellato
13         uint hospitalEntryTime;  // Timestamp di entrata in
       ↪ ospedale
14         uint hospitalExitTime;  // Timestamp di uscita dall'
       ↪ ospedale
15         string currentDepartment;  // Reparto corrente
16     }
17
18     // Mappa dei record sanitari per ogni paziente
19     mapping(address => HealthRecord[]) public healthRecords;
20
21     // Eventi emessi dallo Smart Contract
22     event HealthRecordAdded(address indexed patient, bytes32
       ↪ indexed recordHash);
23     event HealthRecordDeleted(address indexed patient, bytes32
       ↪ indexed recordHash);
24     event HospitalEntry(address indexed patient, string
       ↪ department, uint entryTime);
25     event DepartmentChange(address indexed patient, string
       ↪ newDepartment, uint timestamp);
26     event HospitalExit(address indexed patient, uint exitTime);
27
```

```
28      // Modificatore: permette l'accesso solo al proprietario
    ↪ dello Smart Contract
29      modifier onlyOwner() {
30          require(msg.sender == owner, "Only contract owner can
    ↪ call this function");
31          _;
32      }
33
34      // Modificatore: permette l'accesso solo agli utenti
    ↪ autorizzati
35      modifier onlyAuthorized() {
36          require(authorizedUsers[msg.sender] == true, "Not
    ↪ authorized to access this function");
37          _;
38      }
39
40      // Costruttore dello Smart Contract
41      constructor() {
42          owner = msg.sender;  // L'indirizzo del chiamante diventa
    ↪  il proprietario
43      }
44
45      // Aggiunge un nuovo record sanitario per il paziente
46      function addHealthRecord(bytes32 _hash) public onlyAuthorized
    ↪  {
47          healthRecords[msg.sender].push(HealthRecord(_hash, msg.
    ↪ sender, false, 0, 0, ""));
48          emit HealthRecordAdded(msg.sender, _hash);
49      }
50
51      // Cancella un record sanitario per il paziente
52      function deleteHealthRecord(bytes32 _hash) public
    ↪ onlyAuthorized {
53          for(uint i = 0; i < healthRecords[msg.sender].length; i
    ↪ ++) {
54              if (healthRecords[msg.sender][i].hash == _hash) {
55                  healthRecords[msg.sender][i].isDeleted = true;
56                  emit HealthRecordDeleted(msg.sender, _hash);
57                  break;
58              }
59          }
60      }
61
62      // Concede accesso ad un determinato record a un utente
63      function grantAccess(address _user, bytes32 _hash) public
    ↪ onlyOwner {
64          accessPermissions[_user][_hash] = true;
65      }
66
67      // Revoca l'accesso ad un determinato record per un utente
68      function revokeAccess(address _user, bytes32 _hash) public
    ↪ onlyOwner {
69          accessPermissions[_user][_hash] = false;
70      }
71
72      // Autorizza un utente a utilizzare lo Smart Contract
73      function authorizeUser(address _user) public onlyOwner {
74          authorizedUsers[_user] = true;
```

```
75      }
76
77      // Deautorizza un utente a utilizzare lo Smart Contract
78      function deauthorizeUser(address _user) public onlyOwner {
79          authorizedUsers[_user] = false;
80      }
81
82      // Registra l'entrata in ospedale del paziente in un
    ↪ determinato reparto
83      function hospitalEntry(string memory _department) public
    ↪ onlyAuthorized {
84          HealthRecord[] storage records = healthRecords[msg.sender
    ↪ ];
85          records[records.length - 1].hospitalEntryTime = block.
    ↪ timestamp;
86          records[records.length - 1].currentDepartment =
    ↪ _department;
87          emit HospitalEntry(msg.sender, _department, block.
    ↪ timestamp);
88      }
89
90      // Cambia il reparto corrente del paziente
91      function changeDepartment(string memory _newDepartment)
    ↪ public onlyAuthorized {
92          HealthRecord[] storage records = healthRecords[msg.sender
    ↪ ];
93          records[records.length - 1].currentDepartment =
    ↪ _newDepartment;
94          emit DepartmentChange(msg.sender, _newDepartment, block.
    ↪ timestamp);
95      }
96
97      // Registra l'uscita dall'ospedale del paziente
98      function hospitalExit() public onlyAuthorized {
99          HealthRecord[] storage records = healthRecords[msg.sender
    ↪ ];
100         records[records.length - 1].hospitalExitTime = block.
    ↪ timestamp;
101         emit HospitalExit(msg.sender, block.timestamp);
102     }
103 }
```

Listing 4.3. Smart Contract che tiene traccia ogni passo nel percorso del paziente, garantendo sicurezza e privacy conforme al GDPR

Quando si tratta di dati sanitari e blockchain, è essenziale garantire la sicurezza, la privacy e la conformità normativa GDPR. Nello Smart Contrat `HealthDataRegistry`, diverse caratteristiche sono progettate per rispettare i principi del GDPR. Prima di essere pubblicati sulla blockchain, i dati sanitari devono essere anonimizzati o pseudonimizzati per proteggere l'identità degli individui. Nello smart contract, l'attributo `hash` nella struttura `HealthRecord` memorizza l'hash dei dati anziché i dati stessi, garantendo l'anonimizzazione. È necessario ottenere il consenso informato degli individui cui i dati sanitari si riferiscono. Le funzioni `authorizeUser` e `grantAccess` nello Smart Contract consentono al proprietario di autorizzare gli utenti e ottenere il loro consenso prima di accedere ai dati. Lo Smart Contract raccoglie solo i dati sanitari strettamente necessari per lo scopo previsto. La funzione `addHealthRecord` aggiunge un nuovo record sanitario con l'hash dei dati e altri dettagli essenziali, minimizzando la quantità di informazioni raccolte. Gli individui

hanno il diritto di richiedere la cancellazione dei loro dati se non vi sono più ragioni legittime per conservarli. La funzione `deleteHealthRecord` consente agli utenti autorizzati di cancellare i propri record sanitari dalla blockchain. È garantita una sicurezza adeguata per proteggere i dati sanitari dalla violazione e dall'accesso non autorizzato. L'attributo `owner` memorizza l'indirizzo del proprietario dello Smart Contract, che ha il controllo e la responsabilità principale sulla sicurezza dei dati. Chiunque controlli o tratti i dati sanitari sulla blockchain deve essere in grado di dimostrare la conformità al GDPR. Il proprietario dello Smart Contract ha la responsabilità principale e deve garantire la conformità normativa. Se i dati sanitari vengono trasferiti al di fuori dell'UE/SEE, devono essere garantite protezioni adeguate. Le funzioni `grantAccess` e `revokeAccess` nello Smart Contract gestiscono l'accesso ai dati in conformità con le leggi sulla protezione dei dati in diverse giurisdizioni. Deve essere definito chiaramente chi ha accesso ai dati sanitari sulla blockchain e per quali scopi. I modificatori nello Smart Contract garantiscono che solo il proprietario dello Smart Contract e gli utenti autorizzati possano accedere ai dati, con restrizioni appropriate.

```
pragma solidity ^0.8.0;

contract HealthDataRegistry {
    struct HealthRecord {
        bytes32 hash;
        address patient;
        bool isDeleted;
        uint hospitalEntryTime;
        uint hospitalExitTime;
        string currentDepartment;
        uint age;
        uint weight;
        string firstName;
        string lastName;
        string currentIllness;
        string[] pastIllnesses;
        string[] allergies;
    }

    mapping(address => HealthRecord[]) public healthRecords;

    function getHealthRecord(address _patient) external view
    ↪ returns (HealthRecord memory) {
        return healthRecords[_patient][healthRecords[_patient].
    ↪ length - 1];
    }
}
```

Listing 4.4. Smart Contract di registrazione dati personali del paziente.

La struttura `HealthRecord` rappresenta un record sanitario e contiene informazioni quali l'hash del record, l'indirizzo del paziente, lo stato di eliminazione del record, i timestamp di ingresso e uscita dall'ospedale, il dipartimento attuale del paziente, l'età, il peso, il nome, il cognome, la malattia attuale, l'elenco delle malattie passate e l'elenco delle allergie.

Il mapping `healthRecords` associa un array di HealthRecord a ciascun indirizzo del paziente. Ciò consente di memorizzare più record sanitari per ogni paziente e di recuperarli in base al loro indirizzo.

La funzione `getHealthRecord` consente di ottenere l'ultimo record sanitario di un paziente specificato. Prende in input l'indirizzo del paziente e restituisce il record sanitario più recente presente nell'array dei suoi record sanitari. Utilizza la keyword external per indicare che può essere chiamata solo da Smart Contract esterni e non all'interno dello Smart Contract stesso. La keyword `view` indica che la funzione non modifica lo stato dello Smart Contract e può essere chiamata gratuitamente senza consumare gas.

```
1
2  pragma solidity ^0.8.0;
3
4  contract MedicalDataCalculator {
5      address public healthDataRegistryAddress;
6
7      constructor(address _healthDataRegistryAddress) {
8          healthDataRegistryAddress = _healthDataRegistryAddress;
9      }
10
11     function calculateInsulinDosage() public view returns (uint)
       ↪ {
12     // Ottieni i dati sanitari del paziente dall'
       ↪ HealthDataRegistry
13     HealthDataRegistry healthDataRegistry = HealthDataRegistry(
       ↪ healthDataRegistryAddress);
14     HealthDataRegistry.HealthRecord memory healthRecord =
       ↪ healthDataRegistry.getHealthRecord(msg.sender);
15
16     // Calcolo semplificato del dosaggio dell'insulina in IU
17     uint insulinDosage = 0;
18
19     // Calcolo basato sull'et  e sul peso del paziente
20     // Puoi aggiustare i fattori moltiplicativi in base alle
       ↪ linee guida cliniche
21     insulinDosage += healthRecord.age * healthRecord.weight /
       ↪ 100;
22
23     // Restituisci il dosaggio calcolato
24     return insulinDosage;
25     }
26 }
```

Listing 4.5. Smart Contract eseguito in TEE.

Questo Smart Contract, chiamato `MedicalDataCalculator`, è progettato per calcolare il dosaggio di insulina per un paziente utilizzando i dati sanitari memorizzati nello Smart Contract `HealthDataRegistry`.

La variabile `healthDataRegistryAddress` di stato pubblica di tipo address conterrà l'indirizzo dello Smart Contract `HealthDataRegistry`. Questo indirizzo verrà utilizzato per interagire con lo Smart Contract `HealthDataRegistry`. Il costruttore dello Smart Contract MedicalDataCalculator. Viene chiamato una sola volta al momento della creazione dello Smart Contract e inizializza la variabile `healthDataRegistryAddress` con l'indirizzo dello Smart Contract `HealthDataRegistry` passato come parametro.

La funzione `calculateInsulinDosage` calcola il dosaggio di insulina per il paziente. La funzione è pubblica e dichiarata con il modificatore `view`, il che significa che non modifica lo stato dello Smart Contract e può essere chiamata gratuitamente senza consumare gas. Successivamente viene istanziato un oggetto

`HealthDataRegistry` utilizzando l'indirizzo dello Smart Contract `HealthDataRegistry`. Viene ottenuto il record sanitario del paziente chiamante utilizzando la funzione `getHealthRecord` dello Smart Contract `HealthDataRegistry`. Viene eseguito un calcolo semplificato del dosaggio di insulina basato sull'età e sul peso del paziente. Questo calcolo potrebbe essere modificato per adattarsi a linee guida cliniche specifiche. Il dosaggio calcolato viene restituito come risultato della funzione. Questo Smart Contract consente di calcolare il dosaggio di insulina per un paziente utilizzando i dati sanitari memorizzati nello Smart Contract `HealthDataRegistry`, fornendo un esempio di come sia possibile integrare più Smart Contracts per eseguire operazioni complesse basate su dati sanitari.

Gli Smart Contract però verranno convertiti e usati all'interno della TEE utilizzando `Golang`, `HealthDataRegistry` apparirà come segue:

```
// Code generated - DO NOT EDIT.
// This file is a generated binding and any manual changes will
    ↪ be lost.

package main

import (
  "errors"
  "math/big"
  "strings"

  ethereum "github.com/ethereum/go-ethereum"
  "github.com/ethereum/go-ethereum/accounts/abi"
  "github.com/ethereum/go-ethereum/accounts/abi/bind"
  "github.com/ethereum/go-ethereum/common"
  "github.com/ethereum/go-ethereum/core/types"
  "github.com/ethereum/go-ethereum/event"
)

// Reference imports to suppress errors if they are not otherwise
    ↪  used.
var (
  _ = errors.New
  _ = big.NewInt
  _ = strings.NewReader
  _ = ethereum.NotFound
  _ = bind.Bind
  _ = common.Big1
  _ = types.BloomLookup
  _ = event.NewSubscription
  _ = abi.ConvertType
)

// HealthDataRegistryHealthRecord is an auto generated low-level
    ↪ Go binding around an user-defined struct.
type HealthDataRegistryHealthRecord struct {
  Hash              [32]byte
  Patient           common.Address
  IsDeleted         bool
  HospitalEntryTime *big.Int
  HospitalExitTime  *big.Int
  CurrentDepartment string
  Age               *big.Int
  Weight            *big.Int
```

```
42    FirstName          string
43    LastName           string
44    CurrentIllness     string
45    PastIllnesses      []string
46    Allergies          []string
47  }
48
49  // StorageMetaData contains all meta data concerning the Storage
      ↪ contract.
50  var StorageMetaData = &bind.MetaData{
51    ABI: "[{\"inputs\":[{\"internalType\":\"address\",\"name\":\"
      ↪ _patient\",\"type\":\"address\"}],\"name\":\"
      ↪ getHealthRecord\",\"outputs\":[{\"components\":[{\"
      ↪ internalType\":\"bytes32\",\"name\":\"hash\",\"type\":\"
      ↪ bytes32\"},{\"internalType\":\"address\",\"name\":\"patient
      ↪ \",\"type\":\"address\"},{\"internalType\":\"bool\",\"name
      ↪ \":\"isDeleted\",\"type\":\"bool\"},{\"internalType\":\"
      ↪ uint256\",\"name\":\"hospitalEntryTime\",\"type\":\"uint256
      ↪ \"},{\"internalType\":\"uint256\",\"name\":\"
      ↪ hospitalExitTime\",\"type\":\"uint256\"},{\"internalType
      ↪ \":\"string\",\"name\":\"currentDepartment\",\"type\":\"
      ↪ string\"},{\"internalType\":\"uint256\",\"name\":\"age\",\"
      ↪ type\":\"uint256\"},{\"internalType\":\"uint256\",\"name
      ↪ \":\"weight\",\"type\":\"uint256\"},{\"internalType\":\"
      ↪ string\",\"name\":\"firstName\",\"type\":\"string\"},{\"
      ↪ internalType\":\"string\",\"name\":\"lastName\",\"type\":\"
      ↪ string\"},{\"internalType\":\"string\",\"name\":\"
      ↪ currentIllness\",\"type\":\"string\"},{\"internalType\":\"
      ↪ string[]\",\"name\":\"pastIllnesses\",\"type\":\"string
      ↪ []\"},{\"internalType\":\"string[]\",\"name\":\"allergies
      ↪ \",\"type\":\"string[]\"}],\"internalType\":\"
      ↪ structHealthDataRegistry.HealthRecord\",\"name\":\"\",\"
      ↪ type\":\"tuple\"}],\"stateMutability\":\"view\",\"type\":\"
      ↪ function\"},{\"inputs\":[{\"internalType\":\"address\",\"
      ↪ name\":\"\",\"type\":\"address\"},{\"internalType\":\"
      ↪ uint256\",\"name\":\"\",\"type\":\"uint256\"}],\"name\":\"
      ↪ healthRecords\",\"outputs\":[{\"internalType\":\"bytes32
      ↪ \",\"name\":\"hash\",\"type\":\"bytes32\"},{\"internalType
      ↪ \":\"address\",\"name\":\"patient\",\"type\":\"address
      ↪ \"},{\"internalType\":\"bool\",\"name\":\"isDeleted\",\"
      ↪ type\":\"bool\"},{\"internalType\":\"uint256\",\"name\":\"
      ↪ hospitalEntryTime\",\"type\":\"uint256\"},{\"internalType
      ↪ \":\"uint256\",\"name\":\"hospitalExitTime\",\"type\":\"
      ↪ uint256\"},{\"internalType\":\"string\",\"name\":\"
      ↪ currentDepartment\",\"type\":\"string\"},{\"internalType
      ↪ \":\"uint256\",\"name\":\"age\",\"type\":\"uint256\"},{\"
      ↪ internalType\":\"uint256\",\"name\":\"weight\",\"type\":\"
      ↪ uint256\"},{\"internalType\":\"string\",\"name\":\"
      ↪ firstName\",\"type\":\"string\"},{\"internalType\":\"string
      ↪ \",\"name\":\"lastName\",\"type\":\"string\"},{\"
      ↪ internalType\":\"string\",\"name\":\"currentIllness\",\"
      ↪ type\":\"string\"}],\"stateMutability\":\"view\",\"type
      ↪ \":\"function\"}]",
52  }
53
54  // StorageABI is the input ABI used to generate the binding from.
55  // Deprecated: Use StorageMetaData.ABI instead.
56  var StorageABI = StorageMetaData.ABI
```

```
57
58 // Storage is an auto generated Go binding around an Ethereum
    ↪ contract.
59 type Storage struct {
60   StorageCaller     // Read-only binding to the contract
61   StorageTransactor // Write-only binding to the contract
62   StorageFilterer   // Log filterer for contract events
63 }
64
65 // StorageCaller is an auto generated read-only Go binding around
    ↪  an Ethereum contract.
66 type StorageCaller struct {
67   contract *bind.BoundContract // Generic contract wrapper for
    ↪ the low level calls
68 }
69
70 // StorageTransactor is an auto generated write-only Go binding
    ↪ around an Ethereum contract.
71 type StorageTransactor struct {
72   contract *bind.BoundContract // Generic contract wrapper for
    ↪ the low level calls
73 }
74
75 // StorageFilterer is an auto generated log filtering Go binding
    ↪ around an Ethereum contract events.
76 type StorageFilterer struct {
77   contract *bind.BoundContract // Generic contract wrapper for
    ↪ the low level calls
78 }
79
80 // StorageSession is an auto generated Go binding around an
    ↪ Ethereum contract,
81 // with pre-set call and transact options.
82 type StorageSession struct {
83   Contract     *Storage          // Generic contract binding to
    ↪ set the session for
84   CallOpts     bind.CallOpts     // Call options to use
    ↪ throughout this session
85   TransactOpts bind.TransactOpts // Transaction auth options to
    ↪ use throughout this session
86 }
87
88 // StorageCallerSession is an auto generated read-only Go binding
    ↪  around an Ethereum contract,
89 // with pre-set call options.
90 type StorageCallerSession struct {
91   Contract *StorageCaller // Generic contract caller binding to
    ↪ set the session for
92   CallOpts bind.CallOpts  // Call options to use throughout this
    ↪ session
93 }
94
95 // StorageTransactorSession is an auto generated write-only Go
    ↪ binding around an Ethereum contract,
96 // with pre-set transact options.
97 type StorageTransactorSession struct {
98   Contract     *StorageTransactor // Generic contract transactor
    ↪ binding to set the session for
```

```
99    TransactOpts bind.TransactOpts  // Transaction auth options to
        ↪ use throughout this session
100 }
101
102 // StorageRaw is an auto generated low-level Go binding around an
        ↪  Ethereum contract.
103 type StorageRaw struct {
104   Contract *Storage // Generic contract binding to access the raw
        ↪  methods on
105 }
106
107 // StorageCallerRaw is an auto generated low-level read-only Go
        ↪ binding around an Ethereum contract.
108 type StorageCallerRaw struct {
109   Contract *StorageCaller // Generic read-only contract binding
        ↪ to access the raw methods on
110 }
111
112 // StorageTransactorRaw is an auto generated low-level write-only
        ↪  Go binding around an Ethereum contract.
113 type StorageTransactorRaw struct {
114   Contract *StorageTransactor // Generic write-only contract
        ↪ binding to access the raw methods on
115 }
116
117 // NewStorage creates a new instance of Storage, bound to a
        ↪ specific deployed contract.
118 func NewStorage(address common.Address, backend bind.
        ↪ ContractBackend) (*Storage, error) {
119   contract, err := bindStorage(address, backend, backend, backend
        ↪ )
120   if err != nil {
121     return nil, err
122   }
123   return &Storage{StorageCaller: StorageCaller{contract: contract
        ↪ }, StorageTransactor: StorageTransactor{contract: contract
        ↪ }, StorageFilterer: StorageFilterer{contract: contract}},
        ↪ nil
124 }
125
126 // NewStorageCaller creates a new read-only instance of Storage,
        ↪ bound to a specific deployed contract.
127 func NewStorageCaller(address common.Address, caller bind.
        ↪ ContractCaller) (*StorageCaller, error) {
128   contract, err := bindStorage(address, caller, nil, nil)
129   if err != nil {
130     return nil, err
131   }
132   return &StorageCaller{contract: contract}, nil
133 }
134
135 // NewStorageTransactor creates a new write-only instance of
        ↪ Storage, bound to a specific deployed contract.
136 func NewStorageTransactor(address common.Address, transactor bind
        ↪ .ContractTransactor) (*StorageTransactor, error) {
137   contract, err := bindStorage(address, nil, transactor, nil)
138   if err != nil {
139     return nil, err
```

```
140   }
141   return &StorageTransactor{contract: contract}, nil
142 }
143
144 // NewStorageFilterer creates a new log filterer instance of
    ↪ Storage, bound to a specific deployed contract.
145 func NewStorageFilterer(address common.Address, filterer bind.
    ↪ ContractFilterer) (*StorageFilterer, error) {
146   contract, err := bindStorage(address, nil, nil, filterer)
147   if err != nil {
148     return nil, err
149   }
150   return &StorageFilterer{contract: contract}, nil
151 }
152
153 // bindStorage binds a generic wrapper to an already deployed
    ↪ contract.
154 func bindStorage(address common.Address, caller bind.
    ↪ ContractCaller, transactor bind.ContractTransactor,
    ↪ filterer bind.ContractFilterer) (*bind.BoundContract, error
    ↪ ) {
155   parsed, err := StorageMetaData.GetAbi()
156   if err != nil {
157     return nil, err
158   }
159   return bind.NewBoundContract(address, *parsed, caller,
    ↪ transactor, filterer), nil
160 }
161
162 // Call invokes the (constant) contract method with params as
    ↪ input values and
163 // sets the output to result. The result type might be a single
    ↪ field for simple
164 // returns, a slice of interfaces for anonymous returns and a
    ↪ struct for named
165 // returns.
166 func (_Storage *StorageRaw) Call(opts *bind.CallOpts, result *[]
    ↪ interface{}, method string, params ...interface{}) error {
167   return _Storage.Contract.StorageCaller.contract.Call(opts,
    ↪ result, method, params...)
168 }
169
170 // Transfer initiates a plain transaction to move funds to the
    ↪ contract, calling
171 // its default method if one is available.
172 func (_Storage *StorageRaw) Transfer(opts *bind.TransactOpts) (*
    ↪ types.Transaction, error) {
173   return _Storage.Contract.StorageTransactor.contract.Transfer(
    ↪ opts)
174 }
175
176 // Transact invokes the (paid) contract method with params as
    ↪ input values.
177 func (_Storage *StorageRaw) Transact(opts *bind.TransactOpts,
    ↪ method string, params ...interface{}) (*types.Transaction,
    ↪ error) {
178   return _Storage.Contract.StorageTransactor.contract.Transact(
    ↪ opts, method, params...)
```

```
179 }
180
181 // Call invokes the (constant) contract method with params as
    ↪ input values and
182 // sets the output to result. The result type might be a single
    ↪ field for simple
183 // returns, a slice of interfaces for anonymous returns and a
    ↪ struct for named
184 // returns.
185 func (_Storage *StorageCallerRaw) Call(opts *bind.CallOpts,
    ↪ result *[]interface{}, method string, params ...interface
    ↪ {}) error {
186   return _Storage.Contract.contract.Call(opts, result, method,
    ↪ params...)
187 }
188
189 // Transfer initiates a plain transaction to move funds to the
    ↪ contract, calling
190 // its default method if one is available.
191 func (_Storage *StorageTransactorRaw) Transfer(opts *bind.
    ↪ TransactOpts) (*types.Transaction, error) {
192   return _Storage.Contract.contract.Transfer(opts)
193 }
194
195 // Transact invokes the (paid) contract method with params as
    ↪ input values.
196 func (_Storage *StorageTransactorRaw) Transact(opts *bind.
    ↪ TransactOpts, method string, params ...interface{}) (*types
    ↪ .Transaction, error) {
197   return _Storage.Contract.contract.Transact(opts, method, params
    ↪ ...)
198 }
199
200 // GetHealthRecord is a free data retrieval call binding the
    ↪ contract method 0xbd8f1752.
201 //
202 // Solidity: function getHealthRecord(address _patient) view
    ↪ returns((bytes32,address,bool,uint256,uint256,string,
    ↪ uint256,uint256,string,string,string,string[],string[]))
203 func (_Storage *StorageCaller) GetHealthRecord(opts *bind.
    ↪ CallOpts, _patient common.Address) (
    ↪ HealthDataRegistryHealthRecord, error) {
204   var out []interface{}
205   err := _Storage.contract.Call(opts, &out, "getHealthRecord",
    ↪ _patient)
206
207   if err != nil {
208     return *new(HealthDataRegistryHealthRecord), err
209   }
210
211   out0 := *abi.ConvertType(out[0], new(
    ↪ HealthDataRegistryHealthRecord)).(*
    ↪ HealthDataRegistryHealthRecord)
212
213   return out0, err
214
215 }
216
```

```
217 // GetHealthRecord is a free data retrieval call binding the
    ↪ contract method 0xbd8f1752.
218 //
219 // Solidity: function getHealthRecord(address _patient) view
    ↪ returns((bytes32,address,bool,uint256,uint256,string,
    ↪ uint256,uint256,string,string,string,string[],string[]))
220 func (_Storage *StorageSession) GetHealthRecord(_patient common.
    ↪ Address) (HealthDataRegistryHealthRecord, error) {
221   return _Storage.Contract.GetHealthRecord(&_Storage.CallOpts,
    ↪ _patient)
222 }
223
224 // GetHealthRecord is a free data retrieval call binding the
    ↪ contract method 0xbd8f1752.
225 //
226 // Solidity: function getHealthRecord(address _patient) view
    ↪ returns((bytes32,address,bool,uint256,uint256,string,
    ↪ uint256,uint256,string,string,string,string[],string[]))
227 func (_Storage *StorageCallerSession) GetHealthRecord(_patient
    ↪ common.Address) (HealthDataRegistryHealthRecord, error) {
228   return _Storage.Contract.GetHealthRecord(&_Storage.CallOpts,
    ↪ _patient)
229 }
230
231 // HealthRecords is a free data retrieval call binding the
    ↪ contract method 0x85b852c1.
232 //
233 // Solidity: function healthRecords(address , uint256 ) view
    ↪ returns(bytes32 hash, address patient, bool isDeleted,
    ↪ uint256 hospitalEntryTime, uint256 hospitalExitTime, string
    ↪  currentDepartment, uint256 age, uint256 weight, string
    ↪ firstName, string lastName, string currentIllness)
234 func (_Storage *StorageCaller) HealthRecords(opts *bind.CallOpts,
    ↪  arg0 common.Address, arg1 *big.Int) (struct {
235   Hash              [32]byte
236   Patient           common.Address
237   IsDeleted         bool
238   HospitalEntryTime *big.Int
239   HospitalExitTime  *big.Int
240   CurrentDepartment string
241   Age               *big.Int
242   Weight            *big.Int
243   FirstName         string
244   LastName          string
245   CurrentIllness    string
246 }, error) {
247   var out []interface{}
248   err := _Storage.contract.Call(opts, &out, "healthRecords", arg0
    ↪ , arg1)
249
250   outstruct := new(struct {
251     Hash              [32]byte
252     Patient           common.Address
253     IsDeleted         bool
254     HospitalEntryTime *big.Int
255     HospitalExitTime  *big.Int
256     CurrentDepartment string
257     Age               *big.Int
```

```
258     Weight            *big.Int
259     FirstName         string
260     LastName          string
261     CurrentIllness    string
262   })
263   if err != nil {
264     return *outstruct, err
265   }
266
267   outstruct.Hash = *abi.ConvertType(out[0], new([32]byte)).(*[32]
        ↪ byte)
268   outstruct.Patient = *abi.ConvertType(out[1], new(common.Address
        ↪ )).(*common.Address)
269   outstruct.IsDeleted = *abi.ConvertType(out[2], new(bool)).(*
        ↪ bool)
270   outstruct.HospitalEntryTime = *abi.ConvertType(out[3], new(*big
        ↪ .Int)).(**big.Int)
271   outstruct.HospitalExitTime = *abi.ConvertType(out[4], new(*big.
        ↪ Int)).(**big.Int)
272   outstruct.CurrentDepartment = *abi.ConvertType(out[5], new(
        ↪ string)).(*string)
273   outstruct.Age = *abi.ConvertType(out[6], new(*big.Int)).(**big.
        ↪ Int)
274   outstruct.Weight = *abi.ConvertType(out[7], new(*big.Int)).(**
        ↪ big.Int)
275   outstruct.FirstName = *abi.ConvertType(out[8], new(string)).(*
        ↪ string)
276   outstruct.LastName = *abi.ConvertType(out[9], new(string)).(*
        ↪ string)
277   outstruct.CurrentIllness = *abi.ConvertType(out[10], new(string
        ↪ )).(*string)
278
279   return *outstruct, err
280
281 }
282
283 // HealthRecords is a free data retrieval call binding the
        ↪ contract method 0x85b852c1.
284 //
285 // Solidity: function healthRecords(address , uint256 ) view
        ↪ returns(bytes32 hash, address patient, bool isDeleted,
        ↪ uint256 hospitalEntryTime, uint256 hospitalExitTime, string
        ↪  currentDepartment, uint256 age, uint256 weight, string
        ↪ firstName, string lastName, string currentIllness)
286 func (_Storage *StorageSession) HealthRecords(arg0 common.Address
        ↪ , arg1 *big.Int) (struct {
287   Hash              [32]byte
288   Patient           common.Address
289   IsDeleted         bool
290   HospitalEntryTime *big.Int
291   HospitalExitTime  *big.Int
292   CurrentDepartment string
293   Age               *big.Int
294   Weight            *big.Int
295   FirstName         string
296   LastName          string
297   CurrentIllness    string
298 }, error) {
```

```
299   return _Storage.Contract.HealthRecords(&_Storage.CallOpts, arg0
      ↪ , arg1)
300 }
301
302 // HealthRecords is a free data retrieval call binding the
      ↪ contract method 0x85b852c1.
303 //
304 // Solidity: function healthRecords(address , uint256 ) view
      ↪ returns(bytes32 hash, address patient, bool isDeleted,
      ↪ uint256 hospitalEntryTime, uint256 hospitalExitTime, string
      ↪  currentDepartment, uint256 age, uint256 weight, string
      ↪ firstName, string lastName, string currentIllness)
305 func (_Storage *StorageCallerSession) HealthRecords(arg0 common.
      ↪ Address, arg1 *big.Int) (struct {
306   Hash               [32]byte
307   Patient            common.Address
308   IsDeleted          bool
309   HospitalEntryTime  *big.Int
310   HospitalExitTime   *big.Int
311   CurrentDepartment  string
312   Age                *big.Int
313   Weight             *big.Int
314   FirstName          string
315   LastName           string
316   CurrentIllness     string
317 }, error) {
318   return _Storage.Contract.HealthRecords(&_Storage.CallOpts, arg0
      ↪ , arg1)
319 }
```

Listing 4.6. Smart Contract HealthDataRegistry.go eseguito in TEE.

Mentre `MedicalDataCalculator.go` apparirà:

```
1
2  // Code generated - DO NOT EDIT.
3  // This file is a generated binding and any manual changes will
       ↪ be lost.
4
5  package main
6
7  import (
8    "errors"
9    "math/big"
10   "strings"
11
12   ethereum "github.com/ethereum/go-ethereum"
13   "github.com/ethereum/go-ethereum/accounts/abi"
14   "github.com/ethereum/go-ethereum/accounts/abi/bind"
15   "github.com/ethereum/go-ethereum/common"
16   "github.com/ethereum/go-ethereum/core/types"
17   "github.com/ethereum/go-ethereum/event"
18 )
19
20 // Reference imports to suppress errors if they are not otherwise
       ↪  used.
21 var (
22   _ = errors.New
23   _ = big.NewInt
24   _ = strings.NewReader
```

```
25    _ = ethereum.NotFound
26    _ = bind.Bind
27    _ = common.Big1
28    _ = types.BloomLookup
29    _ = event.NewSubscription
30    _ = abi.ConvertType
31  )
32
33  // StorageMetaData contains all meta data concerning the Storage
        ↪ contract.
34  var StorageMetaData = &bind.MetaData{
35    ABI: "[{\"inputs\":[{\"internalType\":\"address\",\"name\":\"
        ↪ _healthDataRegistryAddress\",\"type\":\"address\"}],\"
        ↪ stateMutability\":\"nonpayable\",\"type\":\"constructor
        ↪ \"},{\"inputs\":[],\"name\":\"calculateInsulinDosage\",\"
        ↪ outputs\":[{\"internalType\":\"uint256\",\"name\":\"\",\"
        ↪ type\":\"uint256\"}],\"stateMutability\":\"view\",\"type
        ↪ \":\"function\"},{\"inputs\":[],\"name\":\"
        ↪ healthDataRegistryAddress\",\"outputs\":[{\"internalType
        ↪ \":\"address\",\"name\":\"\",\"type\":\"address\"}],\"
        ↪ stateMutability\":\"view\",\"type\":\"function\"}]",
36  }
37
38  // StorageABI is the input ABI used to generate the binding from.
39  // Deprecated: Use StorageMetaData.ABI instead.
40  var StorageABI = StorageMetaData.ABI
41
42  // Storage is an auto generated Go binding around an Ethereum
        ↪ contract.
43  type Storage struct {
44    StorageCaller     // Read-only binding to the contract
45    StorageTransactor // Write-only binding to the contract
46    StorageFilterer   // Log filterer for contract events
47  }
48
49  // StorageCaller is an auto generated read-only Go binding around
        ↪  an Ethereum contract.
50  type StorageCaller struct {
51    contract *bind.BoundContract // Generic contract wrapper for
        ↪ the low level calls
52  }
53
54  // StorageTransactor is an auto generated write-only Go binding
        ↪ around an Ethereum contract.
55  type StorageTransactor struct {
56    contract *bind.BoundContract // Generic contract wrapper for
        ↪ the low level calls
57  }
58
59  // StorageFilterer is an auto generated log filtering Go binding
        ↪ around an Ethereum contract events.
60  type StorageFilterer struct {
61    contract *bind.BoundContract // Generic contract wrapper for
        ↪ the low level calls
62  }
63
64  // StorageSession is an auto generated Go binding around an
        ↪ Ethereum contract,
```

```
65 // with pre-set call and transact options.
66 type StorageSession struct {
67   Contract     *Storage          // Generic contract binding to
       ↪ set the session for
68   CallOpts     bind.CallOpts     // Call options to use
       ↪ throughout this session
69   TransactOpts bind.TransactOpts // Transaction auth options to
       ↪ use throughout this session
70 }
71
72 // StorageCallerSession is an auto generated read-only Go binding
       ↪  around an Ethereum contract,
73 // with pre-set call options.
74 type StorageCallerSession struct {
75   Contract *StorageCaller // Generic contract caller binding to
       ↪ set the session for
76   CallOpts bind.CallOpts  // Call options to use throughout this
       ↪ session
77 }
78
79 // StorageTransactorSession is an auto generated write-only Go
       ↪ binding around an Ethereum contract,
80 // with pre-set transact options.
81 type StorageTransactorSession struct {
82   Contract     *StorageTransactor // Generic contract transactor
       ↪ binding to set the session for
83   TransactOpts bind.TransactOpts  // Transaction auth options to
       ↪ use throughout this session
84 }
85
86 // StorageRaw is an auto generated low-level Go binding around an
       ↪  Ethereum contract.
87 type StorageRaw struct {
88   Contract *Storage // Generic contract binding to access the raw
       ↪  methods on
89 }
90
91 // StorageCallerRaw is an auto generated low-level read-only Go
       ↪ binding around an Ethereum contract.
92 type StorageCallerRaw struct {
93   Contract *StorageCaller // Generic read-only contract binding
       ↪ to access the raw methods on
94 }
95
96 // StorageTransactorRaw is an auto generated low-level write-only
       ↪  Go binding around an Ethereum contract.
97 type StorageTransactorRaw struct {
98   Contract *StorageTransactor // Generic write-only contract
       ↪ binding to access the raw methods on
99 }
100
101 // NewStorage creates a new instance of Storage, bound to a
       ↪ specific deployed contract.
102 func NewStorage(address common.Address, backend bind.
       ↪ ContractBackend) (*Storage, error) {
103   contract, err := bindStorage(address, backend, backend, backend
       ↪ )
104   if err != nil {
```

```
105     return nil, err
106   }
107   return &Storage{StorageCaller: StorageCaller{contract: contract
        ↪ }, StorageTransactor: StorageTransactor{contract: contract
        ↪ }, StorageFilterer: StorageFilterer{contract: contract}},
        ↪ nil
108 }
109
110 // NewStorageCaller creates a new read-only instance of Storage,
        ↪ bound to a specific deployed contract.
111 func NewStorageCaller(address common.Address, caller bind.
        ↪ ContractCaller) (*StorageCaller, error) {
112   contract, err := bindStorage(address, caller, nil, nil)
113   if err != nil {
114     return nil, err
115   }
116   return &StorageCaller{contract: contract}, nil
117 }
118
119 // NewStorageTransactor creates a new write-only instance of
        ↪ Storage, bound to a specific deployed contract.
120 func NewStorageTransactor(address common.Address, transactor bind
        ↪ .ContractTransactor) (*StorageTransactor, error) {
121   contract, err := bindStorage(address, nil, transactor, nil)
122   if err != nil {
123     return nil, err
124   }
125   return &StorageTransactor{contract: contract}, nil
126 }
127
128 // NewStorageFilterer creates a new log filterer instance of
        ↪ Storage, bound to a specific deployed contract.
129 func NewStorageFilterer(address common.Address, filterer bind.
        ↪ ContractFilterer) (*StorageFilterer, error) {
130   contract, err := bindStorage(address, nil, nil, filterer)
131   if err != nil {
132     return nil, err
133   }
134   return &StorageFilterer{contract: contract}, nil
135 }
136
137 // bindStorage binds a generic wrapper to an already deployed
        ↪ contract.
138 func bindStorage(address common.Address, caller bind.
        ↪ ContractCaller, transactor bind.ContractTransactor,
        ↪ filterer bind.ContractFilterer) (*bind.BoundContract, error
        ↪ ) {
139   parsed, err := StorageMetaData.GetAbi()
140   if err != nil {
141     return nil, err
142   }
143   return bind.NewBoundContract(address, *parsed, caller,
        ↪ transactor, filterer), nil
144 }
145
146 // Call invokes the (constant) contract method with params as
        ↪ input values and
147 // sets the output to result. The result type might be a single
```

```
    ↪ field for simple
148 // returns, a slice of interfaces for anonymous returns and a
    ↪ struct for named
149 // returns.
150 func (_Storage *StorageRaw) Call(opts *bind.CallOpts, result *[]
    ↪ interface{}, method string, params ...interface{}) error {
151   return _Storage.Contract.StorageCaller.contract.Call(opts,
    ↪ result, method, params...)
152 }
153
154 // Transfer initiates a plain transaction to move funds to the
    ↪ contract, calling
155 // its default method if one is available.
156 func (_Storage *StorageRaw) Transfer(opts *bind.TransactOpts) (*
    ↪ types.Transaction, error) {
157   return _Storage.Contract.StorageTransactor.contract.Transfer(
    ↪ opts)
158 }
159
160 // Transact invokes the (paid) contract method with params as
    ↪ input values.
161 func (_Storage *StorageRaw) Transact(opts *bind.TransactOpts,
    ↪ method string, params ...interface{}) (*types.Transaction,
    ↪ error) {
162   return _Storage.Contract.StorageTransactor.contract.Transact(
    ↪ opts, method, params...)
163 }
164
165 // Call invokes the (constant) contract method with params as
    ↪ input values and
166 // sets the output to result. The result type might be a single
    ↪ field for simple
167 // returns, a slice of interfaces for anonymous returns and a
    ↪ struct for named
168 // returns.
169 func (_Storage *StorageCallerRaw) Call(opts *bind.CallOpts,
    ↪ result *[]interface{}, method string, params ...interface
    ↪ {}) error {
170   return _Storage.Contract.contract.Call(opts, result, method,
    ↪ params...)
171 }
172
173 // Transfer initiates a plain transaction to move funds to the
    ↪ contract, calling
174 // its default method if one is available.
175 func (_Storage *StorageTransactorRaw) Transfer(opts *bind.
    ↪ TransactOpts) (*types.Transaction, error) {
176   return _Storage.Contract.contract.Transfer(opts)
177 }
178
179 // Transact invokes the (paid) contract method with params as
    ↪ input values.
180 func (_Storage *StorageTransactorRaw) Transact(opts *bind.
    ↪ TransactOpts, method string, params ...interface{}) (*types
    ↪ .Transaction, error) {
181   return _Storage.Contract.contract.Transact(opts, method, params
    ↪ ...)
182 }
```

```
183
184 // CalculateInsulinDosage is a free data retrieval call binding
    ↪ the contract method 0xc05b6e58.
185 //
186 // Solidity: function calculateInsulinDosage() view returns(
    ↪ uint256)
187 func (_Storage *StorageCaller) CalculateInsulinDosage(opts *bind.
    ↪ CallOpts) (*big.Int, error) {
188   var out []interface{}
189   err := _Storage.contract.Call(opts, &out, "
    ↪ calculateInsulinDosage")
190
191   if err != nil {
192     return *new(*big.Int), err
193   }
194
195   out0 := *abi.ConvertType(out[0], new(*big.Int)).(**big.Int)
196
197   return out0, err
198
199 }
200
201 // CalculateInsulinDosage is a free data retrieval call binding
    ↪ the contract method 0xc05b6e58.
202 //
203 // Solidity: function calculateInsulinDosage() view returns(
    ↪ uint256)
204 func (_Storage *StorageSession) CalculateInsulinDosage() (*big.
    ↪ Int, error) {
205   return _Storage.Contract.CalculateInsulinDosage(&_Storage.
    ↪ CallOpts)
206 }
207
208 // CalculateInsulinDosage is a free data retrieval call binding
    ↪ the contract method 0xc05b6e58.
209 //
210 // Solidity: function calculateInsulinDosage() view returns(
    ↪ uint256)
211 func (_Storage *StorageCallerSession) CalculateInsulinDosage() (*
    ↪ big.Int, error) {
212   return _Storage.Contract.CalculateInsulinDosage(&_Storage.
    ↪ CallOpts)
213 }
214
215 // HealthDataRegistryAddress is a free data retrieval call
    ↪ binding the contract method 0x08af1d3d.
216 //
217 // Solidity: function healthDataRegistryAddress() view returns(
    ↪ address)
218 func (_Storage *StorageCaller) HealthDataRegistryAddress(opts *
    ↪ bind.CallOpts) (common.Address, error) {
219   var out []interface{}
220   err := _Storage.contract.Call(opts, &out, "
    ↪ healthDataRegistryAddress")
221
222   if err != nil {
223     return *new(common.Address), err
224   }
```

```
225
226   out0 := *abi.ConvertType(out[0], new(common.Address)).(*common.
        ↪ Address)
227
228   return out0, err
229
230 }
231
232 // HealthDataRegistryAddress is a free data retrieval call
        ↪ binding the contract method 0x08af1d3d.
233 //
234 // Solidity: function healthDataRegistryAddress() view returns(
        ↪ address)
235 func (_Storage *StorageSession) HealthDataRegistryAddress() (
        ↪ common.Address, error) {
236   return _Storage.Contract.HealthDataRegistryAddress(&_Storage.
        ↪ CallOpts)
237 }
238
239 // HealthDataRegistryAddress is a free data retrieval call
        ↪ binding the contract method 0x08af1d3d.
240 //
241 // Solidity: function healthDataRegistryAddress() view returns(
        ↪ address)
242 func (_Storage *StorageCallerSession) HealthDataRegistryAddress()
        ↪   (common.Address, error) {
243   return _Storage.Contract.HealthDataRegistryAddress(&_Storage.
        ↪ CallOpts)
244 }
```

Listing 4.7. Smart Contract MedicalDataCalculator.go eseguito in TEE.

Capitolo 5

Discussione e prospettive future

Il capitolo 5 si concentra sulla discussione e prospettive future esplorando l'impatto degli Smart Contract sulla decentralizzazione e l'auto-applicazione, delineando il loro ruolo rivoluzionario in settori come la finanza, la sanità e l'energia. Viene presentato un approfondimento sulle sfide e le opportunità future, fornendo una base informativa per ulteriori ricerche e sviluppi nel campo della tecnologia blockchain e degli Smart Contracts.

5.0.1 Discussione

La decentralizzazione e la capacità di auto-applicazione degli Smart Contract, abilitati dalla tecnologia blockchain, stanno rivoluzionando i paradigmi moderni. Questo nuovo modello elimina la necessità di un'autorità centrale o di un server fidato, consentendo l'esecuzione delle regole in una rete peer-to-peer. Questa relazione presenta una ricerca approfondita sugli Smart Contract, esplorando la loro natura, le applicazioni e le implicazioni. Vengono fornite una tassonomia delle soluzioni esistenti e una classificazione della letteratura accademica, offrendo una panoramica completa del campo. Durante il mio periodo di tirocinio, ho focalizzato gli sforzi sul trasferimento di uno Smart Contract precedentemente scritto in Solidity, in un file Go, un linguaggio di programmazione più adatto per l'esecuzione all'interno di un ambiente di una TEE come Intel SGX. Il progetto è stato strutturato in varie fasi, iniziate con un'analisi approfondita del funzionamento dello Smart Contract sulla blockchain Ethereum. Questa fase iniziale è stata cruciale per comprendere appieno la logica e le operazioni svolte dallo Smart Contract nell'ambiente decentralizzato.

Successivamente, ci siamo concentrati sulla conversione del codice Solidity in linguaggio Go utilizzando strumenti come Geth, una delle implementazioni principali del client Ethereum. Abbiamo anche sfruttato altre risorse come Abigen, che si è rivelato particolarmente vantaggioso. Abigen ha semplificato notevolmente il processo di trasformazione, automatizzando la generazione dei wrapper in Go per gli Smart Contract. Questo strumento ha ridotto significativamente il carico di lavoro manuale e ha garantito una maggiore coerenza nel codice convertito.

Un'altra importante fase del lavoro è stata l'esecuzione del codice Go all'interno dell'ambiente della TEE Intel-SGX. Questo ambiente fornisce un livello aggiuntivo di sicurezza, proteggendo il codice eseguito all'interno da accessi non autorizzati e manipolazioni esterne. L'utilizzo di un'immagine Docker per simulare il comportamento dell'ambiente SGX ha permesso di condurre una serie di test e simulazioni per verificare l'efficacia e la sicurezza del sistema implementato.

Il lavoro svolto durante il tirocinio ha rappresentato un importante contributo alla creazione di un ambiente sicuro ed efficiente per l'esecuzione dei contratti intelligenti sulla blockchain Ethereum. Questa esperienza ha permesso di acquisire competenze fondamentali nel campo dello sviluppo blockchain e della sicurezza informatica, aprendo nuove prospettive di ricerca e sviluppo nel settore delle tecnologie distribuite. Inoltre mi ha fornito una visione chiara dei risultati ottenuti e delle sfide affrontate durante lo sviluppo e l'implementazione di soluzioni innovative per l'esecuzione degli Smart Contracts all'interno delle TEEs. In particolare, il progetto mi ha portato a comprendere appieno l'importanza di garantire la sicurezza e la privacy delle transazioni blockchain, nonché a esplorare le potenzialità offerte dalle tecnologie TEE per raggiungere tali obiettivi.

5.0.2 Limitazioni e Lavori Futuri

Si prevede che gli Smart Contract avranno un impatto significativo su vari settori, tra cui la finanza, la sanità e l'energia. Questi strumenti digitali automatizzati hanno il potenziale per semplificare le transazioni, ridurre i costi e aumentare l'efficienza.

Le tendenze future nel campo degli Smart Contract continuano a evolvere, ed è probabile che vedremo nuove applicazioni e miglioramenti degli Smart Contract.

Questo studio fornisce un supporto informativo per coloro interessati alla ricerca degli Smart Contract, offrendo una panoramica completa e dettagliata del campo. Con la continua evoluzione della tecnologia blockchain, gli Smart Contract rappresentano un'area di ricerca promettente e in rapida crescita.

Tuttavia, il progetto presenta alcune limitazioni che ne influenzano la portata e l'applicabilità pratica. In primo luogo, l'utilizzo di una singola TEE può costituire un punto di vulnerabilità potenziale, in quanto la compromissione del dispositivo potrebbe compromettere l'intero sistema. Pertanto, sarebbe auspicabile esplorare l'implementazione di più TEE distribuiti in modo da aumentare la resilienza e la sicurezza complessiva del sistema.

Inoltre, il focus esclusivo sull'ambiente di esecuzione EVM di Ethereum potrebbe limitare le possibilità di adattare la soluzione a diverse blockchain e piattaforme distribuite. Per superare questa limitazione, sarebbe necessario esplorare la compatibilità con altre blockchain e valutare l'utilizzo di linguaggi e tecnologie più adatti alle specifiche esigenze delle diverse piattaforme.

Infine, una valutazione più approfondita dei costi e delle prestazioni su diverse piattaforme blockchain potrebbe offrire una migliore comprensione dei trade-off tra sicurezza, efficienza e scalabilità. Questo consentirebbe di prendere decisioni più informate durante il processo di progettazione e implementazione delle soluzioni blockchain basate sulle TEEs.

Bibliografia

[1] Akash Arora, Kanisk, and Shailender Kumar. Smart contracts and nfts: non-fungible tokens as a core component of blockchain to be used as collectibles. In *Cyber Security and Digital Forensics: Proceedings of ICCSDF 2021*, pages 401–422. Springer, 2022.

[2] Marcella Atzori. Blockchain technology and decentralized governance: Is the state still necessary? *Available at SSRN 2709713*, 2015.

[3] Mehmet Aydar, Salih Cemil Cetin, Serkan Ayvaz, and Betul Aygun. Private key encryption and recovery in blockchain. *arXiv preprint arXiv:1907.04156*, 2019.

[4] Massimo Bartoletti and Livio Pompianu. An empirical analysis of smart contracts: platforms, applications, and design patterns. In *Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers 21*, pages 494–509. Springer, 2017.

[5] Dirk G Baur and Jonathan R Karlsen. Do crypto investors care about energy use and climate change? evidence from ethereum's transition to proof-of-stake. *Evidence from Ethereum's Transition to Proof-of-Stake (January 15, 2024)*, 2024.

[6] Dave Bayer, Stuart Haber, and W. Scott Stornetta. Improving the efficiency and reliability of digital time-stamping. In *Sequences II*, pages 329–334. Springer, 1992.

[7] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, pages 91–96, 2016.

[8] Andrzej Białas. Specification means definition for the common criteria compliant development process–an ontological approach. In *Complex Systems and Dependability*, pages 37–53. Springer, 2012.

[9] Andreas Bogner, Mathieu Chanson, and Arne Meeuw. A decentralised sharing app running a smart contract on the ethereum blockchain. In *Proceedings of the 6th International Conference on the Internet of Things*, pages 177–178, 2016.

[10] Anselm Busse, Jacob Eberhardt, and Stefan Tai. Evm-perf: High-precision evm performance analysis. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–8. IEEE, 2021.

[11] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. 2014.

[12] Vitalik Buterin. A next-generation smart contract and decentralized application platform: Ethereum. 2014.

[13] Vitalik Buterin. On collusion. 2019.

[14] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[15] Somnath Chakrabarti, Thomas Knauth, Dmitrii Kuvaiskii, Michael Steiner, and Mona Vij. Trusted execution environment with intel sgx. In *Responsible Genomic Data Sharing*, pages 161–190. Elsevier, 2020.

[16] Kaylash C Chaudhary, Vishal Chand, and Ansgar Fehnker. Double-spending analysis of bitcoin. In *Pacific Asia conference on information systems*. Association for Information Systems, 2020.

[17] Intel Corporation. Overview of intel sgx enclave. Technical report, Intel, 2022.

[18] Ivan De Oliveira Nunes, Xuhua Ding, and Gene Tsudik. On the root of trust identification problem. In *Proceedings of the 20th International Conference on Information Processing in Sensor Networks (Co-Located with CPS-IoT Week 2021)*, pages 315–327, 2021.

[19] Edgeless Systems. Ego.

[20] Etherscan. Etherscan - ethereum transaction history chart, Year the chart was accessed, 2015-2023.

[21] S Fernández-Vázquez, R Rosillo, D De La Fuente, and P Priore. Blockchain and smart contracts: A revolution. In *International Conference on Industrial Engineering and Operations Management*, pages 29–36. Springer, 2018.

[22] Ethereum Foundation. Proof of stake (pos), 2020.

[23] Yuefei Gao, Shin Kawai, and Hajime Nobuhara. Scalable blockchain protocol based on proof of stake and sharding. *Journal of Advanced Computational Intelligence and Intelligent Informatics*, 23(5):856–863, 2019.

[24] Javier González. *Operating System Support for Run-Time Security with a Trusted Execution Environment*. PhD thesis, 03 2015.

[25] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, 1991.

[26] Ryan Hasting. Smart contracts: Implications on liability and competence. *U. Miami Bus. L. Rev.*, 28:358, 2019.

[27] Daojing He, Zhi Deng, Yuxing Zhang, Sammy Chan, Yao Cheng, and Nadra Guizani. Smart contract vulnerability analysis and security audit. *IEEE Network*, 34(5):276–282, 2020.

[28] Frank Hofmann, Simone Wurster, Eyal Ron, and Moritz Böhmecke-Schwafert. The immutability concept of blockchains and benefits of early standardization. In *2017 ITU Kaleidoscope: Challenges for a Data-Driven Society (ITU K)*, pages 1–8. IEEE, 2017.

[29] Nisha Jacob, Johann Heyszl, Andreas Zankl, Carsten Rolfes, and Georg Sigl. How to break secure boot on fpga socs through malicious hardware. In *Cryptographic Hardware and Embedded Systems–CHES 2017: 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, pages 425–442. Springer, 2017.

[30] Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Trusted execution environments: Properties, applications, and challenges. *IEEE Security Privacy*, 18(2):56–60, 2020.

[31] Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ecdsa). *International journal of information security*, 1:36–63, 2001.

[32] Michał Król, Onur Ascigil, Sergi Rene, Alberto Sonnino, Mustafa Al-Bassam, and Etienne Rivière. Shard scheduler: object placement and migration in sharded account-based blockchains. In *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, pages 43–56, 2021.

[33] Randhir Kumar and Rakesh Tripathi. Secure healthcare framework using blockchain and public key cryptography. *Blockchain Cybersecurity, Trust and Privacy*, pages 185–202, 2020.

[34] Rujia Li, Qin Wang, Qi Wang, David Galindo, and Mark Ryan. Sok: Tee-assisted confidential smart contract. *arXiv preprint arXiv:2203.08548*, 2022.

[35] Lodovica Marchesi, Michele Marchesi, Giuseppe Destefanis, Giulio Barabino, and Danilo Tigano. Design patterns for gas optimization in ethereum. In *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 9–15, 2020.

[36] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 1–9. 2016.

[37] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[38] Peng Qin, Jingzhi Guo, Bingqing Shen, and Quanyi Hu. Towards self-automatable and unambiguous smart contracts: Machine natural language. In *Advances in E-Business Engineering for Ubiquitous Computing: Proceedings of the 16th International Conference on e-Business Engineering (ICEBE 2019)*, pages 479–491. Springer, 2020.

[39] Ruben Recabarren and Bogdan Carbunar. Tithonus: A bitcoin based censorship resilient system. *arXiv preprint arXiv:1810.00279*, 2018.

[40] F Rizal Batubara, Jolien Ubacht, and Marijn Janssen. Unraveling transparency and accountability in blockchain. In *Proceedings of the 20th annual international conference on digital government research*, pages 204–213, 2019.

[41] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/Ispa*, volume 1, pages 57–64. IEEE, 2015.

[42] T Takenobu. Ethereum evm illustrated. *Github Pages*, 2018.

[43] Maddipati Varun, Balaji Palanisamy, and Shamik Sural. Mitigating frontrunning attacks in ethereum. In *Proceedings of the Fourth ACM International Symposium on Blockchain and Secure Critical Infrastructure*, pages 115–124, 2022.

[44] Oleksandr Vashchuk and Roman Shuwar. Pros and cons of consensus algorithm proof of stake. difference in the network safety in proof of work and proof of stake. *Electronics and Information Technologies*, 9(9):106–112, 2018.

[45] Yunsen Wang and Alexander Kogan. Designing confidentiality-preserving blockchain-based transaction processing systems. *International Journal of Accounting Information Systems*, 30:1–18, 2018.

[46] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. sgx-perf: A performance analysis tool for intel sgx enclaves. In *Proceedings of the 19th International Middleware Conference*, pages 201–213, 2018.

[47] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.

[48] Rui Yuan, Yu-Bin Xia, Hai-Bo Chen, Bin-Yu Zang, and Jan Xie. Shadoweth: Private smart contract on public blockchain. *Journal of Computer Science and Technology*, 33:542–556, 2018.

[49] Efpraxia Zamani, Ying He, and Matthew Phillips. On the security risks of the blockchain. *Journal of Computer Information Systems*, 60(6):495–506, 2020.

[50] Wei Zheng, Ying Wu, Xiaoxue Wu, Chen Feng, Yulei Sui, Xiapu Luo, and Yajin Zhou. A survey of intel sgx and its applications. *Frontiers of Computer Science*, 15:1–15, 2021.

[51] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Lutan Zhao, Fengkai Yuan, Peinan Li, Zhongpu Wang, Boyan Zhao, et al. Enabling privacy-preserving, compute-and data-intensive computing using heterogeneous trusted execution environment. *arXiv preprint arXiv:1904.04782*, 2019.